Shady Documentation

Release 1.9.1

Jeremy Hill, Scott Mooney

Jun 20, 2019
# Table of Contents

1 To download and install the Python package: 3

2 To get started: 5

3 To learn more: 7

4 To ask questions: 9

5 To cite Shady: 11

5.1 Overview ................................................................. 11

5.2 Setup ................................................................. 13

5.2.1 Installing the Shady Package .................................... 13

If you already have a Python environment that you use for development ...................................... 13

If you are new to Python development ................................................. 13

Using the right Python ......................................................... 14

Updating your Shady installation .................................................. 15

5.2.2 Compatibility ............................................................ 15

Hardware and Operating Systems ................................................. 15

Python and Third-Party Python Packages ........................................ 16

Known issues ..................................................................... 17

“Legacy” vs “Modern” OpenGL and random noise generation .................... 17

5.2.3 The Binary “Accelerator” Component (ShaDyLib) .................. 18

Windowing and Rendering ....................................................... 18

Building the Accelerator from Source ............................................ 19

5.3 Getting Started .............................................................. 21

5.3.1 Creating a World ....................................................... 21

Running single-threaded .......................................................... 21

Running the Shady engine in a background thread (Windows only) ............ 23

Multi-threaded operation on non-Windows platforms ............................... 23

Limitations on multi-threaded performance in Python ............................ 24

5.3.2 Command-Line Options ............................................... 24

5.3.3 Example Scripts ......................................................... 25

List of examples ................................................................. 25

examples/animated-textures.py ............................................... 25

examples/artifacts.py ........................................................ 30

examples/capture-video.py ..................................................... 32

examples/custom-functions.py ................................................. 35
<table>
<thead>
<tr>
<th>Example</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples/dithering.py</td>
<td>41</td>
</tr>
<tr>
<td>examples/dots1.py</td>
<td>45</td>
</tr>
<tr>
<td>examples/dots2.py</td>
<td>47</td>
</tr>
<tr>
<td>examples/dots3.py</td>
<td>53</td>
</tr>
<tr>
<td>examples/dots4.py</td>
<td>55</td>
</tr>
<tr>
<td>examples/dynamic-range.py</td>
<td>59</td>
</tr>
<tr>
<td>examples/dynamics1.py</td>
<td>63</td>
</tr>
<tr>
<td>examples/dynamics2.py</td>
<td>67</td>
</tr>
<tr>
<td>examples/events.py</td>
<td>72</td>
</tr>
<tr>
<td>examples/fancy-hardware.py</td>
<td>76</td>
</tr>
<tr>
<td>examples/foreign-stimulus.py</td>
<td>85</td>
</tr>
<tr>
<td>examples/interactive-gamma.py</td>
<td>90</td>
</tr>
<tr>
<td>examples/noise.py</td>
<td>92</td>
</tr>
<tr>
<td>examples/precision.py</td>
<td>94</td>
</tr>
<tr>
<td>examples/sharing.py</td>
<td>98</td>
</tr>
<tr>
<td>examples/showcase.py</td>
<td>105</td>
</tr>
<tr>
<td>examples/tearing.py</td>
<td>112</td>
</tr>
<tr>
<td>examples/text.py</td>
<td>114</td>
</tr>
<tr>
<td>examples/video.py</td>
<td>118</td>
</tr>
<tr>
<td>examples/world.py</td>
<td>123</td>
</tr>
</tbody>
</table>

5.4 Key Concepts ........................................... 124
   5.4.1 Shader Pipeline Details .......................... 124
   5.4.2 Precise Control of Timing ........................ 126
      General system settings ............................ 126
      Vertical sync issues .............................. 126
      Optimization ..................................... 127
         Diagnostic tools ............................... 127
         Shady usage tips .............................. 127
   5.4.3 Gamma Correction, Dynamic Range Enhancement, and the Canvas .......................... 128
      Overview ......................................... 128
      Gamma Correction ................................ 129
      Dynamic Range Enhancement ....................... 130
      Look-up Table ................................... 131
      Canvas .......................................... 133
      Avoiding image artifacts ......................... 133
   5.4.4 Luminance and Contrast ............................ 134
      Definitions ...................................... 134
      Conversion utility functions .................... 136
      Contrast-ratio computation functions (ideal or physical) .......................... 136
   5.4.5 Managed Properties and Managed Shortcuts ........ 136
      Managed Properties .............................. 136
      Managed Shortcuts ............................... 138
      Unmanaged Dynamic Properties .................... 139
      List of Managed Properties for the World Class ........................................... 139
      List of Managed Properties for the Stimulus Class ....................................... 140
   5.4.6 Making Properties Dynamic ......................... 141
      Individual Properties ........................... 141
      The Animate Method ................................ 143
      Order of Dynamic Evaluations .................... 145
   5.4.7 Property Sharing ................................ 145
5.5 Shady API Reference ................................. 147
   5.5.1 Shady Package ................................ 147
      The World Class .................................. 147
      The Stimulus Class ................................ 164
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global Functions and Constants</td>
<td>189</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Shady.Dynamics Sub-module</td>
<td>192</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Shady.Linearization Sub-module</td>
<td>200</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Shady.Contrast Sub-module</td>
<td>203</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Shady.Utilities Sub-module</td>
<td>205</td>
</tr>
<tr>
<td>5.5.6</td>
<td>Shady.Text Sub-module</td>
<td>215</td>
</tr>
<tr>
<td>5.5.7</td>
<td>Shady.Video Sub-module</td>
<td>217</td>
</tr>
<tr>
<td>5.6</td>
<td>License</td>
<td>218</td>
</tr>
<tr>
<td>6</td>
<td>Indices and tables</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>223</td>
</tr>
</tbody>
</table>
Welcome to Shady. Shady is for programmers who work in neuroscience, especially vision science. It is a two-dimensional graphics engine and Python framework, for generating and presenting psychophysically-accurate visual stimuli, and for manipulating them in real time with minimal CPU usage and minimal “boilerplate” coding. For an outline of Shady’s structure and features, see the “Overview” topic documentation.
To download and install the Python package:

• The short answer: `python -m pip install shady`
• The long answer: see the “Installing the Shady Package” topic documentation.
Chapter 1. To download and install the Python package:
To get started:

- To run an interactive tour of the main features: `python -m Shady demo showcase`
- To get more information about that demo (such as the command-line options it supports): `python -m Shady help showcase`
- To see a list of other similar demos: `python -m Shady list`
Chapter 2. To get started:
To learn more:

- Class and method docstrings—e.g. `help(Shady.World)` or `help(Shady.Screens)` from within Python (or better, if you are using IPython: `Shady.World?` etc.)
- For topic documentation: docstrings of objects inside the `Shady.Documentation` submodule—e.g. `help(Shady.Documentation.PreciseControlOfLuminance)`
- It’s also all on [https://shady.readthedocs.io](https://shady.readthedocs.io)
To ask questions:

- You can ask programming questions on stackoverflow, using the [shady] tag (https://stackoverflow.com/questions/tagged/shady)
- For other questions and support issues, there’s an online forum at https://groups.google.com/d/forum/shady-project
- If an issue is confirmed to be a technical problem or bug, you can submit details at https://bitbucket.org/snapproject/shady-hg/issues
Chapter 4. To ask questions:
CHAPTER 5

To cite Shady:

When reporting any study in which you used Shady, please cite:


A pre-print is also available on https://shady.readthedocs.io

5.1 Overview

Shady is a two-dimensional graphics engine and Python programmer’s toolkit, designed to make stimulus display easy in neuroscience—especially vision science.

It enables precise presentation of visual stimuli even on mass-produced screens and video cards, without additional specialized hardware. It allows you to render arbitrary stimulus patterns linearly with high dynamic range and high timing precision. It lets you manipulate them in real time while leaving the CPU maximally free to perform other time-critical tasks.
At its core, Shady is a customizable, specialized “shader” program, written in GLSL, that runs on the graphics processor (GPU). An “engine”, compiled from C++, performs the frame-by-frame OpenGL calls required to drive the shader. The Python package provides high-level wrappers and an API for controlling the shader via the engine (it also contains a fallback implementation of the engine in pure Python, but for performance reasons we do not recommend using this).

Shady provides the following functionality:

- lets you take control of your display screen, ensuring that you have precise control over every physical pixel, despite the rescaling tricks modern operating systems try to do;
- handles gamma-correction and dynamic-range enhancement of stimuli automatically and transparently;
- provides simple hooks for injecting your own small segments of GLSL code, to define custom signal functions, windowing functions and contrast modulation functions, as well as any custom parameters they require;
- eliminates “boilerplate” coding and minimizes the length of your program: it takes one line of code to initialize the system, open a full-screen window and start the rendering engine, one line to create a stimulus, and (often) only one line to define how the stimulus should be animated;
- provides a surrounding ecosystem of:
  - ancillary functions (e.g. functions for computing/converting contrast ratios and visual angles)
  - high-level diagnostic tools: test patterns, timing performance checks, interactive perceptual linearization, image and video capture, …
  - flexible tools for controlling stimulus behavior in time: state machines, general-purpose function objects that can be manipulated arithmetically, as well as specialized functions that integrate, smooth, oscillate, self-terminate, …

Shady was conceived to play a modular role in larger, more complex multi-modal neuroscience applications. As such, it avoids dominating or constraining the user’s programming environment. Rather, it aims to be a “good citizen” in three ways:

- Maximize compatibility: Shady’s Python, C++ and GLSL code has been confirmed to run correctly on recent versions of Windows, macOS and Ubuntu Linux. Beyond the operating system itself, no proprietary software is required to run or to develop with Shady. It is fully compatible with both Python 2 and Python 3. It does not require third-party packages over and above the base Python system. It does leverage a few highly prevalent packages (numpy, pillow, matplotlib, ipython, opencv-python) to expand and improve its functionality where possible. However, we have designed Shady to be tolerant of variation in these packages’ version numbering, and to degrade gracefully even in their absence.
- Embrace Windows: support for Windows is not a grudgingly provided afterthought. Windows is our primary platform for performance optimization, because it is the platform on which we expect to find most prevalent support for specialized neuroscientific hardware (eye-trackers, EEG amplifiers, etc.) as well as novel human interface devices. Therefore, it is the platform on which we expect to see the greatest development of integrated multi-modal neuroscience applications, and hence the place where Shady has most to offer, as a modular component of such systems.
- Minimize CPU load: the Shady engine uses the GPU for as much as possible of the frame-by-frame stimulus generation and processing, leaving the CPU free for other tasks like real-time biosignal processing.

To minimize CPU load, the shader pipeline ensures that the following steps are handled by the GPU on every frame:

1. Generation of a carrier pattern (pre-computed texture, and/or procedurally-generated signal, and/or color);
2. Translation and/or rotation and/or scaling of the carrier pattern (NB: this may create interpolation artifacts if the carrier contains a pre-computed texture);
3. Contrast effects: overall scaling, and/or spatial windowing, and/or other (customizable) spatial modulation functions;
4. Repositioning and/or rotation and/or scaling of the complete stimulus (NB: rotation and scaling may create interpolation artifacts, in any stimulus);

5. Optional addition of dynamic Gaussian or uniform noise;

6. \textit{Gamma-correction and dynamic-range enhancement}, by one of a number of possible strategies (some requiring specialized equipment, some not);

7. Quantization down to native precision of the graphics card.

See the \textit{Shady.Documentation.Pipeline} topic documentation for a more detailed breakdown of these steps.

\section*{5.2 Setup}

\subsection*{5.2.1 Installing the Shady Package}

\begin{Verbatim}
\begin{verbatim}
python -m pip install shady
python -m Shady demo showcase
\end{verbatim}
\end{Verbatim}

Shady is a \textbf{Python} package. Python is a powerful, intuitive language, although it can be \textit{difficult to install}. Fortunately, it is worth the one-time effort required to set up.

Out of the box, Shady should work on typical modern Windows and macOS systems running either Python 2.7 or Python 3.4+. Other systems may be supportable with a little extra effort. Real-time performance is optimized on Windows; anywhere else, your mileage may vary more. See the \textit{Compatibility} documentation for more details.

\textbf{If you already have a Python environment that you use for development}

The primary and recommended way to get Shady is to use the \texttt{pip} package manager. You should already have this, if your Python version is reasonably up-to-date (Python 2 versions 2.7.9 and up, or Python 3 versions 3.4 and up). If not, see the \textit{pip installation instructions}. Assuming you have \texttt{pip}, the following command will install Shady:

\begin{Verbatim}
\begin{verbatim}
python -m pip install Shady
\end{verbatim}
\end{Verbatim}

This will automatically download the latest distribution from its home at \url{http://pypi.python.org/pypi/shady}, and install it. It will also automatically install the \textit{recommended third-party packages} (\texttt{numpy}, \texttt{pillow}, \texttt{matplotlib}, and \texttt{ipython}) if you do not already have them. (If you do not want this to happen, you can explicitly prevent it by adding the \texttt{--no-dependencies} flag.)

Test your Shady installation using the interactive script \texttt{examples/showcase.py}:

\begin{Verbatim}
\begin{verbatim}
python -m Shady demo showcase
\end{verbatim}
\end{Verbatim}

\textbf{If you are new to Python development}

macOS and Linux distributions come with Python included. However, since other parts of the OS rely on it, your system manages the packages that it contains. Therefore, for your own development work, it is advisable to have a completely separate installation of Python. For scientists, the \texttt{Anaconda} distribution is a good choice.

On Windows, Python doesn't ship as part of the operating system, but there are many ways to install it. If you find that you already have a Python installation, no integral part of the OS itself will be relying on it, but you should still
Follow these instructions if you just want to get going with Shady as quickly as possible and aren’t familiar with the many ways to install Python.

1. Install an Anaconda distribution of Python. We recommend installing a 64-bit Python 3 (at the time of writing the latest release is 3.7). But if you have specific reasons for needing Python 2, Shady is also compatible with Python 2.7.

   **OR:**

   If you want to avoid installing Anaconda’s giant 3 GB library of third-party packages, you can download the 50 MB bare-bones Miniconda version instead, and install Shady’s *recommended third-party packages* yourself (see below).

2. Launch the Anaconda Prompt, which is simply a command prompt that starts off inside your new Python environment.

3. If you installed the bare-bones Miniconda, install Shady’s four basic recommended dependencies yourself, using the following command:

   ```
   python -m conda install numpy pillow matplotlib ipython
   ```

4. (Optional) If you want to use Shady’s video playback and recording features, you will need to install one more package, **opencv**, which (at the time of writing) is available via conda for some Python versions:

   ```
   python -m conda install opencv
   ```

   but if that is unavailable or reports a conflict, you can fall back to using the more standard Python package manager, **pip** (note the change to the package name):

   ```
   python -m pip install opencv-python
   ```

5. Install Shady, also using **pip**:

   ```
   python -m pip install Shady
   ```

6. That’s it! Experience some of Shady’s numerous features with the interactive script `examples/showcase.py` by typing:

   ```
   python -m Shady demo showcase
   ```

**Using the right Python**

As explained above, any system may have multiple Python distributions. And any Python distribution may have multiple “virtual environments” which are separate silos into which you can install independent sets of packages. Anaconda distributions even have the ability to manage and switch between different versions of the Python interpreter itself in different environments.

All of this means that, before you type `python` at a system command prompt, you should take care to ensure that it will launch the version/configuration of Python that you intend. (This in turn means that there is no one-size-fits-all set of instructions for installing a given Python package, which explains why this page is so long…)

Anaconda distributions come with a script called `activate` that allows you to deal with this issue. You would call this once at the beginning of your console session, and it will configure your `PATH` and other environment variables for
the remainder of the session, such that the Anaconda version of Python answers when you call `python`. On Windows that looks like:

```
> call C:\PATH\TO\ANACONDA\Scripts\activate.bat
```

and on others:

```
$ source /PATH/TO/ANACONDA/bin/activate
```

By default this puts you in an Anaconda environment called `root`. But if you have set up other environments, you can pass the name of another environment to the `activate` script in order to switch to it.

On Windows, the even-simpler way is just to double-click on the shortcut to the Anaconda Prompt, which will open a console window and perform the `activate` step automatically. We recommend using the Anaconda Prompt whenever you use Python interactively.

If you are using a non-Anaconda distribution of Python, you may have to roll your own solution for configuring the `PATH` variable appropriately.

### Updating your Shady installation

If you installed with `pip` as above, then the following command is recommended for updating:

```
python -m pip install --upgrade Shady --no-deps
```

(you can omit `--no-deps` if you don’t mind it also upgrading Shady’s various third-party dependencies).

### 5.2.2 Compatibility

#### Summary

- We optimize Shady’s performance on Windows. It works on other operating systems, but there your mileage will vary more widely.
- Shady works with Python 2 or 3.
- Recommended third-party packages `numpy`, `pillow`, `matplotlib`, and `ipython` are not strictly required, but they improve functionality. By default, they will be installed automatically if you install Shady via `python -m pip install shady`.
- In addition, `opencv-python` allows reading/saving of video files.

#### Hardware and Operating Systems

Shady was conceived to play a modular role in larger, more complex multi-modal neuroscience applications. These may include novel human interface devices and/or specialized neuroscientific equipment, such as eye-trackers and EEG amplifiers. Manufacturers of such equipment are overwhelmingly more likely to support Windows than anything else.

Hence, the Windows platform is where we aim to optimize performance, and most of our experience in doing so has been with Windows 10. On other platforms, it should not be any harder to get Shady running, but it may be harder to get it to perform really well. Therefore, we had better describe our support for non-Windows platforms as “experimental”. However, our experiences so far (with macOS 10.9 through 10.13, and with Ubuntu 14 in a VirtualBox)
indicate that both the C++ code and CMake files for the *accelerator*, and the Python code of the rest of the module, are cross-platform compatible.

Shady probably will not work on big-endian hardware. Since most commercial CPUs are little-endian, at least by default, we have had no opportunity to test it on big-endian systems and little interest in doing so.

**Python and Third-Party Python Packages**

Scientific software packages in Python have an unfortunate tendency to rely on a “house of cards” made up of specific versions of other third-party packages. Somewhere in the hierarchy of dependencies, sooner or later you end up locked into a legacy version of something you don’t want. With this in mind, we limited Shady’s dependencies to a small number of well established, very widely used, and actively developed general-purpose packages. We test it with 5-year-old versions of its principal dependencies as well as current versions. We also take care to ensure that Shady’s functionality degrades gracefully even in their absence.

Shady supports Python versions 2 and 3 (specifically CPython, which is the standard, most prevalent implementation). Shady doesn’t have *hard* dependencies on third-party packages beyond that. On any CPython implementation of version 2.7.x, or 3.4 and up, some of Shady’s core functionality should be available. This claim comes with two caveats.

The first caveat is that we are assuming availability of the ShaDyLib *accelerator* which is a compiled binary (dynamic library). Compiled binaries for 32-bit Windows, 64-bit Windows and 64-bit macOS (10.9+) are bundled as part of the Shady distribution. If you are using a different OS (e.g. some flavour of Linux, or macOS version earlier than 10.9) or if the dynamic library fails to load for any reason (some form of dynamic-library dependency hell, no doubt) then you may need to (re)compile the accelerator. *Without* the accelerator, you can still run Shady, but its timing performance will be more inconsistent and generally worse, and you will need to install another third-party package named *pyglet* for Shady to use as a graphics backend (*python -m pip install pyglet*).

The second caveat is that there are a few *recommended* third-party packages, without which Shady’s functionality is relatively limited. Without any third-party packages at all, you can display rectangular or oval patches, with or without a customizable 2-D carrier signal function, a 2-D contrast modulation function, a spatial windowing function, colour, and dynamic pixel noise. You will also have *powerful tools* for governing the way these properties change over time. The following third-party packages, if available, add specific extra types of functionality over and above the core:

```
numpy:
    • Create or render arbitrary textures defined as pixel arrays.
    • Subtle, powerful improvements to existing functionality—for example, dynamic objects like `Shady.Dynamics.Integral` can be multi-dimensional (see `examples/dots4.py`).

pillow:
    • Load texture data from common image formats on disk (also requires `numpy`).
    • Render text stimuli, in a mono-spaced font (also requires `numpy`).
    • Save screen capture data to disk in common image formats.

matplotlib:
    • Render text stimuli in any of the fonts installed on your system (also requires `numpy` and `pillow`).
    • Plot timing diagnostics and image histograms.

opencv-python:
    • Requires `numpy`.
    • Save stimulus sequences as movie files.
    • Display stimuli from video files or live camera feeds.
```
• Write live camera feeds to disk in common video formats.

**ipython**:

• Improve interactive configuration of Shady stimuli.
• Improve user experience at the command prompt (e.g. tab completion, dynamic object introspection, cross-session command history).

With the exception of opencv, these packages are extremely prevalent, used in every conceivable type of scientific application, all around the world. They will get installed by default when you say `python -m pip install shady` (although, if you have an Anaconda installation, you may prefer to first ensure they’re installed via conda rather than letting pip do it). opencv is a more special-purpose package, so we leave it to you to install it if you want it.

To install everything in a minimal “Miniconda” environment:

```bash
python -m conda install numpy pillow matplotlib ipython
python -m pip install shady
python -m conda install opencv
```

Depending on versions, opencv may or may not be available via conda—if not, you can use pip. If your Python distribution is not Anaconda-flavored, you can let pip do everything:

```bash
python -m pip install shady
python -m pip install opencv-python
```

**Known issues**

**“Legacy” vs “Modern” OpenGL and random noise generation**

Shady is based on OpenGL, and makes extensive use of OpenGL Shading Language (GLSL) to implement pixel processing in parallel on the graphics card. OpenGL/GLSL come in two flavours: “modern” means OpenGL version (and corresponding GLSL version) 3.3 and up, whereas “legacy” refers to earlier versions. The drawing commands under the hood are different in these two different contexts, and the features supported by a given graphics card may also differ between the two contexts.

Different operating systems and different graphics drivers may handle the schism differently. On Windows, it is common for the default OpenGL context to be a permissive “best-of-both-worlds” setting in which both old and new features are available. On macOS, by contrast, you must commit to a given version and cannot use legacy and modern features within the same program.

Shady is written to be compatible with both. By default, legacy compatibility is disabled on non-Windows platforms, where an OpenGL 3.30/GLSL 3.30 context will be created unless you specify otherwise. This is to improve the quality of the GLSL random-number generator used for dithering and noise generation. If for some reason you need to mix legacy OpenGL code into your stimulus display (as we do in `examples/foreign-stimulus.py`) then you should create your `Shady.World()` with the constructor argument `legacy=True`, thereby instructing Shady to revert to the bare bones of OpenGL 2.1/GLSL 1.2. (NB: to add your own custom OpenGL code, you will need to install either PyOpenGL or pyglet to gain access to the OpenGL API in Python.)

On any system, `legacy=False` makes Shady switch, under the hood, to modern drawing commands that should be more compatible with future graphics cards. On Windows, you might not notice any difference. On other platforms, specifying `legacy=False` also causes the default value of another option, `openglContextVersion`, to be changed from 0 to 330.

On any system, a non-zero `openglContextVersion` value will force `legacy=False`, and also cause Shady to create an OpenGL context of the specified version (330 requests OpenGL 3.3.0, 410 requests OpenGL 4.1.0, and so on). This will disable support for legacy features: depending on your graphics card and its driver version you may
find, for example, that you no longer have control of the thickness of lines that you draw, or the smoothing of points. So far in our experience of typical Shady use-cases, there is no advantage to doing this on Windows systems, but on macOS this is the price one has to pay for improving the quality of the random-number generation—macOS users compare:

```
python -m Shady noise --legacy=True
python -m Shady noise --legacy=False
```

### 5.2.3 The Binary “Accelerator” Component (ShaDyLib)

#### Summary

Shady comes with an “accelerator”—a dynamic library that greatly improves its performance. The Shady Python package installation includes pre-built .dll files for 32-bit Windows and 64-bit Windows, and a .dylib file for macOS 10.9+. If your platform is not supported by these included binaries, you can build the accelerator yourself, from our C++ sources.

Shady works by harnessing the graphics processor (GPU) to perform, in parallel, most of the pixel-by-pixel operations entailed in signal generation, contrast modulation, windowing, linearization and dithering. For most stimulus arrangements, this leaves relatively little for the CPU to do on each frame: it just has to issue the OpenGL commands that clear the screen and then, for each stimulus in turn, transfer a small set of variable values from CPU to GPU. Nonetheless this may amount to a few hundred separate operations per frame—we’ll call these the “CPU housekeeping” operations (and we’ll consider them separate from the optional further computations that can be performed between frames to animate stimuli).

The CPU housekeeping is performed by an “engine”. The earliest versions of Shady implemented the engine in pure Python. Some of the time, this worked fine, but it was prone to sporadic frame skips. So we transitioned to using the “accelerator”, which is a binary engine compiled from C++ and packaged as a dynamic library called ShaDyLib. Some binary builds of ShaDyLib are included with the Shady Python package: .dll files for 32- and 64-bit Windows, and a .dylib file for 64-bit macOS 10.9 (Mavericks, 2013) and up. These are used by default where possible. To make the corresponding .so file for Linux (or the .dylib for macOS versions between 10.4 through 10.8), you would need to build it yourself from the C++ sources (see below).

The pure-Python engine is still included (as the Shady.PyEngine submodule) and it is used automatically as a fallback when the accelerator is not available. But it is much better to use the accelerator (and to attempt to compile the accelerator from source if the dynamic library for your platform is not already part of Shady). The problem is that Python, being a high-level dynamic interpreted language, is inefficient for performing large numbers of simple operations. Relative to the equivalent operations compiled from C++, Python not only requires extra time but, critically, adds a large amount of variability to the time taken when running on a modern operating system that tends to perform a lot of sporadic background tasks (the effect of these is especially noticeable on Windows). Hence the frequent sporadic frame skips when using the PyEngine, which become much rarer when you use the accelerator.

#### Windowing and Rendering

These are two separate issues:

**Windowing** is about creating a window and an associated OpenGL context, synchronizing the double-buffer flipping with the display hardware’s frame updates, and handling events such as keyboard and mouse input.

**Rendering** is about the OpenGL calls that comprise most of the frame-by-frame CPU housekeeping. Rendering is implemented in a windowing-independent way (i.e. without reference to the windowing environment or its particular implementation).
The ShadyLib accelerator provides independent implementations of both windowing (using a modified GLFW C library) and rendering. Assuming you have the accelerator, you have three options:

1. Use the accelerator for both windowing and rendering. When the accelerator is available, this is the default option, and is highly recommended for performance reasons.

2. Use a different windowing environment (such as pyglet) and still use the accelerator for rendering. There is no great advantage to doing this, and there are disadvantages here and there (e.g. failure to take full advantage of Mac Retina screen resolution).

3. Do not use the accelerator at all. Fall back to the PyEngine, which requires a third-party package to expose the necessary OpenGL calls. Either pyglet or PyOpenGL will work for this (pyglet is probably the better choice because, in the absence of the accelerator, you will also need it for windowing anyway). As explained above, this option is not recommended if you can avoid it.

The `Backend()` function allows you to change the default windowing and rendering implementations.

**Building the Accelerator from Source**

As we mentioned above, binaries are included in the Shady download, for 32-bit Windows, 64-bit Windows, and for macOS 10.9+. Therefore, we hope you will not need to build the accelerator from source. However, if you need to do so for any reason (for example to support Linux, or macOS 10.4-10.8) then it should be relatively easy. (“Should” is every engineer’s most heavily loaded word.)

You can obtain the complete Shady source code from the master Mercurial repository which is hosted on Bitbucket:

```
hg clone https://bitbucket.org/snapproject/shady-hg
```

Or you can go to that URL with a browser and clickety-clickety-download-unzippety if you really must. But there are several advantages to installing Mercurial and then managing things with the `hg` command from within the `shady-hg` directory. For example, it makes it very easy to get our latest updates:

```
hg pull --update
```

or to switch between bleeding-edge code:

```
hg update default
```

and the latest released version:

```
hg update release
```

Your working-copy of the repository will include a copy of the Shady package itself, inside the `python` subdirectory. You can “install” this copy as your default Shady package if you want: first change your working directory so that you’re in the root of the working-copy (i.e. the place that contains `setup.py`) and then call:

```
python -m pip install -e .
```

The `-e` flag stands for “editable copy” and this type of “installation” does not actually copy or move any files. Instead, it merely causes whichever Python distribution you just invoked to make a permanent record of the location of the appropriate directory, thereby ensuring that it is found when you say `import Shady` in subsequent sessions.

Your working-copy of the repository will also include the `accel-src` directory tree which contains the C++ sources for the accelerator. To build these, you need to have CMake installed (version 3.7+) as well as a C++ compiler. On Windows, the compiler we use is Visual C++, installed as part of a free (“Express” or “Community”) edition of Visual Studio 2012 or later. On macOS, we use gcc installed from the “XCode Command Line Tools” package (we don’t need the full-blown XCode).
The script `accel-src/devel/build/go.cmd` can be run from a Windows Command Prompt or from a bash command-line (e.g. from the “Terminal” app on macOS) and will run the entire CMake + build process. If you’re on Windows, and either your OS or your Python distribution is 32-bit, then you need to explicitly say `go.cmd Win32`. Further details are provided in the comments at the top of the `go.cmd` script.

The accelerator has two third-party dependencies: GLEW and GLFW. GLEW is provided as source. Binary builds of GLFW (slightly modified) are also provided in the repository. If for any reason you need to rebuild that GLFW library, see the instructions in `accel-src/devel/glfw-3.2.1/build-notes.txt`.

On Linux, we also found it necessary to install various developer tools, libraries and headers. Here is our script for setting up our development environment for Shady, on the basis of a fresh installation of Ubuntu 18.x LTS for Desktops:

```bash
sudo apt-get update
sudo apt-get install \\
  mercurial cmake g++  # essentials for versioning Shady and building ShaDyLib`\\
  libglu1-mesa-dev libxrandr-dev libxi-dev libxcursor-dev libxinerama-dev  # libraries required for building ShaDyLib`\\
  curl libudev-dev libtool autotools-dev automake pkg-config  # build tools and libraries required for libusb build (part of dpxmode build)`\\
  python pip python-tk  # Python 2 basics`\\
  python3-pip python3-tk  # Python 3 basics`
};
sudo pip install numpy matplotlib ipython pillow opencv-python pyglet pyserial  # Python 2 third-party packages
sudo pip3 install numpy matplotlib ipython pillow opencv-python pyglet pyserial  # Python 3 third-party packages

# get Shady
mkdir -p ~/code
cd ~/code
hg clone https://bitbucket.org/snapproject/shady-hg
cd shady-hg

# "install" Shady as an editable package
sudo pip install -e.
sudo pip3 install -e.

# build the accelerator
./accel-src/devel/build/go.cmd

# build and incorporate the mode-changer utility for the ViewPixx monitor
./dpxmode-src/build.cmd
./dpxmode-src/release.cmd

# In addition, to use Shady on the primary screen, we had to auto-hide the
# Ubuntu dock (Applications -> Settings -> Dock -> Auto-hide the Dock) and
# and the top bar (search for and install the "Hide Top Bar" extension)
```

A successfully built shared library will end up in the `accel-src/release/` directory. What do you do with it then? Well:

- If you are using the repository copy of the Shady Python package (i.e. you have performed `python -m pip install -e` as described above, or you are working in the `python` directory next-door to `accel-src` when you start Python) then Shady will be smart enough, by default, to look for the accelerator in `/accel-src/release/` and to prefer it over any copy that it finds “bundled” in its own package direc-
You can also explicitly control which version it prefers, by supplying either `acceleration='devel'` or `acceleration='bundled'` as a keyword argument, either to `Shady.BackEnd()` or to the `Shady.World()` constructor.

- You can verify which version of the accelerator is being loaded by looking under ShaDyLib in the output of the `ReportVersions()` method of an instantiated `World`, or failing that the global `Shady.ReportVersions()` function.
- Finally, maybe you would like to move the newly-built shared library into the “bundled” location within the accompanying Shady package directory? If so, you can run `python accel-src/devel/build/release.cmd`. This will copy all the relevant material from `accel-src/release/` into the `python/Shady/accel` subdirectory, and remove the dynamic libraries from `accel-src/release/`.

## 5.3 Getting Started

The best way to learn about Shady is to use our `Example Scripts` as interactive tutorials. The best one to start with is `python -m Shady demo showcase`. Others can be listed with `python -m Shady list`. The topics below go into more detail about the different ways of starting Shady:

### 5.3.1 Creating a World

Shady stimulus displays revolve around an object called the `World`. Most Shady applications would begin by creating a `World` instance. There are two ways of designing your application around the `World`: either run everything in a single thread, or allow Shady’s graphical operations to happen in one thread while continuing to work in another. The multi-threaded way is our preferred approach, particularly as it allows the programmer to construct and refine stimuli interactively during the design and implementation of an application.

- **Running single-threaded**
- **Running the Shady engine in a background thread (Windows only)**
- **Multi-threaded operation on non-Windows platforms**
- **Limitations on multi-threaded performance in Python**

### Running single-threaded

Here’s an example of how you can use Shady in a single-threaded way:

```python
import Shady

w = Shady.World( threaded=False )
# This may (depending on platform) open a window already, but
# if so it will be inactive.

s = w.Stimulus(
    signalFunction = Shady.SIGFUNC.SinewaveSignal,
    signalAmplitude = 0.5,
    plateauProportion = 0.0,
    atmosphere = w,
)
# create a Stimulus...
```

(continues on next page)
In the above example, World construction, rendering, and all animation and event-handling callbacks happen in the main thread. You should not try to type the above commands line-by-line into an interactive prompt, because the second line may (on some platforms) create a frozen full-screen window that may then obscures your console window and, because it is not processing events, may not respond to your attempts to alt-tab away from it.

A slightly different way to organize the above would be to put the stimulus-initialization code in the Prepare() method of a World subclass:

```python
import Shady

class MyWorld( Shady.World ):

    def Prepare( self, speed=50 ):
        self.Stimulus(
            signalFunction = Shady.SIGFUNC.SinewaveSignal,
            signalAmplitude = 0.5,
            plateauProportion = 0.0,
            cx = Shady.Integral( speed ),
            atmosphere = self,
        )

    def Animate( self, t ):
        # ... the `.Animate()` method will be used as the
        # animation callback unless you replace it using the
        # `@w.AnimationCallback` decorator or (equivalently) the
        # `w.SetAnimationCallback()` method.
        pass

w = MyWorld( threaded=False, speed=16 )
# The `speed` argument, unrecognized by the constructor, is simply
# passed through to the `.Prepare()` method (the prototype for
# which may have any arguments you like after `self`).

w.Run()
# As before, because the `World` was created with `threaded=False`,
# the window will be inactive until you call `.Run()`.
```
Running the Shady engine in a background thread (Windows only)

The following has worked nicely for us on Windows systems:

```python
import Shady
w = Shady.World()  # threaded=True is the default
# the `World` starts rendering and processing events immediately,
# in a background thread

w.Stimulus( sigfunc=1, siga=0.5, pp=0, cx=Shady.Integral(50) ), atmosphere=w )
# thread-sensitive operations like this are automatically deferred
# and will be called in the `World`'s rendering thread at the end
# of the next frame.

@w.AnimationCallback
def DoSomething( t ):
    # ... you can set set the animation callback as before, if
    # you need one (with or without the `self` argument)
    pass
```

In this case, a synchronous call to `w.Run()` is optional: all that would do is cause your main thread to sleep until the World has finished.

This relies on using the binary “ShaDyLib” accelerator as the `Shady.Rendering.BackEnd()`. Without the accelerator (using, for example, pyglet as the back-end) you may find that some functionality (such as keyboard and mouse event handling) does not work properly when the `Shady.World` is in a background thread.

It also relies on Windows. On other platforms, the graphical toolkit GLFW, which underlies the ShaDyLib windowing back-end, insists on being in the main thread (nearly all windowing/GUI toolboxes seem to do this). If you try to create a `Shady.World` on non-Windows platforms without saying `threaded=False`, it will automatically revert to `threaded=False` and issue a warning, together with a reminder that you will have to call `Run()` explicitly. Unless, of course, you use a sneaky workaround, as described in the next section...

Multi-threaded operation on non-Windows platforms

It is convenient and readable, and especially conducive to interactive construction of a World and its stimuli, to be able to say:

```python
import Shady
w = Shady.World()
# ...
```

and have the World immediately start running in a different thread, while you continue to issue commands from the main thread to update its content and behavior. However, as explained above, you can only do this on Windows: on other platforms, the World will only run in the main thread.

There is a workaround, implemented in the utility function `Shady.Utilities.RunShadyScript()`, which is used when you start an interactive session with the `-m Shady` flag:

```bash
python -m Shady
```

or when you invoke your python script with the same flag:

```bash
python -m Shady my_script.py
```

(In the latter case the `run` subcommand is assumed by default, so this is actually a shorthand for:)

5.3. Getting Started
There are other subcommands you can use, such as `demo`, which allows you to run scripts as interactive tutorials if they are specially formatted—as many of our example scripts are.)

Starting Python with `--m Shady` (or equivalently, calling `RunShadyScript()` from within Python) starts a queue of operations in the main thread, to which thread-sensitive `Shady.World` operations will automatically be directed. It then redirects everything else (either the interactive shell prompt, or the rest of your script) to a subsidiary thread.

For many intents and purposes, this is just like starting the `Shady.World` in a background thread: its main advantage is that it allows you to build and test your `World` interactively on the command line. It has its limitations, however. For one thing, you can only create one `World` per session this way, whereas threaded `World` instances, on Windows, can be created one after another (you can even have two running at the same time—although we have no data and only pessimistic suspicions about their performance in that case). The fun also comes to a crashing end when you try to do something else that requires a solipsistic graphical toolbox, like plotting a `matplotlib` graph.

**Limitations on multi-threaded performance in Python**

So far, we have found that our multi-threaded `Shady` applications have generally worked well on Windows. This is largely because most of the rendering effort is performed on the GPU, and most of the remaining CPU work is carried out (at least by default if you have the ShaDyLib `accelerator`) in compiled C++ code rather than Python. Very very little is actually done in Python on each frame.

However, as soon as your Python code (animation callbacks, dynamic property assignments, and event handlers) reaches a certain critical level of complexity, you should be aware of the possibility that Python itself may cause multi-threaded performance to be significantly worse than single-threaded. This is because the Python interpreter itself cannot run in more than one thread at a time, and multi-threading is actually achieved by deliberately, cooperatively switching between threads at (approximately) regular intervals, mutexing the entire Python interpreter and saving/restoring its state on each switch. This is Python’s notorious Global Interpreter Lock or GIL, and a lot has been written/ranted about it on the Internet, so we will not go into the details here. Just be aware that it exists, and that consequently it is often better to divide concurrent operations between `processes` (e.g. using the standard `multiprocessing` module) rather than between threads. You might decide to design your system such that all your `Shady` stuff, and only your `Shady` stuff, runs in a single dedicated process. That process would then use the tools in `multiprocessing`, or other inter-process communication methods, to talk to the other parts of the system.

**5.3.2 Command-Line Options**

Our example scripts all use the `WorldConstructorCommandLine` utility. This means that when you launch them from the command-line, for example with:

```
python -m Shady run custom-functions
```

or:

```
python <path-to-package-installation-location>/Shady/examples/custom-functions.py
```

then in either case, the demo (here `custom-functions.py`) will support additional optional command-line arguments. One of these is `--help`, so you can find out what other arguments are supported. Some of them may be specific to the particular demo. But most of them are used to determine the arguments that get passed to the `World` constructor. For example:

```
python -m Shady run custom-functions --screen=2
```
will cause the custom-functions example to pass `screen=2` to the `World()` constructor and hence to run on
your second screen (perhaps you would have run `python -m Shady screens` first, to help you decide on the
correct screen number).

One command-line option, `--console`, arises from the `python -m Shady` mechanism itself rather than the
demos and their `WorldConstructorCommandLine()`. It is supported by the `run` and `demo` subcommands and
determines the level and type of interactivity while the script is running.

For more details, see:

- The `World` class constructor doc.
- The `WorldConstructorCommandLine` utility doc.
- The `RunShadyScript` utility doc.
- The output of `python -m Shady help`

### 5.3.3 Example Scripts

The following example scripts are included as part of the Shady package. They can be run conventionally like any
normal Python script. Alternatively, you can explore them interactively piece-by-piece with the `demo` subcommand,
like so:

```bash
python -m Shady demo showcase
```

From inside Python, this is equivalent to:

```python
from Shady import RunShadyScript, PackagePath
RunShadyScript( PackagePath( 'examples/showcase.py' ) )
```

### List of examples

An annotated version of the full list of examples can be obtained by typing:

```bash
python -m Shady list
```

The source code of the examples can be browsed below:

**examples/animated-textures.py**

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal
Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo
animated-textures`

```bash
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
```

(continues on next page)
#: How to switch between frames of a multi-frame image

This file demonstrates two different ways of switching between frames of a multi-frame image.

This demo requires third-party packages `numpy` and `pillow`.

```python
if __name__ == '__main__':
    # First deal with the demo's command-line arguments, if any:
    import Shady
    cmdline = Shady.WorldConstructorCommandLine()
    cmdline.Help().Finalize()
    Shady.Require( 'numpy', 'Image' )  # die with an informative error if either is missing
    w = Shady.World( **cmdline.opts )

    # Now we'll create an inhabitant.
    filename = Shady.PackagePath( 'examples/media/alien1.gif' )
    s = w.Stimulus( filename )

    # Now we'll make him walk, by setting his `.frame` property to a function of time:
    s.frame = lambda t: t * 16

    # The `.frame` property, like many World and Stimulus properties, supports dynamics. That means that, instead of setting it to a constant numeric value, you can assign a function of time.
    # If you ask to retrieve `s.frame`...
```
...then you get the instantaneous numeric value of
the property. If we do it repeatedly, we will get
different values:
```python
print( s.frame )
import time; time.sleep(0.5)
print( s.frame )
```

Stimuli can be created in this way from animated GIFs
(or equivalently from lists of pixel arrays, each array
specifying one frame). What actually happens is that the
frames are concatenated horizontally to form one wide
strip in the "carrier" texture:
```python
s.frame = 0
s.scaling = Shady.Transition( s.scaling, w.width / float( s.textureSize[0] ) )
s.WaitFor('scaling')
s.width = Shady.Transition( s.width, s.textureSize[0] )
```

Normally, the first element of `s.envelopeSize` (a.k.a `s.width`)
is set so that only one frame is visible. A change of `s.frame`
is actually realized by changing `s.carrierTranslation[0]` (a.k.a.
`s.cx`), so the carrier moves one frame-width at a time under the
envelope, like a zoetrope.

Well that’s all very nice, and it only uses one OpenGL texture,
but there’s a limit to the dimensions that an OpenGL texture can
have. So if the frame width multiplied by the number of frames
were to exceed the limit (which is hardware-/driver-defined, but
I’ve seen it be as low as 8192 pixels) then you will not be able to
do things this way. So there is a different animation mechanism,
called the "page" mechanism, if you need it...
```python
```

First let's put our friend back the way he was:
```python
s.Set(  
    frame = lambda t: t * 16,  
    scaling = Shady.Transition( s.scaling, 1.0 ),  
    width = Shady.Transition( s.width, s.frameWidth ),
)
```

Now let’s load the frames from disk into RAM:
```python
frames = Shady.LoadImage( filename )
```

It's a list of PIL Image objects. Type `frames` and press
return if you don't believe me. Go ahead, I'll wait.
***#

***
Now we'll create a new empty Stimulus:
***#
s2 = w.Stimulus( x=300 )
***
It currently has no texture, and its .backgroundColor is set to the default mid-grey. Let's load each frame of the image into a new "page". A new page corresponds to a new allocated texture in OpenGL, and its associated dimension settings:
***#
for i, frame in enumerate( frames ):
    s2.NewPage( frame, key=i )
***
You may have noticed that we could see the textures being loaded one by one. In practice you might want to create the Stimulus with `visible=False` and only make it visible after all the textures are in place. Alternatively, you can automate the process in one call:
***#
s2.LoadPages( frames )
# you can also construct the Stimulus with the option `multipage=True`
***
Either way, now we can use the `.page` property, which also supports dynamics in the same way as `.frame`. This time, for fun, let's use a special callable object from the `Shady.Dynamics` sub-module:
***#
s2.page = Shady.Integral(16)
***
Are they marching out of step with each other? They may or may not be, depending on exactly when you executed that last line, because the newly-constructed `Integral()` would have started from zero on the next frame after that. If it bothers you, I can think of a few different ways of putting these two guys into lock-step. The first is to retrieve the instantaneous numeric value of `s.frame` and add it to a new `Integral()`:
***#
s2.page = Shady.Integral(16) + s.frame
***
That at least demonstrates how you can do arithmetic operations with `Shady.Function` objects. But this approach is overkill when you could instead assign to `s2.page` a simple function of time that always returns the current value of `s.frame`:
***#
s2.page = lambda t: s.frame
***
Note that, while `.page` and `.frame` support dynamic value
assignment, they are not fully-fledged "managed properties". Often, you will want to share properties between stimuli, and can take advantage of "property sharing" which allows this kind of linkage *without* requiring additional Python instructions to run on each frame. The `sharing` demo has more details. However, for indirect "unmanaged" properties like `.page` and `.frame`, adding a lambda function that executes on each frame is about the best we can do to synchronize them.

```python
You can even combine the `.page` and `.frame` concepts to select between different animations for the same stimulus.
```

To illustrate this, let’s gather some resources created by craftpix.net and released under the OpenGameArt.org license. Let’s use a `glob` pattern to see what we’ve got:

```python
import os, glob

patterns = {
    os.path.basename( d ) : d + '/*.png'
    for d in glob.glob( Shady.PackagePath( 'examples/media/alien2/*' ) )
    if os.path.isdir( d )
}

for key, pattern in sorted( patterns.items() ):
    print( '% 6s : %s' % ( key, pattern ) )
```

Now let’s create a single stimulus that can switch between these collections of frames:

```python
alien2 = w.Stimulus( x=-300, frame=Shady.Integral( 5 ) )

for key, pattern in sorted( patterns.items() ):
    alien2.NewPage( pattern, key=key )
    print( 'loaded %r' % key )
```

Now we can switch between them:

```python
alien2.page = 'fire'
```

We can address the pages by those string names we gave them, or numerically. The latter means we could even cycle through the different animations automatically:

```python
alien2.page = Shady.Integral( 0.3 )
```

```python
Shady.AutoFinish( w ) # tidy up, in case we're not running this with 'python -m Shady'
```
This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo artifacts

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
# $END_SHADY_LICENSE$

#: Geometric transformations can lead to interpolation artifacts

""
This demo illustrates the artifacts that may be
created (especially in unlinearized stimuli) when a
texture stimulus is rotated, scaled or translated by
an amount that has not been appropriately rounded.

See `Shady.Documentation.PreciseControlOfLuminance`
for a section that describes when such artifacts
can occur and how to avoid them.
""

if __name__ == '__main__':
    import os
    import sys
    import Shady

    ""
    Wrangle command-line options:
    ""
    cmdline = Shady.WorldConstructorCommandLine()
    cmdline.Option( 'gamma', -1, type=( int, float ), min=-1, doc="Gamma-
correction parameter for some of the stimuli (-1 means sRGB)."
    )
    adjust = cmdline.Option( 'adjust', False, type=bool, container=None, doc="Whether or not to adjust gamma of the lower stimuli using the mouse/touchscreen." )
    cmdline.Help().Finalize()
    Shady.Require( 'numpy' ) # die with an informative error if this is missing
```

(continues on next page)
Create a World:

```python
w = Shady.World(bg=0.5, **cmdline.opts)
```

Some gamma-corrected stimuli:

```python
import numpy
size = min(w.width // 3 - 100, w.height // 2 - 100)
noise = numpy.random.uniform(size=[size, size])
xshift = size * 1.1
yshift = (w.height + size) // 6

s1 = w.Stimulus(noise, x=-xshift, y=-yshift, atmosphere=w)
s2 = w.Stimulus(noise, x=0, y=-yshift, atmosphere=w)
s3 = w.Stimulus(noise, x=+xshift, y=-yshift, atmosphere=w)
```

And some definitely-uncorrected stimuli:

```python
s4 = w.Stimulus(noise, x=-xshift, y=+yshift, gamma=1.0)
s5 = w.Stimulus(noise, x=0, y=+yshift, atmosphere=s4)
s6 = w.Stimulus(noise, x=+xshift, y=+yshift, atmosphere=s4)
```

Now let's make the stimuli on the left rotate very slightly back and forth; the ones in the middle will shrink and grow very slightly; and the ones on the right will get translated diagonally back and forth by a sub-pixel amount. All of these transformations will create interpolation artifacts, which may be very noticeable in the unlinearized stimuli.

```python
freq = 0.5
s4.envelopeRotation = s1.envelopeRotation = Shady.Oscillator(freq) * 2
s5.envelopeScaling = s2.envelopeScaling = Shady.Oscillator(freq) * 0.02 + 1.0
s6.envelopeOrigin = s3.envelopeOrigin = Shady.Oscillator(freq) * 1.0
```

If the "gamma-corrected" stimuli are, in fact, not well...
gamma-corrected on your particular screen, then perhaps
`gamma=-1` (i.e. the sRGB profile) was not the correct choice.
You could always experiment with adjusting `w.gamma` by hand.
Or adjust it using the mouse/touch-screen, with::
```
"""
Shady.Utilities.AdjustGamma( w )  # we don't frequently use this

→function
→which is why it is not in the top-level
→`Shady.` namespace. Normally we would
→use `Shady.FindGamma` which wraps it but
→which also renders a linearization pattern.
"""
print( ""

Adjusting gamma with mouse/touchscreen. Press escape
TWICE to close: once to exit the adjustment procedure,
and once to close the window.
"""
)
Shady.AutoFinish( w )

examples/capture-video.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal
Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo
capture-video

#: Capturing video of a Stimulus animation (non-realtime animation)
This demo demonstrates how the a rendered `Stimulus` animation can be captured to a video file. To do this, Shady must slow down the animation below real time.

It requires the module `cv2` from the third-party package `opencv-python`.

```python
if __name__ == '__main__':

    import Shady

    Parse command-line options:
    
    cmdline = Shady.WorldConstructorCommandLine( width=700, height=700, top=100, frame=True, canvas=True )

    gamma = cmdline.Option( 'gamma', -1, type=( int, float ), min=-1, doc='This controls the World/canvas `.gamma` property describing the screen non-linearity (-1 means "sRGB").' )
    noise = cmdline.Option( 'noise', -0.2, type=( int, float, tuple, list ), min=-1.0, max=1.0, minlength=3, maxlength=3, doc='This controls the World/canvas `.noiseAmplitude` property. Supply a scalar to specify a gray luminance level, or an R,G,B triplet to specify a color. Negative values get you a uniform distribution, positive get you a Gaussian distribution.' )
    speed = cmdline.Option( 'speed', 100, type=( int, float ), container=None, doc='This controls the drift speed of the sinusoidal carrier wave in pixels per second.' )
    output = cmdline.Option( 'output', 'example_movie', type=str, minlength=1, container=None, doc='This is the filename (or file stem) for saving the video.' )

    cmdline.Help().Finalize()

    Shady.Require( 'cv2', 'numpy' ) # die with an informative error if either is missing

    Create a `World` and a `Stimulus`:
    
    world = Shady.World( **cmdline.opts )
    stim = world.Stimulus(
        signalFunction = Shady.SIGFUNC.SinewaveSignal,
        signalAmplitude = 0.5,
        plateauProportion = 0,
        cx = lambda t: t * speed,
        contrast = ( 0.5 + noise ) if noise < 0.0 else ( 0.5 - 3 * noise ),
        atmosphere = world )

    Capturing each rendered frame is itself a slow process, and will slow our animation down below real time. We are forced to choose between accurate timing of the animation as it appears in real time on screen, and accurate timing in the movie file. In this case, of course, we care about timing in the file. Since all Shady's animation routines are functions of time `t`, we can no longer pass the real wall time as `t`. Instead we

(continues on next page)
must pass a "fake" clock output based on the nominal frame rate we want and the number of frames that have passed.

This is done by setting the `world.fakeFrameRate` (which would normally be left as `None`):

```python
    world.fakeFrameRate = 60.0  # ensures accurate slower-than-real-time animation
```

Now we make a `VideoRecording` instance. We can pass it our `World` instance in the `fps` argument---this is a syntactic shorthand for passing `world.fakeFrameRate`.

```python
    movie = Shady.VideoRecording( output, fps=world )
```

To avoid corruption, the movie file will need to be explicitly closed. Let’s ensure that happens, at the latest, when the `World` ends:

```python
    world.BeforeClose( movie.Close )
```

Now we need to set up an animation callback that calls `movie.WriteFrame` each time a new frame is rendered. We’ll attach the animation callback to the `Stimulus` (though we could equally attach to the `World`).

The `frame` argument to `.WriteFrame()` can be a `numpy` array, or as a shorthand it can be a `World` or `Stimulus` instance---anything with a `.Capture()` method that returns a numpy array; this method is then automatically called on each frame.

```python
    @stim.AnimationCallback
def StimFrame( self, t):
        movie.WriteFrame( self )
        print( 'Now streaming output to ' + movie.filename )
```

Note that the real-time animation has slowed down. The speed should be correct when the file is played back at 60 fps, however.

Note that the default codec is lossy, so the pixel values in the movie may not be 100% accurate (lossy movie files should not be used for analysis of stimulus content).

Streaming will end when the `World` closes (which you can trigger with any key-press).

```python
    @world.EventHandler
def AnyKeyToExit( self, event ):
This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo custom-functions`

```python
if __name__ == '__main__':
    
    import Shady

    cmdline = Shady.WorldConstructorCommandLine( canvas=True, reportVersions=True )
    cmdline.Help().Finalize()
```

5.3. Getting Started
Customization is performed *before* any instances of `World` or `Stimulus` are created. Let's start by defining a custom signal function. A signal function is written in GLSL and follows one of the following two prototypes:

```glsl
float func( vec2 coords ) { ... } // monochromatic output
vec3 func( vec2 coords ) { ... } // RGB output
```

where `coords` is a 2-D coordinate in pixels measured from the center of the stimulus.

The one pre-existing signal function is called `SinewaveSignal`, which has the associated index number 1. Signal function names are mapped to numbers in the namespace `Shady.SIGFUNC`:

```python
print( Shady.SIGFUNC.SinewaveSignal )
```

More names will appear in this namespace as you define more functions yourself. The idea is that, while you can activate the sine-wave signal function for your stimulus instance `stim` by saying:

```python
stim.signalFunction = 1
```

it makes for more transparent, readable code if you express that as:

```python
stim.signalFunction = Shady.SIGFUNC.SinewaveSignal
```

Let's look at the source code for `SinewaveSignal` in the actual shader program:

```python
sourceFileName = Shady.PackagePath( 'glsl/FragmentShader.glsl' )
sourceCode = open( sourceFileName ).read()
import re
match = re.search( r'\n\S+\s+SinewaveSignal\s*\{.+?\n}', sourceCode, re.S)
if match :print( match.group() )
```

(retrieved from glsl/FragmentShader.glsl) inside the Shady package).

You'll see that `SinewaveSignal` uses a variable called `uSignalParameters`. This is a "uniform" variable in the shader, meaning that its value does not change from pixel to pixel in a given stimulus and that we are able to change its value from the CPU side. According to Shady's naming conventions, the `uSignalParameters` variable receives its value from a managed
property called `.signalParameters` (which, in this case, belongs only to the 'Stimulus' class).

The function also uses the `sinusoid()` helper function. For our first example, let's use both of these pre-existing tools to implement an antialiased square-wave signal function, as a finite sum of sinusoid() components. We'll do this with the global function 'AddCustomSignalFunction', to which we need to pass a multi-line string containing the GLSL shader code:

```glsl
float SquarewaveSignal( vec2 coords )
{
    float y = 0.0;
    for( float harmonic = 1.0; ; harmonic += 2.0 )
    {
        float cyclesPerPixel = uSignalParameters[ 1 ] * harmonic;
        if( cyclesPerPixel > 0.5 ) break;
        y += sinusoid(coords,
                      cyclesPerPixel,
                      uSignalParameters[ 2 ],
                      harmonic * uSignalParameters[ 3 ]
                    ) * uSignalParameters[ 0 ] / harmonic;
    }
    return y;
}
```

Note that this has automatically added a new index to the `Shady.SIGFUNC` namespace:

```glsl
print( Shady.SIGFUNC.SquarewaveSignal )
```

For our next trick, let's make a signal function that displays a frozen random uniform noise. We can use the helper function `random()` which takes a 2-dimensional coordinate as its seed, and returns a pseudo-random number from the uniform distribution over the range [-1, +1]. For scaling we'll use the existing `.signalAmplitude` property which is a shortcut to the first element of the `.signalParameters` property, and hence to the first element of the `uSignalParameters` uniform variable in the shader.

```glsl
float RandomSignal( vec2 coords )
{
    vec3 seed3 = vec3( ( 1.0 + coords ) / ( 2.0 + uTextureSize ), uSeed,
                     1.0 );
    return random( seed3 ) * uSignalParameters[ 0 ]; // output of
```

(continues on next page)
Note that we also introduced a customizable random-seed as the uniform variable `uSeed`. This does not exist yet, but it can be created, and manipulated as a `Stimulus` property, if we use the property name `seed`:

```python
Shady.Stimulus.AddCustomUniform( seed=1.0 )
```

# Note that this is not a global function, but rather a class method of the `Stimulus` class. You can also add custom properties/uniform variables to the `World` class if you want.

Let's do another one. What else do psychophysicists like? Of course: plaids! Let's create a Plaid() signal function, parameterizing the angle between plaid components with another new uniform:

```python
Shady.Stimulus.AddCustomUniform( plaidAngle=90.0 )
Shady.AddCustomSignalFunction(""
float Plaid( vec2 coords )
{
    );
}
"")
```

Hopefully you've got the idea how to customize signal functions. What about contrast-modulation functions? You may have noticed that we carefully included the word "Signal" in the name `SquarewaveSignal`. This allows us to mirror the existing distinction, in the shader, between `SinewaveSignal` and `SinewaveModulation`, which is important because the two functions use different uniform variables for their parameters, and a different convention for interpreting the amplitude parameter.

Modulation functions always have scalar output, so their prototype is always:

```python
float f( vec2 coords ) { ... } # scalar output only
```

With all of this in mind, let's make the analogous `SquarewaveModulation` function:

```python
Shady.AddCustomModulationFunction(""
```
float SquarewaveModulation( vec2 coords )
{
    // NB: *not* spatially antialiased (that's left as an exercise for...
    float y = sign( sinusoid( coords, uModulationParameters[ 1 ],
                        uModulationParameters[ 2 ], uModulationParameters[ 3 ] ) );
    return 1.0 + uModulationParameters[ 0 ] * ( y - 1.0 ) / 2.0;
}

Finally, the last class of functions that is customizable
is the windowing function. The prototype for a custom
windowing function is:

    float f( float r ) { ... }

where \( r \) varies between 0 at the peak (or throughout the
plateau, if any) and 1 at the edge of the stimulus. So, if
you absolutely positively have to have, say, Blackman-Harris windows instead of Hann windows:

```
Shady.AddCustomWindowingFunction(""
float BlackmanHarris( float r )
{
    r += 1.0;
    r *= PI;
    float w = 0.35875;
    w += -0.48829 * cos( r );
    w += +0.14128 * cos( r * 2.0 );
    w += -0.01168 * cos( r * 3.0 );
    return w;
}
```

...or Gaussian windows, extending out to a configurable
number of sigmas:

```
Shady.Stimulus.AddCustomUniform( gaussianSigmas=3.0 )
Shady.AddCustomWindowingFunction(""
float Gaussian( float r )
{
    r *= uGaussianSigmas;
    float v = exp( -0.5 * r * r );
    // uncomment the following to ensure the window really comes down to
    // zero:
    // float tailThickness = exp( -0.5 * uGaussianSigmas * uGaussianSigmas );
    // v = { v - tailThickness } / ( 1.0 - tailThickness );
    return v;
}
```

# 5.3. Getting Started

# All of this customization had to be done *before* `World`
Now let's put it all together, and test each of our custom additions. First, create a `World`:
```python
w = Shady.World( **cmdline.opts ).Set( gamma=2.2 )
```

Now the stimuli. First, a square-wave signal patch, windowed a la Blackman-Harris:
```python
s1 = w.Stimulus(
    size = 500,
    x = -250,
    y = +250,
    plateauProportion = 0,  # non-negative: turns windowing on
    windowingFunction = Shady.WINFUNC.BlackmanHarris,  # custom
    signalAmplitude = 0.5,
    signalFunction = Shady.SIGFUNC.SquarewaveSignal,  # custom
    atmosphere = w,
)
```

Then a plaid, with the usual Hann window:
```python
s2 = w.Stimulus(
    size = 500,
    x = +250,
    y = +250,
    plateauProportion = 0,  # non-negative: turns windowing on
    signalFunction = Shady.SIGFUNC.Plaid,  # custom
    signalAmplitude = 0.25,
    signalOrientation = 45,
    atmosphere = w,
)
```

Now a frozen noise, again in a Hann window:
```python
s3 = w.Stimulus(
    size = 500,
    x = -250,
    y = -250,
    plateauProportion = 0,  # non-negative: turns windowing on
    signalFunction = Shady.SIGFUNC.RandomSignal,
    signalAmplitude = 0.5,
    atmosphere = w,
)
```

And finally a Gabor with additional square-wave contrast modulation:
```python
s4 = w.Stimulus(  
```
Now let's animate some aspects of our stimuli, including the new property `.plaidAngle` in the plaid stimulus:

```python
s1.cx = Shady.Integral( 50 )
s1.windowingFunction = lambda t: 
    Shady.WINFUNC.BlackmanHarris if 0 <= ( t % 3 ) < 1 else 
    Shady.WINFUNC.Gaussian if 1 <= ( t % 3 ) < 2 else 
    Shady.WINFUNC.Hann
s2.plaidAngle = Shady.Oscillator( 0.2 ) * 45 + 45
s3.seed = lambda t: 1 + int( t )
s4.modulationDepth = Shady.Oscillator( 0.2 ) * 0.5 + 0.5
```

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo dithering`
This demo tests the accuracy of the "noisy-bit" dithering algorithm (Allard & Faubert 2008) as implemented in our fragment shader. A "uniform" canvas is rendered with each specified target DAC value in turn. After each is rendered, a function is called to capture the screen and analyze its pixel content. When the target DAC value is an integer, the screen should be truly uniform and the error should be zero. When it is a non-integer value, only the integer values immediately above and immediately below target should appear, and the mean value across all pixels should be very close to the target.

See also the 'precision' demo.

```python
if __name__ == '__main__':
    import Shady
    cmdline = Shady.WorldConstructorCommandLine()
    targetDACs = cmdline.Option('targetDAC', [0.5,1,254,254.5], type=(int,float,tuple,list), min=0, container=None, doc='A DAC value or sequence of DAC values. If your graphics card is 8-bit, like most, then the values here should not exceed 255. Try integer and non-integer values.')
    denom = cmdline.Option('denom', 'auto', type=(int,float,str), strings=['auto'], container=None, doc='You can explicitly override the default. Your graphics card (usually 255).')
    gamma = cmdline.Option('gamma', -1, type=(int,float), min=-1, doc='Screen non-linearity parameter (-1 means "sRGB").')
    cmdline.Help().Finalize()
    numpy = Shady.Require('numpy')  # die with an informative error if this is missing
    cmdline.opts['canvas'] = True
    if denom != 'auto': cmdline.opts['ditheringDenominator'] = denom
    if isinstance(targetDACs, (tuple,list)): targetDACs = list(targetDACs)
    else: targetDACs = [targetDACs]
    #print('='*30); print(cmdline.opts); print('='*30)

    def ScreenNonLinearity(targetDAC, gamma='sRGB', numType=float):
        # same as Shady.ScreenNonLineariry, but can emulate what happens
        # when the calculations are performed at a different numeric precision
        # (e.g. numType=numpy.float32 would emulate the 32-bit float performance)
        # of the shader
        f = numType
        value = f(targetDAC) / f(255.0)
        if gamma in ['sRGB'] or gamma <= 0:
            value = ((value + f(0.055)) / f(1.055)) ** f(2.4)
        else:
            value = value / f(12.92)
```

(continues on next page)
else:
    value *** f( gamma )
return value

""

We'll define a Measure() function that starts by performing
a .Capture() of the current world content, and then computes
statistics on the resulting pixel values.
"""

def Measure( world, targetDAC ):
a = world.Capture()
hist = Shady.Histogram( a, DACmax=world.dacMax, plot=False )
print( '==== targetDAC = %r ====

for k, counts in sorted( hist.items(), reverse=True ):
    nPixels = float( sum( counts ) )
dacValues = numpy.arange( float( len( counts ) ) )
    avg = sum( dacValues * counts ) / nPixels
print( '%s: mean = %.5f' % ( k, avg )
if len( nonzero ) <= 5:
    for dacValue, count in nonzero:
        print( '%3d: %.4f' % ( dacValue, count / nPixels )
else:
    print( '%d unique pixel values' % len( nonzero )
if 0:
    # TODO: this implementation is easier to understand, but it needs
    # threading---either via 'python -m Shady',
    # or (if the OS permits) natively with --threaded=True
    ""
    Render a "uniform" canvas with each specified target DAC value in
    turn. After rendering each, call Measure()
    """
    w = Shady.World( **cmdline.opts )
for targetDAC in targetDACs:
    w.backgroundColor = ScreenNonlinearity( targetDAC, gamma=gamma, numType=numpy.float32 )
    w.Wait() # this will hang forever if we're not threaded
    Measure( w, targetDAC )
    w.Close()
else:
    # TODO: this implementation is more robust, more suitable for a
    # releasable
(continues on next page)
Render a "uniform" canvas with each specified target DAC value in turn. After rendering each, call Measure().

```python
w = Shady.World(**cmdline.opts)
@w.AnimationCallback
def Animate(self, t):
    if not targetDACs: return self.Close()
    targetDAC = targetDACs.pop(0)
    self.backgroundColor = ScreenNonlinearity(targetDAC,
        gamma=gamma, numType=numpy.float32)
    self.Defer(self.Defer, Measure, world=self, targetDAC=targetDAC)
    # Defer()red actions get carried out immediately after this animation callback, before the effect of the new .backgroundColor even gets rendered. So we actually need to double-Defer(), i.e. Defer() a call to Defer() which will schedule the Measure() function.
    else:
        # TODO: this implementation is also robust, and so also suitable for a releasable example script, but the use of a generator function will inevitably be obscure to many
        Render a "uniform" canvas with each specified target DAC value in turn. After rendering each, call Measure().

        We're going to use a slightly obscure trick to ensure correct scheduling: we'll register a Python "generator function" as the animation callback, instead of a regular function. A generator function is any function with the `yield` keyword in it.

        Unusually for an animation callback, there's no `t` argument. This is because the function will actually only be called once (on the first frame after it is registered) rather than repeatedly on every frame. What then happens is that whenever we hit a `yield` statement, we pop out of the function back into Shady's main loop. And when the next frame comes around, we pop back in again to resume the function where we left off. (If we ever needed to know the time inside the generator code, it's always available as `self.t` anyway.)
        #
        w = Shady.World(**cmdline.opts)
        @w.AnimationCallback
        def Animate(self):
            for targetDAC in targetDACs:
                self.backgroundColor = ScreenNonlinearity(targetDAC,
                    gamma=gamma, numType=numpy.float32)
                # allow one frame for the `.backgroundColor` setting to take effect before we...
                yield # allow one frame for the `.backgroundColor`
                Measure(w, targetDAC)
                self.Close()
            Shady.AutoFinish(w)
        ```
This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo dots1

```python
if __name__ == '__main__':
    ""
    First let’s wrangle the command-line options:
    ""
    import Shady
    cmdline = Shady.WorldConstructorCommandLine( canvas=False )
    ndots = cmdline.Option( 'ndots', 300, type=int, container=None, doc="Number of dots (may not exceed %d)." % Shady.Rendering.MAX_POINTS )
    bounce = cmdline.Option( 'bounce', False, type=bool, container=None, doc="If True, implement some rudimentary physics to make the dots bounce off each other.
    "")
    gauge = cmdline.Option( 'gauge', True, type=bool, container=None, doc="Whether or not to show a 'FrameIntervalGauge'." )
    thickness = cmdline.Option( 'thickness', 20, type=( int, float ),
    --container=None, doc="Value for the '.penThickness' property, dictating the size of
    --the dots." )
    smooth = cmdline.Option( 'smooth', True, type=bool, container=None, doc="If True, flag the dots as "smooth". This may make them round instead of square, or
    --it may have no effect - unfortunately this is driver-dependent. If you want to
    --guarantee round dots you'll have to use small polygons instead (see dots3 and dots4
demos) or separate 'Stimulus' instances (dots2)."
    "")
```
cmdline.Help().Finalize()
Shady.Require( 'numpy' )  # die with an informative error if this is missing

```python
Create a World and, if requested, a frame interval gauge:
```#
```python
w = Shady.World( **cmdline.opts )
if gauge: Shady.FrameIntervalGauge( w )
```

```python
Now a single Stimulus that will host our random dots.
We'll use the `POINTS` drawing mode.
```#
```python
field = w.Stimulus( size=w.size, color=1, drawMode=Shady.DRAWMODE.POINTS,
→penThickness=thickness, smoothing=smooth )
```

```python
To draw the points, we'll have to set the `.points`
property.
```#
```python
import numpy
location = numpy.random.uniform( low=[ 0, 0 ], high=field.size, size=[
→ndots, 2 ] )
velocity = numpy.random.uniform( low=[ -30, -30 ], high=[ +30, -150 ], size=[
→ndots, 2 ] )
field.points = ( Shady.Integral( velocity ) + location ) % field.size
```

```python
Now, for a bit of fun, we'll define a couple of functions
that allow the dots to bounce off each other.
```#
```python
physics = dict( exponent=10, closest=10, coefficient=800 )
def RepulsionForces( t=None ):
    positions = field.pointsComplex
    vectors = positions[ None, : ] - positions[ :, None ]
    magnitudes = numpy.abs( vectors )
    degenerate = magnitudes < 1e-4
    nondegenerate = ~degenerate
    vectors[ nondegenerate ] /= magnitudes[ nondegenerate ]
    # vectors are
→now unit vectors
    magnitudes[ degenerate ] = numpy.inf
    magnitudes = numpy.clip( magnitudes - field.penThickness / 2.0,
→physics[ 'closest' ], numpy.inf )
    magnitudes /= physics[ 'closest' ]
    magnitudes += -physics[ 'exponent' ]  # now we have inverse square
→(or whatever power) distances, with 0s on diagonal
    forces = magnitudes * vectors * physics[ 'coefficient' ]
    forces = forces.sum( axis=0 )
    forces = Shady.ComplexToReal2D( forces )  # n-by-2 real-valued output
    return forces
def Bounce( **kwargs ):
    physics.update( kwargs )
    field.points = ( Shady.Integral( Shady.Integral( RepulsionForces ) +
→velocity ) + field.points ) % field.size
    if bounce: w.Defer( Bounce )
```

(continues on next page)
# .Defer() will ensure that the function gets called at the end of
# the next frame (this ensures that field.pointsComplex, which
# RepulsionForces() relies on, has already had a value assigned to it

```python
###
If you requested this with the `--bounce` command-line option
then the points should already be bouncing off each other.
If not, you can manually trigger it if you want, by calling
`Bounce()`. Either way, our dots demo is done.

Note that `drawMode=POINTS` is the simplest way of creating
dot patterns, but not necessarily the most powerful. Setting
`smaoothing=True` should in principle make them round, but
whether it successfully does so is dependent on your graphics
driver: they may stubbornly remain square. The best way of
guaranteeing the shape of your dots is to draw them as
tiny polygons (with a high number of sides, if you want them
to look round). This can be explored in the `dots3` and
`dots4` demos. Another option is to make each "dot" an
independent Stimulus - but as the `dots2` demo shows, you
have to worry much more about timing performance in that case.
```#

```python
###
By the way: if you're not pleased with the frame rate of your
colliding dots, and you're reading this console in the foreground,
try switching to the World as your main window. This will put
Shady in charge of synchronizing frame buffer swaps and likely
lead to significant improvements in performance.
```#

```python
Shady.AutoFinish( w )
```

### examples/dots2.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python
script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo dots2`

```python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
# $END_SHADY_LICENSE$
```

(continues on next page)
## How many separate `Stimulus` instances can we manage?

""
In the `dots1` demo we saw a simple way of presenting a large number of independently-moving elements. One of its limitations was that the appearance of the dots could not be guaranteed (some graphics drivers can present rounded dots, and others cannot—spoiler alert: the `dots3` demo has the best answer to this particular problem). Another limitation is that the dots are completely homogeneous in their appearance (because they are part of the same `Stimulus`, hence the same properties apply to all of them).

This demo explores the idea of controlling each and every "dot" as a separate `Stimulus`. This places much greater demand on Shady: while it's easy and cheap to manipulate several thousand shapes in the same `Stimulus`, rendering hundreds of separate `Stimulus` instances pushes Shady much more quickly to its performance limits, due to the accumulated small overhead costs. Nonetheless it is worth exploring where, exactly, these limits lie, and some of the tricks we can use to push them. This demo creates multiple circular stimuli, provides three different strategies for updating their positions independently of each other, and provides various tools for analyzing the relative efficiency of these strategies under different conditions.
"""

```python
import sys
import Shady

""
Let's start by defining a subclass of `World`.

The base `World` class has a `Prepare()` method. When you construct a `World`, any constructor arguments that are not recognized (either as construction options or `World` properties) are passed through to the `Prepare()` method. In the base class, all that happens is that they get printed to the console. Here in our subclass, we will define the `Prepare()` method to perform stimulus setup. The advantage of doing it here is that `Prepare()` is guaranteed to run in the same thread as the `World` rendering loop, so there are no `.Defer`red actions.

"""

class DotWorld( Shady.World ):
    ""
    Here's the Prepare method:
    """
    def Prepare( self, ndots=150, mode='batch', gauge=False, textured=False,
                 swap=1, blur=True, radius=25 ):
```

 Chapter 5. To cite Shady:
import numpy
self.anchor = -1
self.Set( gamma=2.2, bg=0.5 )
s
self.dots = [ self.Stimulus(
    name = 'dot%04d',
    size = radius * 2,
    color = numpy.random.rand(3),
    bgalpha = 0,
    pp = ( float( i ) / ndots ) if blur else 1,
    debugTiming = False,
    atmosphere = self,
) for i in range( ndots ) ]
s
self.start = numpy.array([ self.Place( numpy.random.uniform( -1, 1, ___size=2 ) ) for dot in self.dots ])
s
self.velocity = numpy.random.uniform( low=[ -30, -30 ], high=[ +30, -150 ], size=self.start.shape )
s
self.batch = self.CreatePropertyArray( 'envelopeTranslation', self.

dots ) # could use 'envelopeTranslation' or 'envelopeOrigin'

# for manipulating positions all in one batch array operation:
s

-envelopeTranslation or .envelopeOrigin

self.allPositions[ : ] = self.start

# for manipulating positions in a single loop over arrays:
s
self.each = list( zip( self.allPositions, self.start, self.velocity )

# for manipulating positions by attaching a dynamic to each and every...

stimulus:

def MakeDynamic( start, velocity, worldSize ): return lambda t: ( start + velocity * t ) % worldSize

self.lambdaFunctions = [ MakeDynamic( start, velocity, self.size )

for _, start, velocity in self.each ]

if gauge: Shady.FrameIntervalGauge( self, color=0 )
s
self.master = self.dots[ -1 ]
s
self.master.ShareProperties( self.dots, 'envelopeSize envelopeScaling

textureID textureSlotNumber useTexture' )
s
self.numberOfActiveStimuli = len( self.dots )

if textured: self.ToggleTextures()
s
self.Swap( swap )
s
self.UpdateMode( mode )

```""

One of the key things our 'Prepare()' method did is create
a 'PropertyArray' object called 'self.batch'. Its attribute
'self.batch.A' is a 'numpy' array containing a packed
representation of the '.envelopeTranslation' arrays of all
the "dot" stimuli. So, while we still have the option of
addressing their positions individually, we now also have
the more efficient option of addressing them collectively in
a single array operation.

```""":

```""

Here's a potential 'AnimationCallback': it updates the dot
positions one by one, in a Python loop.

```(continues on next page)
Here's an alternative 'AnimationCallback': it updates the dot positions all in one array operation, taking advantage of the 'PropertyArray' we created.

Here's a method that allows us to switch the way we update 'Stimulus' positions. 'multi' mode uses an individual function call for each stimulus (dynamic value assignment); 'loop' mode uses 'AnimateEach', a single animation callback that updates the stimuli in a Python loop; and 'batch' mode uses 'AnimateAllTogether', a single animation callback that updates everything in one single 'numpy' array operation.

To help us distinguish which mode we're in, we'll turn the 'World' green for multi mode, red for loop mode and blue for batch mode.

The following method will allow us to investigate the performance impact of texture rendering:
def ToggleTextures( self ):
    if self.master.source is None:
        # first-time setup
        self.master.LoadTexture( Shady.PackagePath( 'examples/media/face.png' ), False )
        self.master.cr = Shady.Clock( speed=90 ) # the master

    # rotates - see if you can spot it
    else:
        # toggle texture on or off
        self.master.useTexture = not self.master.useTexture

    ""
    And now here's a method that will allow us (at least on Windows) to halve our frame-rate and thereby hopefully homogenize it, if our dots are pushing the performance envelope too hard:
    ""

def Swap( self, nFrames ):
    msg = 'calling SetSwapInterval( %r )' % nFrames
    if nFrames > 1 and not sys.platform.lower().startswith( 'win' ) :
        msg += ' --- NB: values >1 might not be respected on this system'
    print( msg )
    self.SetSwapInterval( nFrames ) # needs accelerator (and seems to fail on mac)

    ""
    Finally, an event-handler. There are multiple ways to implement these (see the `events` demo) but since we're already defining a subclass the most straightforward way is simply to overshadow the `HandleEvent` method:
    ""

def HandleEvent( self, event ):
    if event.type == 'key_release':
        if event.key in [ 'q', 'escape' ]: self.Close()
        elif event.key in [ 't' ]: self.ToggleTextures()
        elif event.key in [ 'm' ]: self.UpdateMode( 'multi' )
        elif event.key in [ 'l' ]: self.UpdateMode( 'loop' )
        elif event.key in [ 'b' ]: self.UpdateMode( 'batch' )
        elif event.key in [ '1', '2' ]: self.Swap( int( event.key ) )
        elif event.key in [ '-', '+', '=' ]:
            if event.key in [ '+', '=' ]: self.numberOfActiveStimuli += 50
            elif event.key in [ '-' ]: self.numberOfActiveStimuli -= 50
            self.numberOfActiveStimuli = min( len( self.dots ), max( 0, self.numberOfActiveStimuli ) )
            for dot in self.dots[ :self.numberOfActiveStimuli ]:
                dot.Enter()
            for dot in self.dots[ self.numberOfActiveStimuli: ]:
                dot.Leave()
            self.Report()

if __name__ == '__main__':
    ""
    OK, that's the class definition done. Let's sort out command-line options:
    ""
cmdline = Shady.WorldConstructorCommandLine( canvas=False )
cmdline.Option( 'ndots', 150, type=int, doc='''Number of shapes. Unlike the other "dots" demos, each distinct "dot" is a separate, independent `Stimulus` instance. (This will quickly tend to push the boundaries of Shady's performance.)''' )
cmdline.Option( 'mode', 'batch', type=str, strings=['multi', 'loop', 'batch'], doc="Select one of three modes in which to update stimulus positions (each with a different overhead cost)." )
cmdline.Option( 'gauge', True, type=bool, doc="Whether or not to show a `FrameIntervalGauge`." )
cmdline.Option( 'textured', False, type=bool, doc="Whether or not to render a texture on each `Stimulus` (if so, one texture is loaded, and shared between all `Stimulus` instances)." )
cmdline.Option( 'blur', True, type=bool, doc="Whether or not to vary the `.plateauProportion` among stimuli." )
cmdline.Option( 'swap', 1, type=int, min=1, doc="Swap interval, in physical frames.\nUsually swap=1 -> 60 fps; swap=2 -> 30 fps\nThe swap interval, in physical frames.\n(\nUsually swap=1 -> 60 fps; swap=2 -> 30 fps\n\n(work in Windows)\n" )
cmdline.Option( 'radius', 25, type=(int, float), min=1, doc="Radius of each Stimulus, in pixels." )

# since we haven't said `container=None` for any of these options, they will all end
# up in the `cmdline.opts` dict, along with all the standard `World`-construction # items. Our `World` constructor will not recognize them, so it will pass them through # to the `Prepare` method, which we *have* implemented to recognize them.
cmdline.Help().Finalize()
Shady.Require( 'numpy', 'Image' )

""

So now all we have to do is create and run the World subclass:

```
#:
w = DotWorld( **cmdline.opts )
```

""

Keyboard commands:

T         Toggle texture on/off
M / L / B Select 'multi', 'loop' or 'batch' mode for stimulus position updates
1 / 2     SetSwapInterval to 1 or 2 physical frames
           (usually: 1 -> 60fps; 2 -> 30fps; may only work on Windows)
- / +     Reduce / increase the number of stimuli rendered
           on each frame (up to the maximum specified by the --ndots option)
Q / escape Close window

""

Shady.AutoFinish( w )
examples/dots3.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo dots3

```python
#: Need to guarantee rounded dots? Animate multiple polygons!
#
This demo shows how it is very easy, and still fairly computationally inexpensive, to animate a dot stimulus in which each "dot" is a polygon. It uses the POLYGON draw mode, and our setup will be aided by the `ComplexPolygonBase` utility function.

Using polygons with a high number of sides (say, 20) is the only way to guarantee round-shaped dots. The simpler approach of the `dots1` demo, using the POINTS draw mode and 'smoothing=True', is not guaranteed to produce round dots---it depends on your graphics hardware and drivers.

if __name__ == '__main__':
    # First let's parse the command-line arguments for this demo:
    import Shady

    # All the usual World-construction ones we use in most demos:
    cmdline = Shady.WorldConstructorCommandLine( canvas=False )
    # And some that are specific to this demo:

    ndots = cmdline.Option( 'ndots', 300, type=int, container=None, doc= ˓→"Number of shapes. Ensure (nsides+1)*ndots <= %d" % Shady.Rendering.MAX_POINTS )
    nsides = cmdline.Option( 'nsides', 5, type=int, min=3, container=None, doc= ˓→"Number of sides. Ensure (nsides+1)*ndots <= %d" % Shady.Rendering.MAX_POINTS )
    radius = cmdline.Option( 'radius', 25.0, type=( int, float ), doc="Half-width of each shape, in pixels." )
```

(continues on next page)
spin = cmdline.Option( 'spin', 0.2, type=( int, float ), container=None, doc="Spin the shapes at this speed (revolutions per second)."
)  
gauge = cmdline.Option( 'gauge', False, type=bool, container=None, doc="Whether or not to show a `FrameIntervalGauge`." )

Shady.Require( 'numpy', 'Image' )  

# die with an informative error if either is missing

if ( nsides + 1 ) * ndots > Shady.Rendering.MAX_POINTS: raise ValueError(  
'(nsides + 1) * ndots cannot exceed %d' % Shady.Rendering.MAX_POINTS )

""
Create a World:
"":

w = Shady.World( **cmdline.opts )
if gauge: Shady.FrameIntervalGauge( w )

""
Create a Stimulus to act as a container for the shapes. For fun, we're going to give this one a background image:
"":

field = w.Stimulus( Shady.PackagePath( 'examples/media/waves.jpg' ), size=w.size, visible=1 )  
# And now, a slightly awkward manipulation of carrier parameters, to make the image fill
# the screen without stretching the coordinate system in which the shapes are defined (that's
# what would happen if we were to change .envelopeScaling)
field.carrierTranslation = ( field.envelopeSize - field.textureSize ) // 2
field.carrierScaling = max( field.envelopeSize.astype( float ) / field.textureSize )

""
Set up the dynamic that draws the shapes:
"":

shape = Shady.ComplexPolygonBase( nsides )
import numpy
location = numpy.random.uniform( low=[ 0, 0 ], high=field.size, size=[ndots, 2 ] )
velocity = numpy.random.uniform( low=[ -30, -30 ], high=[ +30, -150 ], size=[ndots, 2 ] )

func = Shady.Integral( lambda t: velocity )  
# we could say just Shady.

Integral( velocity )  

# but then we wouldn't be able to

# change the

# velocity on-the-fly

cfunc = location  
cfunc += field.size  
cfunc %= field.size  
cfunc.Transform( Shady.Real2DToComplex )  

cfunc += lambda t: radius * shape * 1j ** ( 4.0 * spin * t )  

# numpy broadcasting in the `+` operator does the rest.
# It is possible to assign a sequence of complex numbers to the `.points`

# property,
# so we will leave the `func` output in complex form.
Apply it:

```python
field.Set( points=func, drawMode=Shady.DRAWMODE.POLYGON, visible=1 )
```

Note that the `lambda` function we defined in the last line of our setup is continually accessing, on every frame, three variables in the current namespace, called `radius`, `shape` and `spin`. This means that any manipulations you make to these variables, at the current command-line, will immediately affect the stimulus.

Try playing with them. Examples of some things to try:

```python
radius *= 2
spin /= 2
shape = Shady.ComplexPolygonBase( 3 )  # or whatever

shape = Shady.ComplexPolygonBase( 3, joined=True )
field.Set( drawMode=Shady.DRAWMODE.LINE_STRIP, penThickness=1 )
```

```python
Shady.AutoFinish( w )  # Finish up the demo (in case we're not running with $ightarrow$ 'python -m Shady')
```

**examples/dots4.py**

This is one of the *example scripts* included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo dots4`

```bash
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/ .
#
# $END_SHADY_LICENSE$
#
#: Interactive real-time demo of the power of multi-element stimuli
```
This demo shows how large numbers of independent elements can be animated and made to respond in real time.

A single `Shady.Stimulus`, while usually rendered as a rectangle, can be rendered as anything up to 20,000 independent points. If these are managed as a `numpy` array, and manipulated in a "vectorized" way such that the bulk of the arithmetic operations happen in compiled binaries, these points can be animated in quite sophisticated ways, well within 60 fps deadlines (verify this with `--gauge` and/or `--debugTiming` provided you have numpy and matplotlib installed).

An `n`-sided polygon uses up `n+1` points, so for polygons, when you specify `--nsides=3` or above, the number of independent polygons `--ndots=m` is limited by `(n+1)*m <= 20000`.

A line segment uses up just 2 points, so for line segments, you can specify either `--nsides=2` or `--nsides=1` (both are equivalent) and then the number of independent segments can be anything up to `--ndots=10000`.

Move the mouse around / touch the touch-screen to interact with the shapes.

```python
if __name__ == '__main__':
    import Shady

    cmdline = Shady.WorldConstructorCommandLine()

    ndots = cmdline.Option( 'ndots' , 3000, type=int, container=None, doc='Number of independent shapes.' )
    nsides = cmdline.Option( 'nsides', 2, type=int, min=1, container=None, doc='1 or 2 for line segments: ndots <= %d
3+ for polygons: ndots*(nsides+1) <= %d' % ( Shady.Rendering.MAX_POINTS / 2, Shady.Rendering.MAX_POINTS ) )
    radius = cmdline.Option( 'radius', 10, type=( int, float ), container=None, doc='Dictates the half-width of each shape, in pixels.' )
    spin = cmdline.Option( 'spin', 0.2, type=( int, float ), container=None, doc='Max. number of revolutions per second of each shape around its own center.' )
    energy = cmdline.Option( 'energy', 0.3, type=( int, float ), container=None, doc='Larger numbers make the storm winds blow harder.' )
    dims = cmdline.Option( 'dims', ( 0, 1 ), type=( tuple, list ), doc='Which two of the three dimensions of the attractor should be plotted? Possibilities: 0,1 1,0 0,2 2,0 1,2 2,1' )
    gauge = cmdline.Option( 'gauge', False, type=bool, container=None, doc='Whether or not to show a `FrameIntervalGauge`.' )
    cmdline.Help().Finalize()
    Shady.Require( 'numpy' ) # die with an informative error if this is missing
```

(continues on next page)
First let's create a shape. For \( nsides > 2 \) we'll use a polygon, delimited by a NaN. A special case will be \( nsides=1 \) or \( nsides=2 \) both of which will be interpreted to mean that each shape is a single line segment. In that case the "LINES" draw-mode will be used, and that draws a separate line segment connecting the points in each successive pair, so we can omit the NaN-break between shapes (we don't have to, but it increases the max number of shapes we can use by 50%).

```python
if nsides == 1: nsides = 2
shape = Shady.ComplexPolygonBase( nsides, appendNaN=nsides>2 ) # 1-by-
→(nsides+1) complex
    if shape.size * ndots > Shady.Rendering.MAX_POINTS:
        raise ValueError( 'too many points: (nsides + 1) * ndots should be <=
→%d' % Shady.Rendering.MAX_POINTS )
```

Create the World according to the usual command-line opts

```python
w = Shady.World( bg=0.2, **cmdline.opts )
if gauge: Shady.FrameIntervalGauge( w )
```

Create a field on which to draw multiple copies of the shape:

```python
field = w.Stimulus(
    anchor = -1,        # position its bottom-left corner...
    position = w.Place( -1 ), # in the bottom-left corner of the window
    size = w.size,      # and (potentially) fill the window
    noise = -0.5,       # with high-contrast uniform noise...
    drawMode = Shady.DRAWMODE.POLYGON if nsides > 2 else Shady.DRAWMODE.
→LINES,  # ...but only where the shapes are
→)
```

Define a center to the maelstrom, and allow the user to move it with the mouse or touch-screen:

```python
w.eyeOfStorm = field.Place( 0, 0, False )
@w.EventHandler( slot=-1 )
def Interact( self, event ):
    if event.type in [ 'mouse_motion', 'mouse_press' ]: # and 'left' in
→
→event.button.split():
        self.eyeOfStorm = Shady.RelativeLocation( [ event.x, event.y
→], field )
```

Choose which two dimensions of the 3-D attractor will be projected onto the screen:

```python
realdim, imagdim = dims
scale = min( w.size ) / 25.0
```

```python
def OriginXYZ():
    global realdim, imagdim # just to make things more readable later on
    realdim, imagdim = dims
    out = [ 0.0, 0.0, 0.0 ]
```
out[ realdim ], out[ imagdim ] = w.eyeOfStorm
if 2 in dims: out[ 2 ] -= scale * 10.0
return out

""
Initialize the position and angular velocity of each shape:
"

import numpy

start = [ OriginXYZ()[ i ] + scale * numpy.random.uniform( -10, 10, ndots ) for i in range( 3 ) ] # 3-by-ndots real coords X,Y,Z

spin = numpy.random.uniform( -spin, spin, [ ndots, 1 ] ) # ndots-by-1 real

""
Set up the differential equations of the attractor:
"

a, b, c = 10.0, 28.0, 8.0 / 3.0 # the Lorenz attractor's three magic numbers
denom = 2; a /= denom; b /= denom; c /= denom # necessary if using

def deriv( t ):
    x, y, z = tap() # defined below: will provide a view into the
    x0, y0, z0 = OriginXYZ()
    x = ( x - x0 ) / scale
    y = ( y - y0 ) / scale
    z = ( z - z0 ) / scale
d_dt = [ a * ( y - x ), x * ( b - z ) - y, x * y - c * z ]

    # Lorenz attractor equations
    return energy * scale * numpy.vstack( d_dt ) # output is 3-by-ndots,
    # real velocities dX/dt, dY/dt, dZ/dt

    ""
Build a `Shady.Function` object that integrates
the above function:
"

func = Shady.Integral( deriv, initial=start )
tap = func.Tap( initial=start ) # allows the Function values to be inspected

""
Go!
"

field.points = func # dynamic value assignment to the managed `Stimulus`

""
This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo dynamic-range

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/ .
#
# $END_SHADY_LICENSE$

#: Noisy-bit dithering and bit-stealing under the microscope

""
This is a demo of the `Loupe` utility function. It creates a `Stimulus` that allows you to examine other stimuli empirically with enhanced contrast, spatial magnification, and temporal sub-sampling. So it's good for verifying the content of very low-contrast stimuli and the behavior of the dynamic-range enhancement tricks (bit-stealing or noisy-bit dithering) that enable them. This demo will allow you to explore `Loupe` behavior with keyboard commands. It requires the third-party packages `numpy` and `pillow`.

The concepts explored in this demo are explained in greater detail in the topic documentation:

>>> help( Shady.Documentation.PreciseControlOfLuminance )

or:

In [1]: Shady.Documentation.PreciseControlOfLuminance?

"""
```
if __name__ == '__main__':
    
    Parse command-line options:
    
    import Shady
    cmdline = Shady.WorldConstructorCommandLine()
```

(continues on next page)
gauge = cmdline.Option( 'gauge', False, type=bool, container=None, doc="Whether or not to show a `FrameIntervalGauge`.")
global_gamma = cmdline.Option( 'gamma', -1, type=( int, float, tuple, list ), container=None, min=-1, minlength=3, maxlength=3, doc="Gamma-correction (when enabled). -1 means sRGB")
cmdline.Help().Finalize()

""
We're going to need the `Shady.Text` plugin---be warned that that sometimes takes several seconds to import. Text-rendering also entails dependency on two third-party packages: `numpy` and `pillow`.
"

import Shady.Text

Create the World. Add a FrameIntervalGauge if requested:

Create the Stimulus that a subject would actually see. We'll use a Gabor patch.

ideal_size = 500
size = int( min( ideal_size, world.width / 3.0 ) )
shrink = size / float( ideal_size )
margin = 100 * shrink

gabor = world.Sine( # convenience wrapper round `world.Stimulus`
    size = size,
    signalFrequency = 0.0125,
    plateauProportion = 0,
    position = world.Place( -1, 0 ) + [ margin, 0 ],
    anchor = [ -1, 0 ],
)

Create our diagnostic tool:

enhanced = Shady.Loupe(
    target = gabor,
    update_period = 1.0,
    scaling = 4,
    position = gabor.Place( +1, 0 ) + [ margin, 0 ],
    anchor = [ -1, 0 ],
)

Compute a bit-stealing LUT, or load a pre-computed one:

lutArray = Shady.BitStealingLUT(
    maxDACDeparture = 2,
    Cmax = 3.0,
    nbits = 16,
    gamma = global_gamma,
DACbits = world.dacBits,
cache_dir = Shady.PackagePath( 'examples' ),
    # should pick up 'examples/BitStealingLUT_maxDACDeparture=2_Cmax=3.0_
    # unless you have requested a different --gamma, or your graphics card is not 8-bit
    # (in which case it will take a little extra time to calculate the LUT)
lutObject = world.LookupTable( lutArray )  # keep this for later


###
Accurate rendering breaks down as contrast gets low (close to detection threshold). Exactly *how* it breaks down depends on whether the background luminance is an integer DAC value (say, 127) or not (say, 127.5 which is the true mid-point of an 8-bit DAC). We want to be able to visualize both cases. Let’s create a function that allows us, regardless of whether we’ve got gamma correction turned on or off, to choose between a background luminance that maps to the nearest integer DAC value, and one that maps halfway between the two nearest integer DAC values.
###:
gabor.rounding = True
approximateBackground = [ 0.5, 0.5, 0.5 ]
targetDAC = []
@world.AnimationCallback
def WrangleBackground( t=None):
    gamma = [ 1.0, 1.0, 1.0 ] if gabor.lut else gabor.gamma
    targetDAC[ : ] = [ int( world.dacMax * Shady.Linearize( val, gamma=g )
        ) + ( 0 if gabor.rounding else 0.5 ) for val, g in zip( approximateBackground,
gamma ) ]
    gabor.bg = [ Shady.ScreenNonlinearity( val / float( world.dacMax ),
        gamma=g ) for val, g in zip( targetDAC, gamma ) ]
    WrangleBackground()

###
Create a dynamic text stimulus that reports all the relevant information about the Gabor’s current linearization and dynamic-range enhancement settings:
###:
def Caption( t ):
    txt = '#g# contrast
% ( gabor.contrast * 100 )
    if gabor.lut or gabor.rednoise: txt += 'dithering on\n' % gabor.dithering
    else: txt += ('dithering off\n' if gabor.ditheringDenominator <= 0.0,
    else 'dithering on\n') + 'gamma = ' + str( list( gabor.gamma ) )
    if gabor.lut or gabor.rednoise: txt += '\nnoise = %g' % gabor.rednoise
    txt += '\nraw BG = %r' % targetDAC

    return txt

    msg = world.Stimulus( position=gabor.Place( -1, -1.2 ), anchor=[ -1, +1 ],
text=Caption, text_align='left', text_size=35 * shrink )

###
Things only start to look interesting at lower contrasts, so let’s start you there:
###:
gabor.contrast = 0.0625

(continues on next page)
Register an event-handler that lets us play with the parameters:

```python
@world.EventHandler( slot=-1 )
def KeyboardControl( self, event ):
    if event.type in [ 'key_release' ]:
        if event.key in [ 'right' ] and enhanced.update_period > 0.0:
            enhanced.update_period /= 2.0
        if event.key in [ 'left' ] and enhanced.update_period < 30.0:
            enhanced.update_period *= 2.0
        if event.key in [ 'down' ]:
            gabor.contrast /= 2.0
        if event.key in [ 'up' ]:
            gabor.contrast *= 2.0
        if event.key in [ 'l' ]:
            gabor.rounding = not gabor.rounding; WrangleBackground()
        enhanced.DeferredUpdate()
```

Print, and render, a reminder of the keyboard commands:

```python
instructions = ""
up / down : raise/lower contrast
left / right : slower/faster capture rate
D : toggle dithering
G : toggle gamma-correction
N : toggle additive noise
L : toggle look-up table
B : toggle integer/non-integer background DAC
+ / - : increase/decrease magnification"
```

```python
legend = world.Stimulus(
    text = instructions.strip( '\n' ),
    text_size = 20 * shrink,
    position = gabor.Place( 0, +1.2 ),
    anchor = ( 0, -1 ),
    z = +0.5,
)
print( instructions )
```

Remember: the Loupe does not fake the effects of dithering and bit-stealing: it actually examines them empirically, enhancing them artificially so you can see them. You can put whatever content you like into the target 'Stimulus' ('gabor' in this case) and see the effects in the Loupe.
This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo dynamics1`

```python
if __name__ == '__main__':
    """
    Let's start by creating a World, configured according to whatever command-line arguments you supplied.
    """
    import Shady
    from Shady.Dynamics import StateMachine, Integral, Transition

    # $BEGIN_SHADY_LICENSE$
    #
    # This file is part of the Shady project, a Python framework for real-time manipulation of psychophysical stimuli for vision science.
    #
    # Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
    #
    # Shady is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
    #
    # This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
    #
    # You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.
    #
    # $END_SHADY_LICENSE$

    #: High-level tools for directing the action (procedural version)
    """
```

5.3. Getting Started

(continues on next page)
cmdline = Shady.WorldConstructorCommandLine()
cmdline.Help().Finalize()
Shady.Require( 'numpy', 'Image' ) # die with an informative error if either is missing

world = Shady.World( **cmdline.opts )

***
We'll use our included example image, 'alien1', as a stimulus. Note that we're telling Shady to load all of the individual frame images from the 'alien1' folder, not the `.gif` file.

We'll start our alien at the left end the world.
***
alien = world.Stimulus(
    Shady.PackagePath( 'examples/media/alien1/*.png' ),
    x = -world.width / 2,
)

***
Now we'll start constructing the pieces of our dynamic world.

We want our alien to stand, run, jump, fall, in that order. He will also spiral out of control if you press a certain key. These states will be fairly simple: the running and spiraling states will be triggered by user key presses, and the will all run on timers (running will lead to jumping which will lead to falling, although a key-press will also speed up the jump).

Shady's `StateMachine` class can handle all most of this automatically by stringing states together in a certain order, but if we want a state to have multiple possible next states, we'll need to define the logic ourselves, as we shall see.
***

We'll start by defining the state machine, which we imported from the Shady.Dynamics module.
***
sm = StateMachine()
***
Let's add our five states one by one. We'll define what happens in each state later. Right now, we want to set up how the *changes* between states will work.

First, we add the 'stand' state, already alluded to in the function above. We tell the state machine the name of the state, and also tell it which state (yet to be defined) will come after this state in the default ordering of the machine.
***
sm.AddState( 'stand', next='run' )
***
We will define 'run' next. Same thing, but we're also
giving this state a duration in seconds before it automatically moves to the next state.

```python
sm.AddState( 'run', duration=2, next='jump' )
```

Same for 'jump' and 'fall'.

```python
sm.AddState( 'jump', duration=0.4, next='fall' )
sm.AddState( 'fall', duration=1.0, next='stand' )
```

For the 'spiral' state, instead of simply specifying the name of its subsequent state as a string literal, we'll define `next` as a *function* that outputs the name of the next state. The state machine will run this function, passing an instance representing the current state (whose name will be 'spiral') as its argument. The function's output will determine the alien's fate at that time.

```python
def ExitSpiral( state ):
    if state.elapsed < 3:
        alien.frame = Transition( 0, 24, duration=0.1 ) # wiggle
        return StateMachine.CANCEL
    return 'stand'
```

```python
sm.AddState( 'spiral', next=ExitSpiral )
```

We've defined the skeleton of our state machine, but none of the actual behavior associated with its states.

We'll give our stimulus an animation callback, which is just a function that is called by Shady on every frame before drawing this stimulus. The animation callback takes the stimulus itself as its first argument and the stimulus time (by default: seconds since World creation) as its second argument.
# Calling the StateMachine instance with time `t` is what actually causes the state-transition logic and management to run. It returns an instance of a `State` object:

```python
state = sm(t)
```

# The `State` instance has an attribute `.fresh` which indicates whether the current time `t` is the earliest time for the current state. It is a handy way of implementing state onset behaviors procedurally.

```python
if state.fresh:
    # Now we'll just check which state we're in and change the alien accordingly. State instances can be compared directly to strings to check their names.
    if state == 'stand':
        self.xy = 0  # stop any dynamic attached to the .xy property
        self.Set(x=-world.width/2 + 100, y=0, frame=0)  # re-position him

    # So far so static. In the other states, we'll use more 'Shady.Dynamics' tricks---in particular, the 'Integral' with respect to time.
    if state == 'run':
        self.Set(x=Integral(400) + self.x, frame=Integral(16))

    # A double-integral of a constant will give us the constant acceleration that you would get from gravity, in both the 'jump' and 'fall' states.
    if state == 'jump':
        self.Set(y=Integral(Integral(-5000) + 1500) + self.y, frame=0)

    if state == 'fall':
        self.frame = Integral(160)  # heeeeeelp

    # Finally we'll use non-linear transformations of an 'Integral' to achieve a faster-and-faster spiralling effect in the 'spiral' state:
    if state == 'spiral':
        radius = min(world.width, world.height) / 2 - 100
        self.Set(x=-radius, y=0, frame=0)  # stop any .x, .y and .frame dynamics

        self.xy = Integral(Integral(0.1)).Transform(Shady.Sinusoid, phase_deg=[270, 180]) * (radius - Integral(20)).Transform(max, 0)
```

(continues on next page)
Our alien will do things now, but we haven't yet implemented a way of getting him started, since there's no condition to terminate the first 'stand' state. We'll hook into Shady's `EventHandler` mechanism.

Like the 'AnimationCallback', this can be done using a decorator on our function definition. The function should take two arguments: the `Shady.World` being hooked into (here, written as 'self') and the event that will be handled, which contains data about mouse and keyboard input.

```python
@world.EventHandler
def KeyboardControl(self, event):
    # Press space to start running, to jump, or to exit from a spiral:
    if event.type == 'text' and event.text == ' ':
        if sm.state in ['stand', 'run', 'spiral']:
            sm.ChangeState() # without args: change to the '.'
        # next state
        # Press enter to start spiralling:
        if event.type == 'key_release' and event.key in ['enter', 'return']:
            sm.ChangeState('spiral') # change to this explicitly named
        # state
        # Press q or escape to stop and close the window:
        if event.type == 'key_release' and event.key in ['q', 'escape']:
            self.Close()
    #
    # And that's it - the alien is away! Try using spacebar and enter key to manipulate his state.

    Remember that we designed `ExitSpiral` to ensure that you cannot get out of a spiral until you have been in it for at least 3 seconds.

    Shady.AutoFinish( world )
```

**examples/dynamics2.py**

This is one of the *example scripts* included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo dynamics2`

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it

(continues on next page)```
This demo shows a different way to create the interactive, dynamic behavior seen in the `dynamics1` demo. We recommend that you go through that demo first, because this demo assumes you’re aware of the behavior we’re trying to achieve.

Rather than keeping the state objects very simple (just `duration` and `next` attributes) and packing all of the logic into a big procedural switch inside the Stimulus `Animate()` method, this version of the same demo compartmentalizes the logic within each state itself. Each state has `onset`, `ongoing`, and `offset` methods assigned to it that are respectively called when the state begins, as the state progresses, and when the state ends.

As before, most of this demo is about walking you through the setup before running the final product at the end.

```python
if __name__ == '__main__':
    ""
    Let’s start by creating a `World`, configured according to whatever command-line arguments you supplied, and creating our alien stimulus.
    ""
    import Shady
    from Shady.Dynamics import StateMachine, Integral, Transition

    cmdline = Shady.WorldConstructorCommandLine()
    cmdline.Help().Finalize()
    Shady.Require( 'numpy', 'Image' ) # die with an informative error if either is missing

    world = Shady.World( **cmdline.opts )

    ""
    While we're here showing alternative ways of doing the things we did before, we'll load the alien using the animated `.gif` file rather than its individual image frames:
    ""
    alien = world.Stimulus(
        Shady.PackagePath( 'examples/media/alien1.gif' ),
        x=-world.width / 2,
    )
```
In `dynamics1`, we started off by defining `duration` and `next` for each state as keyword arguments of the `AddState` method. This time, we'll fully define each state as a subclass of `StateMachine.State`, attach `.next`, `.duration`, etc as class attributes, and load the class definitions into the state machine as our final step.

Our `Stand` state is fairly simple. We define the `next` attribute. All such attributes may be either constant, or callable with the `State` instance as sole argument.

```python
class Stand( StateMachine.State ):
    next = 'Run'
```

Note that a callable class attribute that takes an instance of the class as its first argument is better known as... a method. So let's call it that. And define another method, called `onset`. This method does exactly what we previously accomplished by checking `state.fresh` in the procedural `dynamics1`: its code will be run once each time we enter the state.

```python
def onset( self ):
    alien.xy = 0  # stops any ongoing .xy dynamic
    alien.Set( x=-world.width / 2 + 100, y=0, frame=0 )
```

Sometimes, as in the `onset` example here, your implementation of a `State` method might ignore the instance argument `self`. Your Python IDE may then give you a warning about this. You can suppress such warnings by putting the `@staticmethod` decorator above the method definition, if the warning bothers you more than this extra clutter does. Either way, functionality is the same.

```python
Our `Run` state is also similar to the 'run' section of the animation callback in 'dynamics1', but we've spiced things up a bit by making our alien walk with variable speed as a function of his gait cycle. We've also made the duration attribute a little longer than last time, to accommodate the funky gait.
```

```python
class Run( StateMachine.State ):
    duration = 3.5
    next = 'Jump'

    def onset( self ):
        gait = lambda t: 1 if alien.frame in [ 0, 12 ] else 10
        alien.Set( x=Integral( gait ) * 40 + alien.x,
                   frame=Integral( gait ) * 3 + 1,
                   )
```

(continues on next page)
Shady Documentation, Release 1.9.1

Nothing has changed in our 'Jump' state. (Note, though, that because we set `frame` to 0 for the jump, the alien's dynamic 'gait' and hence his horizontal velocity automatically become constant.)

```python
class Jump(StateMachine.State):
    duration = 0.4
    next = 'Fall'

    def onset(self):
        alien.Set(y=Integral(Integral(-5000) + 1500) + alien.y, frame=0)

The 'Fall' state is a good place to demonstrate the `offset` method (called when the state machine *leaves* the state in question) and the `ongoing` method (which is called repeatedly while we're in the state, every time the state machine itself is called with a new time argument).

```python
import random

class Fall(StateMachine.State):
    duration = 3
    next = 'Stand'

    def onset(self):
        alien.frame = Integral(160)

    def ongoing(self):
        if alien.y < -world.height / 2: return 'Stand'

Typically, the `offset` method is best used to 'clean up' things that may have been affected by the departing state. This method will be called regardless of which state is coming up next, so it's not a good place to initialize what's coming next (use the next state's `onset` for that). Here, we'll assume that the alien falls through a wormhole into a parallel 'World' of a different (randomly-chosen) color.

```python
    def offset(self):
        world.clearColor = [random.random(), random.random(), random.random()]

(continues on next page)
Finally, we have our Spiral, which behaves the same as before.

Note that, whereas before we defined an `ExitSpiral` function globally and passed it as the `next` argument to `AddState`, this time it is very natural to implement the same logic directly in `next` as a method:

```python
class Spiral(StateMachine.State):
    def next(self):
        if self.elapsed < 3:
            alien.frame = Transition(0, 24, duration=0.1)  # wiggle
        return StateMachine.CANCEL
        return 'Stand'

    def onset(self):
        radius = min(world.size) / 2 - 100
        alien.Set(x=-radius, y=0, frame=0)  # stop any .x, .y and .

        alien.xy = Integral(Integral(0.1)).Transform(Shady.Sinusoid, phase_deg=[270, 180]) * (radius - Integral(20)).Transform(max, 0)

    # Now that we have our states, we can define the state machine
    # and pass all of our states as constructor arguments. In the same
    # line, we've set the alien's `.Animate` attribute to the state
    # machine, which tells Shady that this alien should call the
    # state machine once every frame. This saves us the effort of writing:

    @alien.AnimationCallback
    def TediousAnimationDefinition(self, t):
        sm(t)

    in order to attach the state machine to the alien.

    # Sidebar: we took the explicit route to state machine construction
    # there, but you could equivalently, if you find it more readable,
    # construct an empty StateMachine first, and then call `sm.AddState()`
    # with each of the states. The neatest way to do this is with a class
    # decorator:

    sm = StateMachine()

    @sm.AddState
    def Stand(StateMachine.State):
        ...

    The decorator tells Shady to add the State subclass to the
    StateMachine immediately after it is defined.

    # Finally, we'll create the same Event Handler we had in `dynamics1`.

    (continues on next page)
The only difference is that our state names are now capitalized, (because they're classes, and within Shady that is our convention for class names).

```python
@world.EventHandler
def KeyboardControl( self, event ):
    if event.type == 'text' and event.text == ' ':
        if sm.state in [ 'Stand', 'Run', 'Spiral' ]:
            sm.ChangeState()
    elif event.type == 'key_release' and event.key in [ 'enter', 'return' ]:
        sm.ChangeState( 'Spiral' )
    elif event.type == 'key_release' and event.key in [ 'q', 'escape' ]:
        self.Close()
```

All in all, there have been a few changes (some subtle, some not subtle) relative to the alien's previous adventure, but broadly the behavior is the same as in `dynamics1` despite the very different programming approach.

```python
Shady.AutoFinish( world )
```

### examples/events.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo events`

```
#!/usr/bin/env python
#
# This file is part of the Shady project, a Python framework for real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.
#
#: Exploring the event-handling system
```

This demo illustrates Shady's event-handling system.
```python
if __name__ == '__main__':
    import Shady

    cmdline = Shady.WorldConstructorCommandLine(
        width=200, height=200, top=50, frame=True, fullScreenMode=False
    )
    cmdline.Help().Finalize()

    world = Shady.World(**cmdline.opts)

    circle = world.Stimulus(size=[150,150], pp=1, bgalpha=0, color=1)

    world = Shady.World(**cmdline.opts)

    And we'll put a white circle in the middle of it, because why not:

    Every time a keyboard, mouse, or other relevant windowing-system event occurs, an object representing that event is passed through a cascade of the `World`'s event-handler functions. Each handler is associated with a numeric "slot" number, and the handlers are called in increasing slot order. If a handler returns any kind of truthy value after processing a particular event, the cascade is aborted for that particular event.

    By default, a `World` has one handler, in slot 0, and that is its `HandleEvent` method (so you can overshadow this if you make a `World` subclass---then you don't have to do anything else).

    Event handlers can be removed or replaced using the `.SetEventHandler` method, or the `.EventHandler` decorator. All event handlers must take two arguments: `(world, event)`. The default implementation of `HandleEvent` is along the following lines:

    As a simple illustration, we'll implement an additional handler, in slot -1 so that it runs just *before* the one above that implemented q-or-escape-to-close.

    This one will establish a kind of mouse cursor inside the window. It changes color according to whether you're inside or outside the white circle. If you hold down shift as you move, it will stay inside the circle.
```
(continues on next page)
```python
dot = world.Stimulus( color=[1,0,0], bgalpha=0, pp=1, visible=0, size=20 )

@world.EventHandler( slot=-1 )
def MoveDot( self, event):
    if event.type != 'mouse_motion': return

    relX, relY = Shady.RelativeLocation( [ event.x, event.y ], circle, ˓→normalized=True )
    # converts from World-centric to Stimulus-centric coordinates (in this case, normalized)
    normalizedRadialDistance = ( relX ** 2 + relY ** 2 ) ** 0.5
    dot.Set( visible=True, x=event.x, y=event.y, color=[ 1, 0, 0 ] )
    if normalizedRadialDistance > 1.0:
        if 'shift' in event.modifiers.split():
            dot.xy = circle.Place( relX / ˓→normalizedRadialDistance, relY / normalizedRadialDistance )
        # the .Place method converts from normalized Stimulus-centric coordinates
        # to World-centric pixel coordinates
        # TODO: this 'shift' behavior is never triggered in the pyglet back-end
    else:
        dot.color = [ 0, 1, 1 ]

```
The event.x and event.y coordinates are expressed relative to the 'World'’s origin or "anchor" position. Let's add yet another event-handler, to allow you to play with that. It lets you toggle the anchor by pressing the spacebar. After toggling, keep moving the mouse, and observe how the x and y coordinate frame has changed.

Note that our red-dot handler does not need to be updated to accommodate the change of coordinate system. So the effects on the coordinate system will not be immediately obvious to you. However, soon we will be filling your console with event information and then it will make more sense.

```python
world.possibleAnchors = [ [ j, i ] for i in [ -1, 0, +1 ] for j in [ -1, 0, +1 ] ]

@world.EventHandler( slot=-2 )
def ToggleWorldOrigin( self, event ):
    if event.type == 'key_press' and event.key in [ ' ', 'space' ]:
        newAnchor = world.possibleAnchors.pop( 0 )
        world.possibleAnchors.append( newAnchor )
        world.anchor = newAnchor
        msg = ' changed anchor to %r' % newAnchor
        print( '*' * len( msg ), msg, '*' * len(msg) )

@world.EventHandler( slot=-3 )
def PrintEvent( self, thisEvent ):
    # Let's allow the flood of console information to be turned on or off by pressing P
    if thisEvent >> 'kp[p]': # syntactic shorthand---see below
        self.printEvents = not self.printEvents
```

If you ever want to remove a handler from a particular slot, you just set the handler to `None` in that slot:

```python
world.SetEventHandler( None, slot=0 )  # removes our q-or-escape-to-close handler
```

In the end, the best way to learn about any event system is to observe empirically what happens. We’ll set up yet another handler, in slot -3 so that it runs before the others. All the new handler really does is print the event instance, but we’ll add a few bells and whistles.

```python
world.printEvents = True
world.previousEvent = None
@world.EventHandler( slot=-3 )
def PrintEvent( self, thisEvent ):
    # Let's allow the flood of console information to be turned on or off by pressing P
    if thisEvent >> "kp[p]": # syntactic shorthand---see below
        self.printEvents = not self.printEvents
```
elif not self.printEvents:
    return

    # OK, so we're printing. Let's add some context-sensitive
# blank lines to make the output easier to read:
    if self.previousEvent is not None and self.previousEvent.type !=
    thisEvent.type:
        if self.previousEvent.type in [ 'mouse_motion', 'key_release'
    ->]: print( '' )
        elif thisEvent.type in [ 'mouse_motion', 'key_press' ]:
            self.previousEvent = thisEvent

        # Let's report the frame number as well
        thisEvent.frame = self.framesCompleted

        # Main payload:
        print( thisEvent )

    #> 

    Now you can manipulate the mouse and keyboard, and observe what
happens in the event stream (NB: keyboard events are only
registered when the Shady stimulus window is in the foreground).
Here are some final notes before the console gets flooded with
event information:

    Those extra blank lines will generally group sets of related
keyboard and mouse events together, but they are slightly
misleading in the particular case of touch-screen taps, which
generate tight clusters of consecutive `mouse_motion`,
`mouse_press` and `mouse_release` events.

    Note also the shorthand syntax we used above in the last
handler, `thisEvent >> "kp[p]"`. You'll see from the console
output below that each event has a field called `.abbrev`, which
contains an abbreviated form of the event's `.type` and, where
appropriate, its `.key`, `.text` or `.button` content. The
syntax `e >> s` is equivalent to `e.abbrev in s`. This allows
for easier programming of event tests.

    OK, that's all. Try it out...:
    #> 

    Shady.AutoFinish( world )

examples/fancy-hardware.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal
Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo
fancy-hardware

    # BEGIN_SHADY_LICENSE
    #
    # This file is part of the Shady project, a Python framework for
    # real-time manipulation of psychophysical stimuli for vision science.
    #

    (continues on next page)
Demonstration of support for 16-bit mono and 48-bit color modes

Shady was primarily designed to allow psychophysical stimuli to be rendered easily on commodity (non-specialist) hardware, with dynamic range enhancement via "noisy-bit" dithering. However, it also provides support for the high-dynamic range modes of certain specialized display devices, such as the Bits# by Cambridge Research Systems Ltd or the ViewPixx by VPixx Inc. Here we demonstrate how Shady can render stimuli in these special modes.

```python
if __name__ == '__main__':
    ""
    Let's get the command-line arguments out the way first
    ""
    cmdline = Shady.WorldConstructorCommandLine( canvas=True )
    hardware = cmdline.Option( 'hardware', 'auto', type=str, container=None, doc="examples: --hardware=ViewPixx or --hardware=BitsSharp" )
    cmdline.Help().Finalize()
    ""
    Now let's create a World, and a Stimulus:
    ""
    w = Shady.World( **cmdline.opts )
    s = w.Stimulus()
    atmosphere = w,
    signalFunction = Shady.SIGFUNC.SinewaveSignal,
    signalAmplitude = 0.5,
    signalFrequency = 0.005,
    plateauProportion = 0,
    envelopeSize = min( w.size ) / 2 - 50,
    normalizedContrast = Shady.Oscillator( 0.2 ) * 0.5 + 0.5,
    )
    ""'
    Or more concisely:
    ""'
    s = w.SinewaveSignal( sigf=0.01, pp=0.08, size=400, contrast=Shady.Oscillator(0.2) * 0.5 + 0.5)
    ""'
```
We'll manipulate a `World` property called `.bitCombiningMode`. By default, this is 0, or equivalently 'C24', meaning 24-bit color with dithering turned on by default.

Let's set it to 1, also known as 'M16' or 'monoPlusPlus' mode. Dithering will be disabled, and stimuli will be monochrome (their intensity will be determined only by the red component of the signal). Each pixel will be represented as a 16-bit integer with the more-significant byte stored in red channel and the less-significant byte stored in the green. This would get reinterpreted by the Bits# or ViewPixx hardware, and rendered in 16-bit gray-scale.

```
w.bitCombiningMode = 'M16'
```

We can also set it to 2, a.k.a 'C48' or 'colorPlusPlus' mode. Instead of sacrificing color, we now sacrifice resolution. The `World` will change its nominal size: it will now be divided into half the original number of pixels horizontally and (by default) also vertically. The left half of each virtual pixel will contain its more-significant byte, and the right half will contain its less-significant byte. The Bits# or ViewPixx would reinterpret this as a 16-bit-per-channel full-color image.

```
print( ' before: {:4d} x {:4d}'.format( *w.size ) )
w.bitCombiningMode = 'C48'
print( ' after: {:4d} x {:4d}'.format( *w.size ) )
```

Note that the stimulus got physically larger---not because we changed its size in pixels (it's still pixellated at 400 x 400, the way it was defined) but because we doubled the linear physical extent of what the `World` considers to be a "pixel". In real applications, of course you'll set the mode of the `World` at the very beginning, and then create stimuli based on its de-facto `.size`, and there will be no confusion.

```
...no confusion, that is, *unless* you decide that you want to take advantage of full vertical resolution. The Bits# and ViewPixx only require you to reduce *horizontal* resolution, as they combine pairs of horizontally-adjacent pixels in the graphics card's frame buffer, to determine the (yoked) intensity of the corresponding pair of physical pixels. The hardware does not require you to throw away vertical resolution as well, but Shady does so by default, so that the "pixels" you work with remain square and things remain easy to lay out geometrically.

However, if you want to work with non-square pixels, you can. Do you really want to do that?
```

(continues on next page)
Are you sure??

```python
w.SetBitCombiningMode( 'C48', verticalGrouping=1 )
# whereas for this mode, the default verticalGrouping value is 2
```

So now we have a logically square stimulus that appears physically non-square. Perhaps you would like to make it appear square, by doubling the number of pixels at which we sample the signal function vertically?

```python
s.height *= 2
print( 'stimulus: {:4d} x {:4d}'.format( *s.size ) )
```

Just remember, you asked for it. Be careful what you do with geometric transformations of your stimuli. Notice, for example, how with non-square pixels, a simple rotation of the carrier function causes a change in spatial frequency:

```python
s.carrierRotation = Shady.Integral( 20 )
```

...and envelope transformations are even wackier:

```python
s.carrierRotation = 0
s.envelopeRotation = Shady.Integral( 20 )
```

---not that you should be rotating the *envelope* of a precisely-computed stimulus anyway, because of the interpolation artifacts. But this illustrates the more general point about non-square pixels---for laying out stimuli, I recommend sacrificing vertical resolution for the sake of sanity:

```python
# revert the Stimulus to its previous logically-square shape:
s.height /= 2
```

```python
# and sit still, for goodness' sake:
s.envelopeRotation = 0
```
Now let's put each mode's rendering under the microscope. We'll create a stimulus containing only one logical pixel:

```python
p = w.Stimulus( size=1, color=0.5, anchor=-1, pos=w.Place( -1 ),
    →ditheringDenominator=w )
```

We'll write a function that empirically captures, then pretty-prints, the frame-buffer representation of that single logical pixel:

```python
def Pixel( *args ):
    if args:  # you can pass a single value, or three values R, G, B
        p.color = [ arg / 65535.0 for arg in args ]

    image = p.Capture( normalize=False )  # We say normalize=False
    # here because otherwise, when w.bitCombiningMode is > 0, the
    # default behavior of .Capture() would be to re-combine the image
    # bytes and return an image of the correct logical size, with
    # high-dynamic-range pixel values in the range 0 to 1. But in
    # this demo, we actually want to see the raw uncombined bytes.

    PrettyPrint( args, image )  # we defined this function out of
    # sight, because the details are unimportant/distracting.
```

```python
#>
# PrettyPrint( args, image )  # we defined this function out of
# sight, because the details are unimportant/distracting.
```

```python
def PrettyPrint( args, image ):
    if not args:
        args = [ x * 65535.0 for x in p.color ]
    if len( set( args ) ) == 1:
        args = args[ :1 ]
    try:
        import numpy
    except ImportError:
        s = ', '.join( '%3d' % x for x in image )
        labels = ''
    else:
        s = '\n'.join(
            [ '%s' ] * 4 + [ '%s' ] * 3 + [ '%s' ] * 2 + [ '%s' ] * 1 + [ '%s' ] * len( image )
            for pixel in row )
        for pixel in row )
        labels = '# ' + ' R G B A ' * image.shape[ 1 ]
        mode = w.bitCombiningMode
        modeInfo = 'w.bitCombiningMode = %r%s' % ( { 0 : 'C24', 1 : 'M16', 2 →: 'C48' } ).get( mode, mode ), ' if mode else ' (dithering on)' if p.
    if ditheringDenominator > 0 else ' (dithering off) '
    if mode >= 2 and w.pixelGrouping[ 1 ] < 2: modeInfo += ' (but with
    →verticalGrouping=%r)' % w.pixelGrouping[ 1 ]
    s = '\n'.join( '

    Pixel( %s ) --> [' % ' '.join( arg for arg in args ), s, labels )
print( s )

***
Let's see that representation in all the different modes.

In regular C24 mode you'll see dithering (so the exact byte values might be different each time you capture):

```python
w.SetBitCombiningMode( 0 )
Pixel()
Pixel()
Pixel()
```

In the other modes dithering is turned off, and pixel intensities are just rounded to the nearest 1/65535.
Here is 16-bit mono mode:

```python
w.SetBitCombiningMode( 1 )
Pixel()
```

Here is 48-bit color mode with non-square pixels:

```python
w.SetBitCombiningMode( 2, verticalGrouping=1 )
Pixel()
```

...and here is 48-bit color mode with square pixels:

```python
w.SetBitCombiningMode( 2, verticalGrouping=2 )
Pixel()
```

As usual, target intensities are expressed, in `p.color`, as floating-point numbers from 0 to 1. However, note that we wrote our little ad-hoc diagnostic function `Pixel()` such that we can express a new color in 65535ths. So, for example:

```python
Pixel( 65534.5 )  # should round up to 65535 = 0xFFFF = (255,255)
Pixel( 65534.49 ) # should round down to 65534 = 0xFFFE = (255,254)
Pixel( 0, 32767.5, 65535 ) # r, g, b
```

Why not try a few examples yourself, in different modes?

Sometimes you will want to perform gamma-correction using the hardware's own built-in method. For this reason, we've set the default Shady gamma to 1.0 in this demo. However, you can use Shady's built-in gamma-correction in bit-combining modes too, if you want:

```python
w.gamma = 2.2  # or, you know, whatever
```
"""Note that this has affected our Gabor patch `s`, which shares its "atmosphere" properties (including `.gamma`) with the World `w`. But it has not changed the gamma-correction of our single-pixel test stimulus `p`, because we have not linked its `.gamma` to the World in that way:
"""

Pixel( 32767.5 ) # still ends up in the middle of the range

"""
We will leave you in 24-bit color mode, with a low-contrast grating on a low-luminance background. This stimulus makes it relatively easy to see quantization artifacts (especially around the edges) when dithering is turned off. We'll also install an event-handler that lets you toggle dithering on and off by pressing `d` on the keyboard.
"""

```python
w.Set( gamma=1, ditheringDenominator=0, bitCombiningMode=0 )
s.Set( bg=0.1, contrast=0.04 )
# Assuming 8-bit DACs, bg=0.1 at gamma=1 leads to a # background target level of 25.5: the tiniest amount below # that will round to 25, and the tiniest amount above will # round to 26.
@w.EventHandler( slot=-1 )
def ToggleDithering( self, event ):
    if event >> "kp[d]"
        # press d to toggle dithering on or off
        # (NB: only turns on in C24 mode)
        self.ditheringDenominator = 0 if self.ditheringDenominator,
    else
        self.dacMax
```

"""
The final thing to cover is how to use Python to send the appropriate mode-setting signals to the hardware. The manufacturer may already supply Python bindings for this purpose. We have written some bindings of our own for Shady, although due to limitations on access to hardware, we are less likely to be able to maintain them than the respective manufacturers. But if you want to try ours out, you can import machine-specific classes from Shady's optional manufacturer-specific submodules.

If you're running this demo on such a device, let's try it. If you have such a device connected but are currently running this demo on the wrong screen, you should at this point type `exit()` and then re-start the demo but with the appropriate `--screen` number specified on the command line---for example:

```
python -m Shady demo fancy-hardware --screen=2
```

"""

First let's install another event-handler, in a different
slot from the dithering toggle. Once we've set up the 'device' instance in the next step, this handler will allow you to switch between modes with the keyboard, by pressing 0, 1, 2, or shift+2:
```python
#:
device = None
@w.EventHandler( slot=-2 )
def ChangeDeviceAndWorldMode( self, event ):
    if device and event >> "kp[0]":
        # press 0 for 24-bit color mode without dithering
        self.ditheringDenominator = 0
        try: device.mode = 'C24'
        except RuntimeError as err: print( err )
        except ValueError as err: print( err ) # because BitsSharp does not recognize this mode
    if device and event >> "kp[1]":
        # press 1 for 16-bit mono mode
        try: device.mode = 'M16'
        except RuntimeError as err: print( err )
    if device and event >> "kp[2]":
        # press 2 for 48-bit color mode
        try: device.mode = 'C48'
        except RuntimeError as err: print( err )
    if 'shift' in event.modifiers:
        self.SetBitCombiningMode( 'C48', verticalGrouping=1 )

def wrapInput( prompt ):
    try: func = raw_input
    except: func = input
    try: response = func( prompt ).strip()
    except EOFError: response = ''
    print( '' )
    return response
```

```python
try: _SHADY_CONSOLE_INTERACT
except NameError:
    if hardware == 'auto': hardware = 'None'; print( 'No --hardware was specified.' )
    else:
        if hardware != 'auto': print( 'You specified --hardware=%s' % hardware )
```

```python
if hardware == 'auto':
    """
    Are you running this demo on a ViewPixx or similar display by VPixx, Inc.?

    Is it connected to this computer via USB?

    Are the appropriate drivers installed?
    """""
(continued from previous page)

```python
if wrapInput( "Use VPixx device? y/[n]: " ).lower().startswith( 'y' ):  
    hardware = 'ViewPixx'

if hardware == 'auto':
    ""
    Are you running this demo through a Bits# or similar stimulus
    generator by CRS Ltd.?
    Is it connected to this computer via USB?
    Are the appropriate drivers installed and do you know the
    serial port address?
    Have you installed the third-party python package `pyserial'?"#
    if wrapInput( "Use CRS device? y/[n]: " ).lower().startswith( 'y' ):  
        hardware = 'BitsSharp'

if hardware.lower() in [ 'vpixx', 'viewpixx', 'datapixx' ]:
    "OK, let's set it up:"#:  
    from Shady.VPixx import ViewPixx  
    device = ViewPixx( w )

elif hardware.lower() in [ 'crs', 'bits#', 'bits++', 'display++', 'bitssharp',
                        'bitsplusplus', 'displayplusplus' ]:
    "Which serial port is it on?"#>
    serialAddress = wrapInput( "Enter serial port address (e.g. COM19 or /
dev/tty.usbmodem14111 ): " ):
    "OK, let's set it up:"#:  
    from Shady.CRS import BitsSharp # NB: this will fail if pyserial isn
    "t installed
    device = BitsSharp( serialAddress, world=w )

elif hardware.lower() not in [ 'none', 'auto' ]:
    print( "Unrecognized hardware $r' % hardware )

w.ditheringDenominator = 0
if device:
    print( 'Created $r' % device )
else:
    ""
    Alrighty then, come back when you have set up your device.
    ""
    @w.EventHandler( slot=-2 ) # equivalent to the handler above, but
    without the device
    def ChangeModeEvenWithoutHardware( self, event ):  
        event >> "kp[0]" and self.Set( bitCombiningMode=0,
        ditheringDenominator=0 )
        event >> "kp[1]" and self.SetBitCombiningMode( 1 )
        event >> "kp[2]" and self.SetBitCombiningMode( 2,
        verticalGrouping=1 if 'shift' in event.modifiers else 2 )

print( ""
```

(continues on next page)
Use the keyboard (0, 1, 2, shift+2) to change mode.
Press d to toggle dithering on/off in mode 0 (24-bit color).
Note the effect on the quantization artifacts.

```python
Shady.AutoFinish( w )  # tidying up in case we didn't get here via `python -m Shady examples/foreign-stimulus.py
```

**examples/foreign-stimulus.py**

This is one of the *example scripts* included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with `python -m Shady demo foreign-stimulus`

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/ .
#
# $END_SHADY_LICENSE$

#: How to add "foreign" (non-Shady) stimuli to a World

Shady lets you draw anything you like among its stimuli. This script demonstrates how you can write a custom class, for use as a "foreign" (non-Shady) stimulus. The only thing it needs is a `.Draw()` method that takes no additional arguments. You are free to use any OpenGL drawing commands you want in this method (the only caveat being that, depending on whether you are using "legacy" or "modern" OpenGL commands, you should ensure that the `World` is created with 'legacy=True' or 'legacy=False' appropriately). The current demo uses certain Shady features as helpers---specifically the `World` projection matrix, and an `Integral` instance---but in principle there is no need to refer to Shady: you can do whatever you want.

This demo requires third-party modules `OpenGL` (from the

(continues on next page)
```
import random

if __name__ == '__main__':
    import Shady
    Shady.DependencyManagement.Define( 'OpenGL', packageName='pyopengl' )

    cmdline = Shady.WorldConstructorCommandLine( legacy=True )
    cmdline.Help().Finalize()
    Shady.Require( 'OpenGL', 'numpy' )

    Create a World:
    w = Shady.World( **cmdline.opts )
    w.perspectiveCameraAngle = 120 # default is 0, which means orthographic

    Let's define a Shady stimulus for our foreign stimulus to interact with. Normally the `PixelRuler()` stimulus is created at a positive depth to ensure it lies behind most other stimuli, but here we'll explicitly reset its `.z` to 0 (the default for most stimuli). Shady's convention for perspective projections is that stimuli at `.z=0` are pixel-for-pixel; at closer and further depths, pixels will start to get interpolated.

    grid = Shady.PixelRuler( w ).Set( z=0, carrierTranslation=w.size / 2 )

    Import OpenGL command bindings, from the third-party `PyOpenGL` package. (NB: you could use `pyglet.gl` here instead of `OpenGL.GL`, but pyglet does not wrap any of the gl* functions, which means some of them are less easy-to-use: for example you would have to wrap sequences by hand in `ctypes` containers---it's easier just to install PyOpenGL).

    from OpenGL.GL import *

    Now we'll define a custom stimulus class. From Shady's point of view the only requirement is that it should have a method called `.Draw()` or `.draw()`, taking no arguments. There is no need for the class to contain any reference to Shady classes or
```
functions—you can work entirely in OpenGL.

```python

class CustomStimulus( object ):
    def __init__( self ):
        self.vertexColors = {}
        self.last_normal = None
        self.boring = False

        self.angular_velocities = [ 23, 31, 37 ]
        self.rotation = Shady.Integral( lambda t: self.angular_ velocities )

        glEnable( GL_LIGHTING )
        glEnable( GL_LIGHT0 )

    def Normal( self, x, y, z ):
        """
        Define a normal vector (in legacy-OpenGL "direct mode"),
        and remember it (for the purpose of recalling which color
        belongs to which face, when `.boring` is set to `True`).
        """
        glNormal3f( x, y, z )
        self.last_normal = ( x, y, z )

    def Vertex( self, x, y, z ):
        """
        Draw a vertex (in legacy-OpenGL "direct mode") and choose a
        color for it if not already chosen.
        """
        if self.boring: key = self.last_normal # different solid
        else: key = ( x, y, z ) # different color at
        color = self.vertexColors.get( key, None )
        if color is None: color = self.vertexColors[ key ] = self.
        glColor4f( *color ) # only respected if lighting is disabled
        if lighting is enabled:
            glMaterialfv( GL_FRONT, GL_AMBIENT, color ) # only respected
        if lighting is enabled:
            glMaterialfv( GL_FRONT, GL_DIFFUSE, color ) # only respected
        glVertex3f( x, y, z )

    def NewColor( self ):
        color = [ random.random() for channel in 'rgb' ]
        if len( color ) < 4: color.append( 1.0 )
        return color

    def Draw( self ):
        # In this demo we'll use the GL_PROJECTION and GL_MODELVIEW,
        # which
```

(continues on next page)
# is a deprecated ("legacy") OpenGL feature.

### (1) Projection

```python
glMatrixMode(GL_PROJECTION)
glPushMatrix()

# Get the matrix product of w.matrixWorldNormalizer
# and w.matrixWorldProjection (NB: requires numpy):
m = w.projectionMatrix

# In the orthographic case, the z clipping planes are at z=-1
# and z=+1.

# So let's change the z-scaling to give ourselves some room

if 0.99 < m[2, 2] < 1.01:
m[2, 2] = 2.0 / w.width

# Transfer the projection matrix:
glLoadMatrixf(list(m.T.flat))
```

### (2) Scene composition

```python
glMatrixMode(GL_MODELVIEW)
glPushMatrix()

# Foreign stimuli don't have dynamic properties (they have no
# features

# but let's

# make a poor-man's version here, to allow z to be dynamic:
z = self.z

if callable(z):
z = z(w.t)

# fixed-function-pipeline transformations:
glTranslatef(0, 0, z)  # translation to the desired z
glRotatef(omega, phi, kappa)  # rotation about x, y and z axis as a

r = 100  # half the length of one side of the cube, in pixels
N = self.Normal
V = self.Vertex

glBegin(GL_QUADS)  # this is "direct-mode" drawing (also a

N(-1, 0, 0); V(-r,-r,-r); V(-r,-r,+r); V(-r,+r,-r); V(-r,+r,+r)
N(+1, 0, 0); V(+r,+r,-r); V(+r,+r,+r); V(+r,-r,+r); V(+r,-r,-r)
N( 0,-1, 0); V(-r,-r,-r); V(+r,-r,-r); V(+r,-r,+r); V(-r,-r,
N( 0, 1, 0); V(-r,+r,+r); V(+r,+r,-r); V(+r,-r,-r); V(-r,+r,
```

(continues on next page)
Now that it’s defined, we’ll add it to the World. We’ll give it the actual class object, rather than an instance of that class. When the method ‘.AddForeignStimulus()’ receives a callable object, it calls it to obtain an instance. This is an easy way of ensuring that all those OpenGL calls are “deferred” into the correct thread where necessary:

```python
#:
c = w.AddForeignStimulus( CustomStimulus, z=-100 )
```

It doesn’t look quite right, does it? Let’s turn OpenGL’s depth test on, to ensure that the occluded parts of the surfaces are not drawn:

```python
#:
w.Culling( True )
```

```python
@w.EventHandler( slot=-1 )
def Keys( self, event ):
    if event >> "kp[p]":  # toggle perspective/orthographic projection
        self.perspectiveCameraAngle = 0 if self.
perspectiveCameraAngle else 120
    if event >> "kp[g]":  # toggle pixel grid on/off (at z=0)
        grid.visible = not grid.visible
    if event >> "kp[z]":  # toggle depth oscillation on/off
        c.z = 0 if callable( c.z ) else Shady.Oscillator( 0.3 ) * 100
    if event >> "kp[l]":  # toggle lighting on/off
        if glIsEnabled( GL_LIGHTING ): glDisable( GL_LIGHTING )
        else: glEnable( GL_LIGHTING )
    if event >> "kp[c]":  # re-randomize colors (with shift to change mode)
        c.vertexColors.clear()
        if 'shift' in event.modifiers: c.boring = not c.boring
    if event >> "kp[d]":  # toggle depth test on/off
        if glIsEnabled( GL_DEPTH_TEST ): w.Culling( False )
        else: w.Culling( True )
    if event >> "kp[ ] kp[s]":  # pause/unpause spin
        if all( x==0 for x in c.angular_velocities ):
            c.angular_velocities = [ 23, 31, 37 ]
        else:
            c.angular_velocities = [ 0, 0, 0 ]
```

5.3. Getting Started
print(""")
P toggle perspective/orthographic projection
G toggle pixel-grid on/off at z=0
Z toggle z-oscillation on/off (note how the cube interacts with the pixel-grid Stimulus)
S pause/unpause the spin
L toggle lighting effect on/off
C re-randomize colors (press shift to switch between two different coloring strategies)
D toggle depth test on/off
"")
"#>
Shady.AutoFinish( w ) # tidy up, in case we're not running this with 'python -m Shady'

examples/interactive-gamma.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo interactive-gamma

#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.
#
# $END_SHADY_LICENSE$

#: Demonstrates an interactive perceptual-linearization tool
"""
This script demonstrates the `FindGamma()` utility function. It creates a `LinearizationTestPattern()` stimulus, displays it, (continues on next page)
and then installs an `EventHandler` that allows interactive gamma adjustment with the mouse or touchscreen.

This demo requires the third-party package `numpy`, used in the generation of the `LinearizationTestPattern()`.

```python
if __name__ == '__main__':
    import Shady
    
    Parse command-line options:
    
    cmdline = Shady.WorldConstructorCommandLine( canvas=True)
    gamma = cmdline.Option( 'gamma', 1.0, type=(int, float, tuple, list), doc=
        'Start value(s) for exponent `.gamma` that should be corrected for.')
    xBlue = cmdline.Option( 'xblue', True, type=bool, container=None, doc=
        'Whether or not to adjust blue gamma separately, relative to the others, with horizontal movement.')
    text = cmdline.Option( 'text', True, type=bool, container=None, doc='Whether or show the gamma values as text on screen.' }
    cmdline.Help().Finalize()
    Shady.Require( 'numpy' ) # die with an informative error if this is missing
    
    Create a `World`, along with a `Stimulus` that might benefit from linearization:
    
    world = Shady.World( **cmdline.opts )
    gabor = world.Sine( size = min( world.width / 2, world.height ) * 0.75,
        x = -world.width / 4,
        plateauProportion = 0,
        signalFrequency = 0.01, \ # 100 pixels per cycle
        cx = lambda t: t * 100, \ # 100 pixels per second
        contrast = Shady.Oscillator( 0.2 ) * 0.5 + 0.5,
    )
    # NB: The `w.Sine()` method is a wrapper around `w.Stimulus`.
    # Among other settings, it yokes the Stimulus `.gamma`
    # (and other "atmosphere" properties) to the corresponding
    # properties of the `World`
    redgreen = world.Sine().Inherit( gabor ).Set(
        x = world.width / 4,
        color = [1, 0, 0.15], \ # signal function output will be multiplied by this vector
    )
```

First we'll define a callback function to handle the gamma value that is found by the interactive adjustment procedure. The callback must handle the possibility that its input argument is `None` (which is what happens when the user presses the escape key).

```python
def finish( finalGamma ):
    if finalGamma is None:
```

(continues on next page)
print( '\nNo change: world.gamma is still %r' % list( world.
˓→gamma ) )

else:
    print( '\nSetting world.gamma = %r' % list( finalGamma ) )
    world.gamma = finalGamma

""
You can call `Shady.FindGamma()` quite straightforwardly yourself. But for the purposes of this demo, we'll add another layer of complexity: we'll set up an `EventHandler` such that `Shady.FindGamma()` gets launched every time you hit the enter/return key:

""

@world.EventHandler( slot=-1 )
def PressReturnToStartAdjusting( self, event ):
    if event.type == 'key_release' and event.key in [ 'enter', 'return' ]:
        Shady.FindGamma( world, xBlue=xBlue, finish=finish, text=text
˓→)

""

Press the enter/return key to display the linearization pattern and start adjusting gamma.

While adjusting:
- press enter/return again to finish adjusting and adopt the new settings;
- press escape to exit the adjustment without adopting the new settings;
- press any letter key to report the current gamma setting to the console.

""

Shady.AutoFinish( world )

events/noise.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo noise

#!/usr/bin/env python
#
#$BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
#$END_SHADY_LICENSE$
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
# $END_SHADY_LICENSE$

# Testing the properties of our GPU-side random number generator

""
This demo tests properties of the random-number generator (RNG) in our fragment shader. Since we want independent random values generated for each pixel, and the fragment shader processes pixels in parallel without being able to pass a state from one to the other, building a good RNG is a challenge.

This test generates a screen full of dynamic noise until the user presses a key. Then it takes a screenshot, closes down the World, and plots (assuming you have matplotlib installed) a histogram of the screenshot pixel values.

Results are easiest to interpret when the noise is generated without any gamma correction. The test can be parameterized using the --bg and --noise command-line parameters, which affect the `Stimulus.backgroundColor` and `Stimulus.noiseAmplitude` of the canvas, respectively.

If your graphics card, drivers and operating system make GLSL 3.3+ shader functions available (which is the case on most of the modern Windows systems we have tested) then our RNG is of pretty good quality. If these functions are not available, then the histograms are not quite as beautiful and there will sometimes be detectable structure in the noise--this is the case on macOS by default, because macOS does not allow legacy GLSL 1.2 code to be mixed with more modern OpenGL code. On macOS, it may therefore be worth explicitly disabling legacy OpenGL features with the constructor argument `--legacy=False` (compare the results of this demo with the `--legacy=False` command-line option vs `--legacy=True`).
""

import sys
import Shady

""
Juggle command-line construction options.
""

# die with an informative error if this is missing
Shady.Require( 'numpy' )

# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
# $END_SHADY_LICENSE$
threaded = False,
outOfRangeColor = -1,
outOfRangeAlpha = -1,
)

""
Create the World. Replace the default event handler. Also add a handler that runs just before the window closes: any key stops and closes the World, but a screenshot is taken first.
""

w = Shady.World( **cmdline.opts )

@w.EventHandler
def eh( self, event ):
    if event.type == 'key_press':
        self.Close()

@w.BeforeClose
def NoiseStats():
    w.snapshot = w.Capture()[ :, :, :3 ]
    normalized = w.snapshot / w.dacMax
    print( '
      Pixel stats: Red   Green   Blue' )
    print( '      min = [ %.3f, %.3f, %.3f ]' % tuple( normalized.min( axis=(0, 1) ).flat ) )
    print( '      max = [ %.3f, %.3f, %.3f ]' % tuple( normalized.max( axis=(0, 1) ).flat ) )
    print( '      std = [ %.3f, %.3f, %.3f ]' % tuple( normalized.std( axis=(0, 1) ).flat ) )
    print( '' )
    title = '--bg=%s --noise=%s' % ( repr( bg ).replace( ' ', '' ), repr( noise ).replace( ' ', '' ) )
    Shady.Histogram( w.snapshot, DACmax=w.dacMax, title=title )

""
Press any key to stop and analyze the noise. Remember you'll need to close the histogram window before you can exit this Python session.
""

Shady.AutoFinish( w, plot=True )

examples/precision.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo precision

#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
# $END_SHADY_LICENSE$
This demo tests the accuracy of the pipeline consisting of gamma-correction followed by "noisy-bit" dithering (Allard & Faubert 2008) as implemented in our fragment shader.

Render a stimulus in which desired luminance climbs from 0.0 on the left to 1.0 on the right, with a configurable number of steps. For each given target luminance level, the shader’s dithering algorithm renders either the nearest realizable intensity level above the target or the nearest one below---and this is done independently on each frame, in each color channel, and in each row of the image. This demo estimates accuracy by automatically taking an instantaneous screen capture, then using Python code to:

1. transform the resulting pixel values back through the screen nonlinearity, using the same gamma value that the shader had used to linearize the stimulus;

2. compute the quantization error between the desired target levels and the result, averaged across rows and color channels;

3. compute a lower bound on the precision implied by the observed quantization errors. This is the size in bits, of a hypothetical discrete intensity scale in which `max(abs(error))` corresponds to half a step size.

This demo requires third-party packages `numpy` and `matplotlib`. It creates a `matplotlib` figure showing the results.

See also the `dithering` demo.

```
if __name__ == '__main__':
    import Shady

    cmdline = Shady.WorldConstructorCommandLine( fullScreenMode=False,
                                           reportVersions=True )
    gammaValues = cmdline.Option( 'gamma', [ 1.0, 2.2 ], type=( int, float, tuple,
                                           list ), minlength=1, container=None, doc="A gamma value or a sequence of gamma values to use successively." )
    nRows = cmdline.Option( 'nRows', 900, type=( int, None ), min=-1,
                           container=None, doc="Number of image rows. Each row will have identical target values, but will be independently dithered. If `None`, fill the height of the image."")
```
nTargets = cmdline.Option( 'nTargets', None, type=( int, None ), min=-1,
container=None, doc="Number of target values. If `None`, use the highest power of 2
that will fit in the width of the screen." )
waitFrames = cmdline.Option( 'waitFrames', 5, type=( int ), min=1,
container=None, doc="Number of frames to wait before capturing (will affect the
random seed for the dithering noise)." )
cmdline.Help().Finalize()
numpy, matplotlib = Shady.Require( 'numpy', 'matplotlib' )
# die with an
informative error if either is missing
try: gammaValues[ 0 ]
except: gammaValues = [ gammaValues ]
***
Create a World:
###:
w = Shady.World( **cmdline.opts )
***
Create a sequence of target values and render it as a texture:
###:
if nRows in [ None, 0, -1 ]:
    nRows = w.height
if nRows > w.height:
    nRows = w.height
    print( '\nnRows has been reduced to %d to fit in the World\n' % w.height )
if nTargets in [ None, 0, -1 ]:
    nTargets = 2 ** int( numpy.log2( w.width ) )
print( '\nRendering %d target luminance values (%g-bit)' % ( nTargets, numpy.
log2( nTargets ) )
targetValues = numpy.linspace( 0, 1, nTargets, endpoint=True )
s = w.Stimulus( targetValues[ None, : ], height=nRows, atmosphere=w )
***
Create an AnimationCallback that performs adjustment of the stimulus `.gamma`
and `.ditheringDenominator`, and subsequent capture of the rendered pixel
values, on successive frames. This implementation takes advantage of Shady's
built-in support for Python's "generator functions" (i.e. functions that use
the 'yield' keyword) as callbacks. The 'yield' statements in the code delimit
successive frames.
###:
@w.AnimationCallback
def Sequence( self ):
    w.captured = {}
    for i in range( waitFrames ):
        yield # come back here on the next frame
        for gamma in gammaValues:
            for ditheringDenominator in [ -self.dacMax, self.dacMax ]:
                s.Set( gamma=gamma,
ditheringDenominator=ditheringDenominator )
                yield # allows one frame to be rendered, so that the
new settings
(continues on next page)
# can take effect before capturing

# Capture and normalize
rgb = s.Capture()[:, :, :3] / self.dacMax

# Undo the gamma-correction
separateChannels = rgb[:, :, 0], rgb[:, :, 1],
→ rgb[:, :, 2]

→ for x, channelGamma in zip(separateChannels, s.gamma)
→ channelGamma
→ x.flat = Shady.ScreenNonlinearity(x,)

→ # Transpose & reshape into a 2D array,
→ (nRows*nChannels)-by-nTargets
result = rgb.transpose(0, 2, 1).reshape([nRows *
→ 3, nTargets])

→ # Store for later analysis
w.captured[ (gamma, ditheringDenominator) ] = result

→ self.Close()

### Define a couple of functions for analyzing the values that will end
→ up in `w.captured`:

### def MeanErrors( output, targetValues ):

###     Take the mean-across-rows of the captured, processed pixel array,
→ and subtract the target values.
###     return output.mean( axis=0 ) - targetValues

### def EquivalentPrecision( quantizationErrors, method='max' ):

###     Compute the number of bits that would describe a uniformly-spaced
→ in which the maximum absolute value of `quantizationErrors` (or
→ possibly
→ the mean or median absolute value, depending on `method`) corresponds
→ to
→ half a step size.
###     return -numpy.log2(2 * getattr(numpy, method)(numpy.abs(
→ quantizationErrors))

### Now wait a second for the results to appear. You’ll have to close the plot
→ window manually in addition to exiting this prompt.

### def Plot():

import matplotlib.pyplot as plt

def LatexScientificNotation( x, fmt='%3.1e' ):

    import re
    if isinstance(x, (float, int)): x = fmt % x
    return re.sub(r'\([Ee][+-]0*(\[.0123456789\]+)$', r'\times{}', x)
axes = {}
nPlots = len( gammaValues )
layout = [ 1, nPlots ]
transpose = nPlots > 2
if transpose: layout = layout[ ::-1 ]
for ( gamma, ditheringDenominator ), output in sorted( w.captured.
˓→items() ):
    if gamma not in axes:
        axes[ gamma ] = plt.subplot( layout[ 0 ], layout[ 1 ],
˓→len( axes ) + 1 )
quantizationErrors = MeanErrors( output, targetValues )
grandMeanError = quantizationErrors.mean()
maxAbsError = numpy.abs( quantizationErrors ).max()
precisionLowerBound = EquivalentPrecision( quantizationErrors,
˓→'max' )
    print( 'gamma=%g, ditheringDenominator=%+g, grandMeanError=
˓→%+3.1e, maxAbsError=%3.1e, precisionLowerBound=%3.1e bits' % (gamma, ditheringDenominator, grandMeanError,
˓→maxAbsError, precisionLowerBound)
    )
    label = r'Dithering %s: $\bar{\Delta v}=%s$, $|\Delta v|_{\max}=%s$, bits$\geq %.2f$' % ('ON' if ditheringDenominator > 0 else 'off',
˓→LatexScientificNotation( grandMeanError, '+3.1e' ),
˓→LatexScientificNotation( maxAbsError, '+3.1e' ),
˓→precisionLowerBound,
    )
plt.plot( targetValues, quantizationErrors, label=label )
for iPlot, ( gamma, ax ) in enumerate( sorted( axes.items() ) ):
    if iPlot == 0 or transpose: ax.set_ylabel( 'Mean quantization error' )
    if iPlot == nPlots or not transpose: ax.set_xlabel( 'Target intensity level (normalized)' )
    ax.legend( loc='upper left' )
    yl = numpy.abs( ax.get_yticks() ).max()
    #yl += numpy.mean( numpy.diff( ax.get_yticks() ) )
    ax.set_ylim( [-yl, yl ] )
    ax.text( 0.02, 0.02, '$\gamma=%g$' % gamma, transform=ax.
˓→transAxes, size=20 )
plt.subplots_adjust( 0.08, 0.1, 0.98, 0.98 )
Shady.Utilities.FinishFigure( maximize=True )
Shady.AutoFinish( w, plot=Plot )

examples/sharing.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo sharing

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
```
This demo shows how individual properties can be shared between different Shady stimulus objects.

Often, you will want certain groups of `Shady.Stimulus` instances to share one or more properties. The advantages of this are twofold. Firstly, you save time, as you only need to adjust the properties of one linked stimulus in order to affect all the stimuli linked to it. Secondly, you save memory, as any shared properties among linked stimuli will use the exact same value (or array of values).

Once shared, properties can be "un-shared" at any time and can have their values adjusted individually.

```
if __name__ == '__main__':
    ""
    Let's start by creating a World, configured according to whatever command-line arguments you supplied. We'll use the canvas by default, with a gamma of 2.2, so we test out some of the useful sharing techniques later on.
    ""
    import Shady
    cmdline = Shady.WorldConstructorCommandLine( canvas=True )
    cmdline.Help().Finalize()
    world = Shady.World( gamma=2.2, **cmdline.opts )

    ""
    Let's create some rectangles:
    ""
    a = world.Stimulus( size=[400,200], color=[1.0, 0.0, 0.0], bgalpha=0, x=-300, rotation=45 )
    b = world.Stimulus( size=[200,100], color=[0.0, 0.0, 1.0], bgalpha=0, x=+300 )
    c = world.Stimulus( size=[100, 50], color=[0.0, 0.7, 0.0], bgalpha=0, y=+300 )
    d = world.Stimulus( size=[100, 50], color=[0.7, 0.7, 0.0], bgalpha=0, y=-300 )
```
Let's say `b` wants to be at the same angle as `a`. Well, on one level that's trivial:

```python
b.rotation = a.rotation
```

But when `a` moves on, `b` is left behind:

```python
a.rotation += 30
```

Let's use a slightly different syntax:

```python
b.rotation = a  # what??
```

That's right, it looked like we were trying to assign a Stimulus `instance` to a (usually numeric) property of another Stimulus---doesn't make much sense, right? But notice how the rectangles are aligned again? In fact, that was a syntactic shortcut for property sharing. The `.rotation` properties of `a` and `b` now share the same memory location. A change to `a` will affect `b`:

```python
a.rotation = 0  # affects BOTH `a` and `b`
```

...and (because they share the same memory) vice versa:

```python
b.rotation = 90  # ALSO affects both
```

The power of this idea becomes clear when you want properties to be dynamic. Let's attach a dynamic to the `.rotation` of `a`:

```python
a.rotation = lambda t: t * 45
```

Now, a single piece of code is running on every frame, and dumping its result into the memory location of `.a.rotation`. But since the memory location is now shared, that single piece of code can now simultaneously affect two stimuli (and could affect any number of stimuli) with no extra overhead cost.

How do we make `b` independent again? Using the same shorthand, we can tell `b` to "be itself" rather than trying to be somebody else:

```python
b.rotation = b  # The instantaneous value of `b.rotation` does not change, but it becomes unlinked from `a.rotation`.  # And since `a`, not `b`, is the stimulus that has # the dynamic `lambda` function running on it, `b`
# stops rotating.

```python
b.LinkPropertiesWithMaster( a, 'rotation' )  # same as b.rotation = a
# now they're spinning together again
```

and...

```python
b.MakePropertiesIndependent( 'rotation' )  # same as b.rotation = b
# now they're independent again
```

Note the plural "Properties" in those method names: one of the advantages of this more-verbose syntax is that you can link or unlink *multiple* properties in the same call. Optionally, you can also *set* their values in the same breath, using keyword arguments:

```python
b.LinkPropertiesWithMaster( a, 'color', 'envelopeSize', plateauProportion=0 )
# establish a link between `a` and `b` on all three properties:
# .color, .envelopeSize and .plateauProportion
# At the same time, set plateauProportion=0 which will then
# affect both stimuli.
```

The same goes for declaring independence:

```python
b.MakePropertiesIndependent( 'plateauProportion', 'envelopeSize', color=[0,0,1])
# uncouple .plateauProportion and .envelopeSize but don't change their values;
# uncouple .color and immediately change it back to blue
```

There's a third method, `.ShareProperties()` that works from the perspective of the "master" Stimulus. The main advantage to that is that you can propagate the sharing relationship to multiple *stimuli* in one call...

```python
a.ShareProperties( [ b, c, d ], 'rotation' )
```

...as well as multiple properties. And again, it allows you the option of setting one or more properties explicitly at the same time, using keyword arguments:

```python
a.ShareProperties( b, c, d, 'envelopeSize', plateauProportion=1 )
```

Note that only some attributes - specifically, fully-fledged `ManagedProperty` attributes, can be shared. The `.x` attribute alone, for example, cannot be shared, because it is a `ManagedShortcut` to only a single element of the `.envelopeTranslation` array, and you are limited to being able to share property arrays entirely or not at all. Try it yourself: I'm not going to do it, because that would crash the script,
but you can try either or both of the following by typing them at
the prompt:

```python
c.x = a  # wrong
a.ShareProperties( c, 'x' )  # equivalently wrong
```

```python
###

Also, you cannot share *unmanaged* properties like `.frame`
or `.page` or `.text`. Again, feel free to try it:

```python
c.frame = a  # wrong
a.ShareProperties( c, 'frame' )  # equivalently wrong
```

```python
###

To link a shortcut or unmanaged property, the best you
can do is create a dynamic:

```python
c.x = lambda t: a.x
```

# ... but of course that comes at an extra computational
# cost, on each frame (small, in this case, but in complex
# stimulus arrangements such costs can quickly add up)

```python
###

Anyway, now the two stimuli will automatically move together
in just the x dimension:

```python
a.xy = Shady.Transition(
    duration = 3,
    transform = lambda p: Shady.Sinusoid( 3.75 * p ) * 150 - 150,
)
```

```python
###

There is, however, one 'virtual' managed property that
exists principally to facilitate sharing. It is a shorthand
for a bundle of managed properties that affect linearization
and dynamic range enhancement.

Such things matter in visual psychophysics, so let's
illustrate with a psychophysics-y stimulus:

```python
gabor = world.Stimulus(
    signalFunction = Shady.SIGFUNC.SinewaveSignal,
    signalAmplitude = 0.5,
    plateauProportion = 0.0,
)
```

# or, you know, sigfunc=1, siga=0.5, pp=0
# (depends where you like your code to reside on
# the concise <-> readable spectrum)

```python
###

It's immediately, visibly obvious that we have failed to
match the stimulus `.gamma` to that of the surrounding
'World' and its canvas. `.gamma` is one of the bundle of
properties we're talking about, which we collectively
call the `.atmosphere`:
```python
  gabor.atmosphere = world
  # or equivalently:  gabor.LinkAtmosphereWithMaster( world )
```

Now the whole set of properties is matched, and will
track, the `World`. This becomes obvious if we take leave
of our senses and start changing them in real time:
```python
  world.ditheringDenominator = 3
  world.gamma = Shady.Oscillator( 1.0 ) * 0.5 + 2.2
```

Seriously though, stop that:
```python
  world.Set( ditheringDenominator=world.dacMax, gamma=2.2 )
```

Side note: in fact, the `World` itself has no direct
use for these properties: their visible effects are
actually mediated by a `Stimulus` called “canvas”. But
the `World` has them, as placeholders. When a canvas
is created (either by using the `canvas=True` constructor
argument when creating the `World()`, or by a later
explicit call to `world.MakeCanvas()`) these properties
get shared between the `World` and its canvas, using
exactly the kind of property-sharing mechanism we're
learning about today. So then any change to these
properties of the `World` will affect the canvas, and
vice versa.

You can learn more about the canvas and the "atmosphere"
properties by looking at the `PreciseControlOfLuminance`
topic documentation:
```python
  help( Shady.Documentation.PreciseControlOfLuminance )
```

The concept of a "master" is a fairly weak one, since any
change to the property values is symmetric between master
and followers. Being the "master" means two things:

1. when the sharing link is first made, the master’s
current property values are preserved and the followers’
values are overwritten (unless the value is overridden
explicitly by a keyword argument in the method call,
as we saw above).
```python
```
2. depending on the rendering back-end you’re using,
it may be meaningless for the "master" to declare
independence. But it is always meaningful for
the followers to declare independence.  
(This is a difference between the 'ShaDyLib' binary
accelerator and the pure-Python 'PyEngine'. We may
try to iron this out in future, one way or the other,
but it's a low priority.) Let's see what happens
there:

```python
a.MakePropertiesIndependent( 'rotation' )
# If you're using the accelerator (ShaDyLib) for rendering,
# nothing will happen - all colored blobs will keep rotating.
# But if you have disabled the accelerator (for example, by
# starting this script with --backend=pyglet --acceleration=False)
# then 'a' will break away and keep spinning while the others
# will stop.
```

Let's ensure things are back the way they were:

```python
a.ShareProperties( b, c, d, 'rotation' )  # all spinning again, if they weren't before
```

Another unrelated potential gotcha is that you need
to remember how dynamic properties work: they install
a small subroutine that is associated with the rendering
of a «particular» Stimulus on each frame, and which dumps
its results into the memory space of that Stimulus.
This computation survives turning invisible:

```python
a.visible = 0
```

...but not if that Stimulus leaves the stage entirely:

```python
a.Leave()  # all the others stop, because the dynamic
# was specific to the 'a' stimulus
```

They're still linked:

```python
b.rotation += 90  # all change
```

...in both directions:

```python
a.rotation += 90  # all change again
```

...but a dynamic...

```python
a.rotation = lambda t: t * 45
```

...only runs when 'a' is on-stage:

```python
a.Enter( visible=1 )
```

Good luck sharing properties!

```python
Shady.AutoFinish( world )
```
examples/showcase.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo showcase

```python
#: ==> Start here to see a little bit of everything Shady can do <==
#
#: This script gives a taste of most of Shady's main features.
#
#: Unlike most other example scripts, and unlike Shady's default settings, it will open a small framed window that you can drag around. Since this is (hopefully) your first Shady demo, we're going to make the code as simple as possible. That means we won't give you the chance to override window parameters from the command-line, which is something all the other demos allow. Note that rendering performance is nearly always better in a foregrounded frameless window that covers the whole screen (Shady's usual default).

if __name__ == '__main__':
    import sys
    import Shady
    if sys.argv[1:]:
        # Let's allow command-line arguments if the user wants
        # to use them (e.g. for specifying a --backend) but
        # for now let's hide the code for doing it. The `World()`
        # constructor call should look easy to understand when
        # the user first encounters it.
        cmdline = Shady.WorldConstructorCommandLine( width=1200, height=700, 
                                                   top=50, fullScreenMode=False, frame=True, canvas=True )
        cmdline.Help().Finalize()
        world = Shady.World( gamma=2.2, **cmdline.opts )
    else:
        # simpler easier-to-understand case
```

(continues on next page)
Let's set up the "World" in which our Stimuli will reside. Since we don't know you well enough yet to know how you're arranging your windows and screens, let's take the unusual step of making the World a framed, draggable window.

```python
world = Shady.World( width=1200, height=700, top=50, frame=True )
world.MakeCanvas( gamma=2.2 )  # you can also have this step done automatically by including
# in the World constructor line (along
# with any other physical properties
# like gamma)
```

In future demos, we'll stick closer to Shady's defaults and you can manage windowing yourself. If you have a second monitor, maybe you would want to supply the command-line option `--screen=2` to put the 'World' on a second screen. Or, if you have only one screen, maybe you could configure your console window to be semi-transparent. Or just alt-tab back and forth between console and 'World'. You'll figure it out.

```python
gauge = Shady.FrameIntervalGauge( world, corner=[ -1, -1 ] )
```

Throughout this demo, we'll keep an eye on timing. One simple way to do this is to put a 'FrameIntervalGauge' on the screen. It's just a visible scale that measures frame-to-frame intervals in milliseconds (so, for 60 fps it should hover between 16 and 17). Note that performance is generally much better when the window is in the foreground than in the background. Also, it is usually better when the window is frameless and fills the screen, which is the way Shady would normally be used.

```python
gabor = world.Stimulus( size = 300,  # size in pixels (if you want a rectangular here)
                        position = [ -200, +200 ],  # in pixels relative to the World's origin anchor
                        signalFunction = Shady.SIGFUNC.SinewaveSignal,  # there's only one
                        # signal function, but
                        # you can define more. The
                        # value of this property
```
numeric case, 1).

signalAmplitude = 0.5, # the .backgroundColor is 0.5 by default.
so that, we It's to ensure manipulate the from there.

plateauProportion = 0, # non-negative plateauProportion causes a spatial windowing function to be applied that elsewhere. It and those

atmosphere = world, # this uses a couple of shorthand tricks.

Let's animate it too:

```python
gabor.cx = lambda t: t * 100 # .cx is a shortcut for the horizontal component of gabor.carrierTranslation
gabor.normalizedContrast = Shady.Oscillator( 0.2 ) * 0.5 + 0.5 # the Oscillator is an example of a Function, which is a callable
can do arithmetic with

Square or rectangular patches of solid color are easy:

rectangle = world.Stimulus(
    size = [ 300, 200 ],
    position = [ +200, +200 ],
    color = [ 0, 0.7, 0 ],  # foreground color green
)

To make a patch circular or elliptical, we specify the .plateauProportion and .backgroundAlpha:

ellipse = world.Stimulus(
size = [300, 200],
position = [200, -200],
color = [0, 0, 0],
plateauProportion = 1,  # it's plateau all the way to the edge, so...
→the edge is sharp (but curved)
→idea for linearized stimuli)
→backgroundAlpha = 0,  # transparent background (NB: not a good
→idea for linearized stimuli)

# You can also use the method world.Patch() which
# is simply a convenience wrapper around world.Stimulus()
# that sets color=1 and backgroundAlpha=0 by default.

***
So far our stimuli haven't needed third-party packages. But from now on, we will need the `numpy` package, to manipulate arrays of numbers.
***

Shady.Require( 'numpy' )  # die with an informative error if this is missing

***
Let's add a field of dots, demonstrating how a single Stimulus may contain multiple independently-moving shapes. You have to change the .drawMode and then set the .points property. The latter can be an n-by-2 array of coordinates, or a sequence of complex numbers. A NaN in the sequence of points indicates a break in drawing (a place where the pen is taken off the paper, so to speak):
***

field = world.Stimulus(
    size = [300, 200],
    position = [200, -200],
    color = [0, 0, 0],
    drawMode = Shady.DRAWMODE.POLYGON,
    smoothing = 0,
)

nDots = 50
nSides = 20
polygonRadius = 3

import numpy
positions = numpy.random.uniform([0, 0], field.size, size=[nDots, 2])
velocities = numpy.random.uniform([-100, -100], [+100, +100], size=[nDots, 2])
shape = Shady.ComplexPolygonBase(nSides) * polygonRadius  # this handily
→includes the NaN break between polygons

field.points = lambda t: Shady.Real2DToComplex((positions + velocities * t) % field.size) + shape

# Real2DToComplex converts nDots-by-2 real coordinates into nDots-by-1
# complex, and `shape` is 1-by-(nSides+1) complex. numpy "broadcasting"
# in the `+` operator automatically does the magic of creating multiple
# polygons at different locations.

***
The dots initially have a radius of 3 pixels. Let's make them bigger:
```python
shape *= 2
# this is possible because we're working in the same namespace in which
# we also created the function to specify the `.points` property, and
# that function contains a baked-in "global" reference to `shape` in that
# namespace. Therefore, changes to the `shape` variable here will affect
# the function. This behavior is a subtle (and sometimes counterintuitive)
# feature of Python, but very powerful and well worth taking time to learn.
```

From now on, we'll also need the `Image` module from the `pillow` or `PIL`
package, since it is used under the hood whenever we want to load from, or
save to, an image file.
```
Shady.Require( 'Image' )  # die with an informative error if this is missing
```

We'll create a Stimulus from an image file from disk. Our default
element is a multi-frame GIF, so we'll demonstrate texture animation
into the bargain:
```
filename = Shady.EXAMPLE_MEDIA.alien1
image = world.Stimulus( filename, position=[ 0, 0 ], frame=lambda t: t * 16 )
```

Let's go for a walk. The `Shady.Dynamics` sub-module contains
a number of objects that can be used for dynamic property setting.
The general-purpose `Function` class provides a callable object that
can be modified by ordinary arithmetic as well as other transformations.
`Integral` and `Derivative` are wrappers around `Function`
construction: they provide stateful memory and can hence be used to
make discrete numeric approximations to the integrals and derivatives
of arbitrary functions. `Oscillator` is a wrapper around
`Integral`: the `Function` object is transformed so that it produces
sinusoidal oscillations.

Here we demonstrate how you can include `Function` objects in
arithmetic expressions: the output of `Oscillator` is multiplied by
a scalar:
```
radius = min( world.size ) / 2.0
period = 6.0
image.Set(
  position = Shady.Oscillator( 1. / period, phase_deg=[ 0, -90 ] ) * radius,
  rotation = Shady.Integral( 360.0 / period ),
  anchor = ( 0, -1 ),
  z = -0.1
)
```

The `Shady.Dynamics` sub-module also provides a handy `StateMachine`
for changes that follow piecewise logic in time.

The `StateMachine` object is also callable. It takes a single
argument, time `t`, and this is one of the acceptable prototypes
for an `AnimationCallback`---in fact, it's the one that allows direct
assignment to the `.Animate` attribute.

```python
#:
class Noisy( Shady.StateMachine.State ):
    duration = 0.5
    next = 'Solid'
    def onset( self ):
        rectangle.Set( color=-1, noiseAmplitude=-0.5 )

class Solid( Shady.StateMachine.State ):
    duration = 4
    next = 'Noisy'
    def onset( self ):
        rectangle.Set( color=numpy.random.uniform( size=3 ),
                      noiseAmplitude=0 )

rectangle.Animate = Shady.StateMachine( Noisy, Solid )
```

Now let's try text. Text rendering is an optional extra that needs `numpy` and `pillow`. Also, if `matplotlib` is installed, that will get imported too, to enable access to your system fonts. Font setup may take several seconds if you're doing this for the first time, which is one of the reasons that text rendering is not enabled by default. To enable it, we simply say:

```python
#:
import Shady.Text
```

All set? Creating a text stimulus is easy enough:

```python
#:
message = world.Stimulus(
    position = ellipse.Place( 0, 0 ),
    text = 'Hello world!

This is Shady.',
    text_align = 'center',  # this is a way of addressing the text sub-
    # during initialization, before the `.text` instance has
    # been created...

message.text.align = 'center'  # but this is the canonical way of
                               # manipulating sub-
                               # properties
                               # properties of the `.text` object once it is
                               # created.

#:
message.text = lambda t: 'It has been

%d seconds

since the World was

created.' % t
```

Did that have an impact on the frame interval gauge? And on the smoothness of the rest of the animations? Be warned: whenever the text changes, as it now does once per second, the image has
to be re-rendered on the CPU and then transferred to the GPU. This is an inefficient approach by Shady's usual standards, so it may affect timing performance.

```python
message.scaling = Shady.Transition( start=1.0, end=0.5, duration=2.0 )
# or, rather, two seconds.
```

The text doesn't quite fit into the ellipse. So, hold on a second:

```python
message.scaling = Shady.Transition( start=1.0, end=0.5, duration=2.0 )
# or, rather, two seconds.
```

The final Stimulus trick we'll consider here is video. We'll need the 'cv2' module, from 'opencv-python'. This is slightly less stable across platforms and versions than Shady's other dependencies, so your mileage may vary. Therefore, like text rendering, we have made video decoding an optional extra. So it needs to be enabled explicitly, as follows:

```python
import Shady.Video
```

Note that video decoding will be done on the CPU, and each new frame is transferred from the CPU to the GPU---even more so than dynamic text, this is a very inefficient approach by Shady's usual standards. As a result, frame timing performance will often suffer, especially if the video is large.

Whenever precise frame timing matters, examine your timing performance critically, using the `FrameIntervalGauge` and/or `Shady.PlotTimings`.

```python
import os
caption = world.Stimulus(
    text = os.path.basename( filename ),
    position = lambda t: fish.Place( 0, -1.1 ),
    anchor = [ 0, +1 ],
    text_size = 20,
    text_bg = [ 0, 0, 0, 0.5 ],
)
```

...and play it:
Well, it's been fun. For a list of other interactive demos, you can launch Python as follows::

```python
python -m Shady list
```

Meanwhile, here is some version information:

```python
"""#
\n\n'\n\nworld.ReportVersions()
print( '\nPress Q or escape to close the window.' )
"""

Shady.AutoFinish( world )

examples/tearing.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo tearing

```python
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/ .
#
# $END_SHADY_LICENSE$

#: A quick visual test for "tearing" artifacts
"""
Tearing artifacts can arise when the windowing back-end fails to synchronize with the display hardware (possibly because vertical synchronization needs to be explicitly enabled in the control panel of the driver for the graphics card).

This demo shows a `TearingTest()` stimulus which makes tearing

(continues on next page)
artifacts visible. Watch out for this kind of effect::

```
**************
**************
**************
**************
**************
**************
**************
```
as well as fast-moving artifacts that make the edges look "ragged":

```
**************
/**************/
/**************/
/**************/
/**************/
```

Blurry edges are OK; torn or ragged edges are not. Bear in mind, performance will probably be bad while the 'World' is running in a background window. Switch the window to the foreground before you judge.

```python
if __name__ == '__main__':
    import Shady
    cmdline = Shady.WorldConstructorCommandLine()
    ruler = cmdline.Option( 'ruler', False, type=bool, container=None, doc='Whether or not to display a `PixelRuler` to verify screen resolution.' )
    gauge = cmdline.Option( 'gauge', False, type=bool, container=None, doc='Whether or not to display a `FrameIntervalGauge` to help debug timing performance.' )
    period = cmdline.Option( 'period', 4.0, type=(int, float), container=None, doc='Period, in seconds, of the oscillatory motion of the bar stimulus.' )
    swap = cmdline.Option( 'swap', 1, type=int, container=None, doc='Number of physical frames per `World` update (NB: some back-ends and some operating systems may not support anything other than 1).' )
    cmdline.Help().Finalize()

    world = Shady.World( **cmdline.opts )
    if gauge: gauge = Shady.FrameIntervalGauge( world )
```
to allow this to be changed on-the-fly by pressing number keys.

```python
@world.EventHandler( slot=-1 )
def ChangeSwapInterval( self, event ):
    if event.type == 'key_press':
        Swap( self, event.key )

def Swap( world, value ):
    try:
        value = int( value )
        except:
            return
        if value < 0:
            return
        print( '% 7.3f : Attempting world.SetSwapInterval( %r )' % ( world.t, value ) )
        world.SetSwapInterval( value )
    if swap != 1:
        Swap( world, swap )
```

Create the moving bar. Watch its edges.

```python
bar = Shady.TearingTest( world, period_sec=period )#.Set( alpha=0.75 )
```

If ruler:

```python
if ruler:
    Fill in the background with a 'PixelRuler' pattern
    # die with an informative error if this is
    Shady.Require( 'numpy' ) # die with an informative error if this is
    ruler = Shady.PixelRuler( world )
```

```python
print( 'Press Q or escape to close the window' )
Shady.AutoFinish( world )
```

examples/text.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo text

```bash
#!/usr/bin/env python
# $BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
# This program is distributed in the hope that it will be useful,
# (continues on next page)
```
How to manipulate text stimuli

This demo provides a keyboard-interactive exploration of Shady's text-rendering capabilities.

It requires third-party packages: `numpy`, `pillow` and will fail without them. Also, `matplotlib` is needed to access the system's fonts: without it, you will only be able to render in the "monaco" font.

NB: changing the string content, or font style, of a text Stimulus causes the image to be re-rendered on the CPU and then transferred from CPU to GPU. This is less efficient than Shady normally aims to be, and can impact performance, so it's best to ensure it does not happen frequently in timing-critical applications (see the `showcase` demo for more details).

```python
if __name__ == '__main__':
    # Parse command-line options:
    import Shady
    cmdline = Shady.WorldConstructorCommandLine()
    cmdline.Help().Finalize()

    # Enable text rendering by explicitly importing `Shady.Text`, then create a `World`:
    import Shady.Text  # necessary to enable text/font-handling functionality
    w = Shady.World( **cmdline.opts )

    # Now we'll set up a list of test texts, a list of fonts, and a general-purpose function for cycling through them.
    TEXTS = [Shady.Text.TEST, Shady.Text.TEST_UNICODE, Shady.Text.TEST_WRAPPING, Shady.Text.TEST_PANGRAM]
    FONTS = list(Shady.Text.FONTS)
    WRAPPING_MODES = [-800, 800, None]
    def Cycle(lst, backwards=False):
        if backwards: lst.insert(0, lst.pop(-1))
        else: lst.append(lst.pop(0))
        return lst[-1]
    firstSampleText = Cycle(TEXTS)
```
Creating a text stimulus is as easy as:
```python
sample = w.Stimulus( text=firstSampleText )
```

We assigned a string to the `.text` property. In fact, that's a shorthand: what it implicitly does is ensure creation of an appropriate text-handling object in `sample.text`, and then set *its* property `sample.text.string` equal to the desired string.

The `.text` property of a `Stimulus` supports dynamic assignment. Let's demonstrate by creating an informative caption below our text sample:
```python
def ReportFont( t ):
    if not sample.text:
        return ''
    return '{font}({style}){warning}'.format(
        font = sample.text.font,
        style = sample.text.style,
        warning = '' if sample.text.font_found else '

not available',
    )

caption = w.Stimulus(  
    text = ReportFont,  # dynamic-enabled shortcut to .text.string  
    xy = lambda t: sample.Place( 0, -1 ) - [ 0, 30 ],  # wherever the
→sample goes or however it grows,  
→anchor = ( 0, 1 ),  # let the top edge
→of the caption remain 30  # pixels below the
→sample's bottom edge
)

caption.text.align = 'center'
caption.text.blockbg = ( 0, 0, 0, 0.5 )
caption.text.border = 2
```

Let's set up a pixel ruler for judging the size of the finished sample:
```python
pixelruler = Shady.PixelRuler( 1000, world=w ).Set(  
    alpha = 0.7,
    carrierTranslation = lambda t: sample.Place( Shady.LOWER_LEFT,  
→worldCoordinates=False ),
)
```

...but for now, let's hide it:
```python
pixelruler.visible = False
```

Let's set up an event-handler to allow exploration of various text properties using keystroke commands:
@w.EventHandler

def KeyboardControl( world, event ):
    if event.type in [ 'key_press', 'key_auto' ]:
        # When you get an event.type in [ 'key_press', 'key_release',
        # 'key_auto' ]
        # the key information is to be found in event.key:
        # - event.key allows easy case-insensitive matching (it's always lower case);
        # - non-printing keystrokes (e.g. 'escape', 'up', 'down') are reported;
        # - have to be careful what you assume about international keyboard layouts
        # e.g. the condition 'event.key == '8' and 'shift' in event.modifiers'
        # guarantees the '*' symbol on English layouts but not on many others;
        # - when the key is held down, you get one 'key_press' followed by multiple
        #   repeating 'key_auto' events.
        command = event.key
        if command in [ 'q', 'escape' ]: world.Close()
        elif command in [ 'up' ]: sample.text.linespacing *= 1.1
        elif command in [ 'down' ]: sample.text.linespacing /= 1.1

        if event.type == 'text':
            # Detect a event.type == 'text' and examine event.text:
            # - case sensitive;
            # - non-printing keystrokes cannot be detected;
            # - independent of keyboard layout (you get whatever symbol the user intended to type);
            # - 'text' events are re-issued on auto-repeat when the key is held down.
            command = event.text.lower()
            if command in [ 'c' ]: sample.text.align = 'center'
            elif command in [ 'l' ]: sample.text.align = 'left'
            elif command in [ 'r' ]: sample.text.align = 'right'
            elif command in [ 'm' ]: sample.text.font = 'monaco'
            elif command in [ 'd' ]: sample.text.font = [ 'arial unicode',
            # 'devanagari', 'nirmala' ] # whichever matches first
            elif command in [ 'f' ] and FONTS: sample.text.font = Cycle(FONTS,
            'shift' in event.modifiers )
            elif command in [ 'b' ]: sample.text.bold = not sample.text.
            elif command in [ 'i' ]: sample.text.italic = not sample.text.

            # is a shorthand - could also say sample.text.string = Cycle( TEXTS ) # this
            else: sample.text.size /= 1.1
        # size is an alias for .lineHeightInPixels, so these lines will fail if
        # size has most recently been controlled via .emWidthInPixels instead
        elif command in [ 'g' ]: sample.text.blockbg = None if sample.
        # text.blockbg else ( 0, 0.7, 0 )
        else: sample.text.bg = None if sample.text.
        # bg else ( 0.7, 0.7, 0 )
        elif command in [ 'w' ]: sample.text.wrapping = Cycle(WRAPPING_MODES)
elif command in [ 'p' ]: pixelruler.visible = not pixelruler.

elif command in [ ',', ']' ]:
    direction = -1 if command in [ ',' ] else +1
    value = sample.text.border  # could be a scalar

(proportion of line height) or tuple of absolute pixel widths (horizontal, vertical)

    try: len(value)
    except: sample.text.border = max(0, value +

    direction * 0.1)

else: sample.text.border = [max(0, pixels +

    direction * 10) for pixels in value]

""
That was quite a lot to take in. So, let's render a legend that summarizes the possible keystrokes:
""
instructions = ""
L / C / R left / center / right alignment
F / shift+F cycle through system fonts
B / I toggle .text.bold / .text.italic where possible
M set .text.font = 'monaco' (our default font)
T cycle between demo texts
D try to find a Devanagari font, for the Sanskrit
- / + increase / decrease .text.size
[ / ] increase / decrease .text.border
up / down increase / decrease .text.linespacing
Y toggle yellow .text.bg
G toggle green .text.blockbg
W cycle .text.wrapping modes
P toggle visibility of the pixel ruler
Q / escape close window
""
legend = w.Stimulus(
    text = instructions.strip(' 
    z = +0.5,
    anchor = ( -1, -1 ),
    # Place the lower-left corner of
    xy = w.Place( -1, -1 ),
    # ...in the lower-left corner of the
    text_size = 20,
    #) These are shortcuts to the sub-
    text_border = 2,
    #) `legend.text`, addressed *after*
    text_blockbg = ( 0, 0, 0, 0.5 ), #) is created
)
""#>
print( instructions )
Shady.AutoFinish( w ) # in case we didn't get here from `python -m Shady ...`

examples/video.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo video
#: Decoding and rendering of frames from a movie file or camera

This demo script shows some of the things that can be done with video stimuli.

By explicitly importing `Shady.Video`, you can enable the `.video` property of the `Stimulus` class. You can assign the filename of a video file, or the integer ID of a live camera, to this property (it is actually redirected to `.video.source`, with `.video` itself being an object that is implicitly constructed when needed).

Video file decoding and live camera acquisition both require the `cv2` module from the third-party package `opencv-python`, which can be installed via `pip`.

Be warned that video stimuli are fundamentally less efficient than Shady's other more typical stimulus approaches: rather than transferring everything to the graphics card up-front, or allowing everything to be generated on-the-fly on the GPU, this uses CPU code to decode every new frame from the video source and then to transfer it from CPU to GPU. Timing performance will likely suffer as a result.

```python
import random

if __name__ == '__main__':
    import random
    import Shady

    Parse command-line options:
    >>>
    cmdline = Shady.WorldConstructorCommandLine( width=1000, height=750 )
```
source = cmdline.Option( 'source', Shady.PackagePath( 'examples/media/fish.mp4' ), type=( int, str ), container=None, doc="Supply the path-/file-name of a movie file, or an integer (starting at 0) to address a camera." )

loop = cmdline.Option( 'loop', True, type=bool, container=None, doc="Specifies whether or not to play the video on infinite loop." )

transform = cmdline.Option( 'transform', False, type=bool, container=None, doc="Demonstrate ways to use, and not to use, the .video.Transform callback" )
gauge = cmdline.Option( 'gauge', transform, type=bool, container=None, doc="Whether or not to show a `FrameIntervalGauge`." )

multi = cmdline.Option( 'multi', 0, type=int, container=None, doc="Specifies the number of animated copies of the video to make. Copying and animation is efficient (happens all on GPU)." )

cmdline.Help().Finalize()

```python
import Shady.Video
w = Shady.World( **cmdline.opts )
if gauge: Shady.FrameIntervalGauge( w )

s = w.Stimulus( video=source, bgalpha=0 )
s.video.Play( loop=loop )

if transform:
    sm = Shady.StateMachine()

    Some of the steps will use this helper function as the video transform:
    
def Desaturated( x ):
        gray = x[ :, :, :3 ].mean( axis=2 )
x[ :, :, 0 ] = gray
x[ :, :, 1 ] = gray
x[ :, :, 2 ] = gray
    return x

    In the first step, the video is not transformed. Shady only transfers new data from CPU to GPU when a new image is supplied by the camera or decoded from the file. Note however, that even this will affect frame timing: Shady operates most smoothly when all possible image frames have been pre-loaded onto the GPU.
```
class DoNothing( Shady.StateMachine.State ):
    duration = 3
    next = 'TransformNewData'
    def onset( self ):
        s.video.Transform = None

    ""
    The next 3-second state demonstrates an efficient video-frame 
    transformation: every time a *new* frame is decoded, transform 
    it (in this case, desaturate it). Not every display frame 
    will entail a new frame of video content however, so if the 
    video hasn't changed, return 'None' to signal that nothing 
    needs to be done.
    """

class TransformNewData( Shady.StateMachine.State ):
    duration = 3
    next = 'ReturnOriginalEveryTime'
    def onset( self ):
        s.video.Transform = lambda x, changed: Desaturated( x )

    if changed else None

    ""
    Add a 3-second state demonstrating an INEFFICIENT abuse of 
    the transformation mechanism. The transformation is just the 
    identity transform, so no change will be visible, but a 
    "transformed" image is returned on every display frame. This 
    means Shady will think it has to transfer a new texture to 
    the GPU on every display frame, which ordinarily it would not 
    do (normally it only needs to do this at the video frame rate, 
    not the display frame rate).
    """

class ReturnOriginalEveryTime( Shady.StateMachine.State ):
    duration = 3
    next = 'TransformEveryTime'
    def onset( self ):
        s.video.Transform = lambda x, changed: x

    ""
    Another example of what NOT to do: here we have the 
    desaturating transformation implemented INEFFICIENTLY because 
    it returns something on *every* display frame, regardless of 
    whether or not there is new video content.
    """

class TransformEveryTime( Shady.StateMachine.State ):
    duration = 3
    next = 'DoNothing'
    def onset( self ):
        s.video.Transform = lambda x, changed: Desaturated( x )

    ""
    Ensure `sm( t )` is called on every frame:
    """
s.SetAnimationCallback( sm )

```python

def Spawn( multi ):
    s.plateauProportion = 1 # oval/circular
    s.video.aperture = min( s.video.aperture ) # definitely circular

t0 = w.timeInSeconds # for synchronization (see below)
for i in range( multi ):
    cyclical_offset = i / float( multi )
    cycle = Shady.Integral( 0.2 ) + cyclical_offset
    position = Shady.Apply( s.Place, cycle * 360, polar=True )
    anchor = Shady.Apply( Shady.Sinusoid, cycle, phase_deg=[270, 180] )

    # Each newly-created Integral starts its clock the first time it is called.
    # We'll call them once below, explicitly, with t0, before using them
    # as dynamic property values. This ensures that, even if each child stimulus
    # takes time to create, the dynamic properties are in sync across copies.
    # (This would be an issue in a threaded environment if you were to call
    # `Spawn(multi)` directly rather than `w.Defer( Spawn, multi )` because in
    # that case each `w.Stimulus()` call below would be deferred to the end of
    # the current frame, with the engine waiting synchronously for each one
    # to complete - so, it would take one frame per child).
    # create a new Stimulus that shares the old one's texture:
    child = w.Stimulus( s, position=position( t0 ),
                        anchor=anchor( t0 ) )
    # copy all the physical properties of the parent Stimulus:
    child.Inherit( s )
    # actually share some of the properties (texture and texture dimensions):
    s.ShareTexture( 'envelopeSize', 'textureSize', child )
    # individually scale and animate the copy:
    child.Set( position=position, anchor=anchor, scaling=0.2 )
    child.color = [ random.random() for channel in 'rgb' ]

    if multi:
        w.Defer( Spawn, multi ) # use of .Defer() means that the explicit t0 synchronization, above, isn't actually required
```

(continues on next page)
def VideoKeyControl(self, event):
    if event.abbrev in 'kp[ ]':
        s.video.playing = not s.video.playing
    if event.abbrev in 'kp[left] ka[left]' and not s.video.live:
        s.video.Pause()  # another syntax for setting s.video.playing
        s.video.frame = False
    if 'shift' in event.modifiers:
        if not s.video.live:
            s.video.frame = 0  # rewind to
        else:
            s.video.frame -= 1  # step back
    if event.abbrev in 'kp[right] ka[right]' and not s.video.live:
        s.video.Pause()  # another syntax for setting s.video.playing
        s.video.frame = -1  # skip to
    else:
        s.video.frame += 1  # step forward

    print("space pause/unpause
left-/right-arrow step back/forward one frame
shift + left-/right-arrow rewind to first frame/skip ahead to last frame")

examples/world.py

This is one of the example scripts included with Shady. These scripts can be run conventionally like any normal Python script, or you can choose to run them as interactive tutorials, for example with python -m Shady demo world

#!/usr/bin/env python
#$BEGIN_SHADY_LICENSE$
#
# This file is part of the Shady project, a Python framework for
# real-time manipulation of psychophysical stimuli for vision science.
#
# Copyright (c) 2017-2019 Jeremy Hill, Scott Mooney
#
# Shady is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
#$END_SHADY_LICENSE$

#: Creates a 'World', starts a shell and leaves the rest to you.
"""
This script presents an interactive tabula rasa: it creates a `World` instance called `w`, configured according to the command-line options you supply, starts an interactive shell, and then leaves you to it.

```python
if __name__ == '__main__':
    import Shady

    ***
    Parse command-line options that affect 'World' construction:
    ***
    cmdline = Shady.WorldConstructorCommandLine( fullScreenMode=False,
        reportVersions=True )
    gamma = cmdline.Option( 'gamma', 1.0, type=( int, float, str ), strings=['sRGB'] )
    bg = cmdline.Option( 'bg', 0.5, type=( int, float ) )
    gauge = cmdline.Option( 'gauge', False, type=bool, container=None )
    grid = cmdline.Option( 'grid', False, type=( bool, str ), strings=['centered'], container=None )
    cmdline.Help().Finalize()

    ***
    Create a 'World' and report some version information:
    ***
    w = Shady.World( **cmdline.opts )

    ***
    Optional extras:
    ***
    if gauge:
        f = Shady.FrameIntervalGauge( w, corner=Shady.LOWER_LEFT )
    if grid:
        numpy = Shady.DependencyManagement.Import( 'numpy' )
        if numpy:
            p = Shady.PixelRuler( w )
            if grid == 'centered': p.carrierTranslation = w.size / 2
        else:
            print( '
    could not create a PixelRuler (%s)
    ' % numpy )
    Okay, over to you. Bye.
    ```
```

5.4 Key Concepts

5.4.1 Shader Pipeline Details

The heart of Shady is a customizable, specialized “fragment shader” program written in GLSL. It is designed to ensure that the GPU takes on nearly all of the burden of frame-by-frame pixel processing, including signal generation, animation, spatial windowing, contrast modulation in time and space, gamma correction, quantization, and dynamic-range enhancement tricks.

Specifically, the steps in the shader pipeline are as follows:

1. Compute a carrier pattern. Depending on the settings of the Stimulus, this may be (a) a patch of solid color,
(b) a pre-determined texture (i.e. an array of discrete pixel values), (c) a signal function that generates patterns procedurally at run-time, or (d) a combination of these options.

The rules for combining different carrier types are as follows:

- Texture and .signalFunction are additive.
- If you omit the texture, solid .backgroundColor is used as the signal baseline instead.
- If you omit the .signalFunction, it looks the same as a signal function that outputs 0 everywhere.
- .color, if supplied, is a multiplier for both texture and .signalFunction (though often with qualitatively different effects, because texture values are always in the range 0 to 1 whereas signal functions can go negative).
- An exception to all the above rules is if you omit both texture and .signalFunction, but specify a .color; then it is assumed you want a solid patch of the specified .color, independent of the .backgroundColor.

The table below summarizes the possible outcomes:

<table>
<thead>
<tr>
<th>Texture, .color:</th>
<th>.signalFunction</th>
<th>no .signalFunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture, .color:</td>
<td>texture *.color + signal *.color</td>
<td>texture *.color</td>
</tr>
<tr>
<td>Texture, no .color:</td>
<td>texture + signal</td>
<td>texture</td>
</tr>
<tr>
<td>No texture, .color:</td>
<td>.backgroundColor + signal *.color</td>
<td>.color</td>
</tr>
<tr>
<td>No texture, no .color:</td>
<td>.backgroundColor + signal</td>
<td>.backgroundColor</td>
</tr>
</tbody>
</table>

2. Translate, rotate or scale the carrier pattern if requested. Carriers are treated as infinitely, cyclically repeating patterns for this purpose. In the case of procedurally-generated signals, the transformations are actually applied to the coordinate system before the .signalFunction is evaluated.

3. Apply contrast effects. This may include (a) a .windowingFunction that attenuates the edges of the stimulus, (b) another more arbitrary procedural .modulationFunction (for example, sinusoidal contrast modulation), (c) an overall .contrast scaling factor, or (d) a multiplicative combination of these options.

For linearized, psychophysically-accurate stimuli, the backgroundAlpha property should be 1.0, in which case contrast effects cause the carrier to be blended with the specified backgroundColor. On the other hand, if the Stimulus has backgroundAlpha < 1.0, contrast effects are mediated through alpha blending with other stimuli (and you should not expect the composite result to be accurately linearized).

4. Translate, rotate or scale the complete stimulus if requested. (Note that scaling, and any rotation except a multiple of 90 degrees, will compromise the pixel-perfect accuracy of the stimulus content due to interpolation artifacts. As for translations: under normal circumstances Shady automatically rounds to the nearest pixel, to avoid such interpolation artifacts in ordinary stimulus repositioning.)

5. Add noise, if requested. A two-dimensional uniform or Gaussian pixel noise pattern can be added to the stimulus at this stage. It is useful at very low amplitudes if we intend to apply a bit-stealing lookup-table in the next step, or at higher amplitudes if we actually want a visible noise effect.

6. Apply gamma-correction and dynamic-range enhancement effects. This is done by one of the following procedures:

   DEFAULT OPTION: transform each channel’s pixel values through the inverse of the screen non-linearity, then dither each color-channel independently between the DAC values immediately above and immediately below the transformed target level, according to the noisy-bit algorithm of Allard & Faubert (2008);

   OR: transform each channel’s pixel values through the inverse of the screen non-linearity, then re-express the resulting values as 16-bit integers, distributing the more- and less-significant bytes either between color channels or between adjacent pixels in video memory: specialized hardware such as the ViewPixx or Bits#
can then reinterpret the video content as a high-dynamic-range pattern at the expense of either color or resolution;

**OR:** quantize pixel values according to the size of a large (say, 16-bit) pre-generated lookup table, then use the table to look up a triplet of (red, green, blue) DAC values—for monochromatic stimuli this can simultaneously accomplish linearization, bit-stealing (after Tyler, 1997) if desired, and further quantization down to the native precision of the video hardware;

7. If not already accomplished in the previous step, quantize down to 8 bits per color channel (or however many bits are supported natively in video memory).

### 5.4.2 Precise Control of Timing

Shady pushes most of the computational burden of drawing onto the graphics processor. The few remaining necessary CPU operations are performed by an engine, the default implementation of which is an "accelerated" binary written in C++. This provides a significant boost in performance over Python-based engine implementations. Unfortunately, resources are still limited and it is perfectly possible to overload Shady and cause its animations fail to keep up with real time. Shady animations tend to be accurate in the long term, because by design they are all functions of “wall time” rather than frame number or frame-to-frame “delta”. But an excessive amount of Shady content (or load on the GPU and CPU from other sources) will make your animations “skip” frames and hence become uneven in time, and eventually reduce your effective frame rate. Even rare “frame skips” may be problematic in certain scenarios (e.g. if you are conducting motion perception experiments) because of the temporally-broad-band artifacts they introduce.

This document outlines the principal hazards for Shady timing performance, and describes some tips for avoiding them where possible.

#### General system settings

- Configure the desired frame rate on the display you will use for Shady. (e.g. on Windows 10: right-click on Desktop -> Display settings -> Display adapter properties -> “Monitor” tab -> Refresh rate). At the same time, take the opportunity to ensure your screen resolution is set to maximum (this is important for displays that consist of discrete physical pixels, but does not apply if you are using a cathode ray tube display).

- If there is a manufacturer-specific control panel for your graphics card, (for example, “NVidia Control Panel”) then it may expose “vertical sync” as an option (possibly also called “vertical blanking” or “VBL”). Ensure that this is enabled.

- (Windows) If you have multiple displays, ensure that the screen Shady will appear on is selected as the “Main display” in your display settings. We’ve found this is particularly important when running different screens using different graphics cards, as it ensures that the correct display’s intrinsic frame rate is respected. (This does unfortunately mean that the taskbar will need to be on the same screen as Shady.)

- Like most applications, Shady runs more smoothly when its window is in the foreground and fills the screen. For timing-critical applications, do not use a custom window size or a window frame if you want the best performance, and always make sure Shady is the focused window (this is easy to forget when working with the interactive console).

#### Vertical sync issues

- As per above, ensure that this setting is enabled if your manufacturer-specific graphics card settings expose it (as in, for example, the “NVidia Control Panel”).

- Use the `.SetSwapInterval()` method if you need to force a `Shady.World` to slow down its frame rate (e.g. to drop from 60 to 30 frames per second while remaining precisely regular). Note that this only works with the “accelerated” engine and only on some platforms/GPUs—see the docstring for `.SetSwapInterval()`.
• Use Shady’s “tearing test” to check that vertical sync is working: if parts of the vertical stripe appear out of sync with the rest, or you can perceive torn or ragged edges, vertical sync is probably not functioning. You can launch the test from the command line with `python -m Shady tearing` or from inside Python with `Shady.TearingTest(world)`.

Optimization

Though it may seem obvious to say it, you can save time between frames by optimizing any Python code you are using to perform animation updates, aiming to make it maximally efficient. Ensure that your numerical array operations are vectorized, that you’re not reinventing wheels that Shady can do in its accelerated engine or on the GPU (e.g. texture color/contrast modulation using a spatial function), and that you are not computing things unnecessarily. The demo `dots2` allows you to compare three different implementations of the same multi-stimulus animation, and to observe that that `numpy` vectorization (“batch” mode) significantly improves timing performance.

Below are some more specific tips for analyzing and improving your code.

Diagnostic tools

Frame interval gauge: The `Shady.FrameIntervalGauge()` stimulus shows your frame-to-frame interval graphically in real time. Each minor grid unit represents 1 millisecond, and the red lines are spaced every 10 milliseconds. Ideally, the gauge should hover around $1000/f$ milliseconds, where $f$ is your optimal frame rate (e.g. ~16.7 ms for a 60 Hz display). Watch out for spikes, which indicate transient performance drops (frame skips). If these spikes occur unpredictably, check that your operating system hasn’t decided to start a system process behind the scenes (Windows is particularly obnoxious about this). The gauge itself will have a tiny effect on your performance (far less than a text display would, in Shady—see below).

NB: the frame interval gauge requires the third-party package `numpy`

Post-hoc timing plots: Use the `Shady.PlotTimings()` method to show a history of Shady’s frame-to-frame intervals, along with optional additional information about the timings of specific aspects of the engine and individual stimuli. The information is enriched if the `debugTiming` attribute is set to `True` for your `World` and `Stimulus` instances. You can also plot timings from a Shady log file with `Shady.Utilities.PlotTimings(logfilename)”—or the same thing can be invoked from outside Python with `python -m Shady timings LOGFILENAME`.

NB: timing plots require the third-party packages `numpy` and `matplotlib`.

Shady usage tips

Stimulus overload: Unsurprisingly, drawing too many stimuli at once will slow you down— particularly when they overlap, which will require blending operations and/or cause Shady to draw the same pixels multiple times. Keep in mind that drawing a few large stimuli is faster than drawing more numerous smaller stimuli, even if the total number of pixels is the same, due to the parallel architecture of GPU computations. If you have reached your system’s draw limit, consider whether you could:

• combine multiple linked stimuli into one larger stimulus;

• disable stimuli whenever you expect them to be out of sight—you can disable rendering by setting `visible=False`, or disable rendering and inter-frame property updates by calling `Leave()`;

• use other Shady tricks (such as `property sharing`) to reduce the number of update computations per frame;

• use a single multi-shape `Stimulus` (as exemplified in the demos `dots1`, `dots3` and `dots4`) rather than multiple separate `Stimulus` instances (as in `dots2`).
Text properties: Beware of changing the text properties of your stimuli too often if using Shady.Text functionality. Shady is smart enough that it does not re-compute the pixel values of a text stimulus unless there is an actual de-facto change to the text content or style. However, when this does occur, Shady must re-render the texture on the CPU and send the result to the GPU. This is out of step with Shady's usual approach of doing all pixel processing on the GPU, and is almost as expensive as creating a new stimulus every time you change the text content. One workaround for rapidly switching between a number of pre-determined text objects is to create them in advance, each as a separate page of the same Stimulus instance, and switch between them.

Video file playback: Similarly, be mindful of the cost of video playback using Shady.Video. If you’re working from a video file, each new frame must be decoded on the CPU. In addition, regardless of whether it came from a file or from a live camera, the frame must be sent from CPU to GPU. Unfortunately, we currently have no workaround for this other than rendering the particular frames you need as a multi-frame Stimulus, and this may not be feasible for long videos. Lowering the video file’s resolution, and/or restricting the video aperture, may reduce the impact on performance somewhat.

External operations: Be mindful of the impact of concurrent non-Shady CPU and GPU operations, both inside and outside of Python.

If a significant number of concurrent operations are being performed inside the same instance of Python that is running Shady, you should read the note at the end of Shady.Documentation.Concurrency about multi-threading in Python—the short version is that you may really want to run multiple processes of Python, and give Shady its own process, rather than using Python’s controversial (arguably illusory) multi-threading.

5.4.3 Gamma Correction, Dynamic Range Enhancement, and the Canvas

- Overview
- Gamma Correction
- Dynamic Range Enhancement
- Look-up Table
- Canvas
- Avoiding image artifacts

Overview

One good way to start a visual psychophysics experiment might be:

```python
world = Shady.World( canvas=True, backgroundColor=0.5, gamma=2.2 )
```

This automatically creates a canvas, and configures its background color and gamma correction. “Noisy-bit” dithering, for dynamic range enhancement, is also turned on by default: the default value of ditheringDenominator, for the World and all Stimulus instances, will be automatically set to 255 for most graphics cards, or the appropriate larger \( 2^{\text{n}} - 1 \) value if your graphics card offers \( n>8 \) bits per DAC.

Now, when you create another stimulus (let’s say, a Gabor patch), you’ll most likely want it to have the same backgroundColor, gamma, ditheringDenominator, noiseAmplitude and look-up table as the World and its canvas. One way of ensuring that these properties always match the surrounding World is to use property sharing, and one powerful shortcut is to share the virtual property atmosphere which encompasses all of these linearization- and dynamic-range-related properties. So:
That’s equivalent to, but easier to type than:

```python
gabor = world.Stimulus(
    signalFunction = Shady.SIGFUNC.SinewaveSignal,
    signalAmplitude = 0.5,
    plateauProportion = 0,
    atmosphere = world,            # matched and linked until further notice
    backgroundColor = world,
    gamma = world,
    ditheringDenominator = world,
    noiseAmplitude = world,
    lut = world,
)
```

... and that in turn is equivalent to, but easier to type than:

```python
gabor = world.Stimulus(
    signalFunction = Shady.SIGFUNC.SinewaveSignal,
    signalAmplitude = 0.5,
    plateauProportion = 0,
)
    .LinkPropertiesWithMaster( world,
    'backgroundColor', 'gamma', 'ditheringDenominator', 'noiseAmplitude',
    'lookupTableTextureSize', 'lookupTableTextureSlotNumber', 'lookupTableTextureID',
)
```

In the following sections, we’ll unpack some of the issues covered above.

**Gamma Correction**

The mapping from pixel intensity values to physical luminance is not necessarily linear. Instead, it often appears to take the shape of an exponential “gamma” function. Display software must usually correct for this by applying the inverse gamma function to pixel intensities before displaying them: this cancels out the non-linearity and causes pixel luminance to be linear again.

You can apply gamma correction to each Shady Stimulus instance by setting its `gamma` property, either during construction or subsequently:

```python
# ...
stim = world.Stimulus( gamma=2.2 ) # a scalar works, for setting
    # all three color channels the same
# ...
stim.gamma = (2.3, 2.2, 2.1) # a sequence also works
```

The standard gamma correction that modern displays are supposed to adopt is the sRGB profile, which is a standard piecewise function that follows the \( \gamma = 2.2 \) exponential curve quite closely (although the exponent it uses in its exponential portion is actually slightly higher). You can tell Shady to use the sRGB correction instead of an exponential function by setting `gamma` to -1:
Note that gamma correction will depend on the resolution of your monitor. Displays will only be configured to the sRGB profile at their native resolution, and there will likely be a profile in your monitor’s settings for sRGB specifically. For more about gamma correction, see the Wikipedia article.

**Dynamic Range Enhancement**

Monitors have a limited dynamic range, which determines how precisely they can present small variations in luminance: close to threshold contrast (not coincidentally) the monitor’s ability to display very small contrasts breaks down due to the discrete number of available pixel intensities.

The limitations on maximum contrast are innate to the display hardware, but there are tricks to work around the constraints on minimum contrast. Shady provides two such techniques: “noisy bit” dithering (which is the recommended approach, enabled by default) and additive noise. Both of these techniques apply spatiotemporal noise to the drawn pixel values of whichever Stimulus objects you apply them to (including the World’s canvas).

“Noisy-bit” dithering (Allard & Faubert 2008) applies a simple stochastic algorithm before converting floating-point RGB pixel intensity values to the discrete integer DAC values that are passed to the monitor hardware. Floating-point RGB values that map to non-integer DAC values are rounded with a weighted probability inversely proportional to their distance from the integer values. For example, every time Shady is instructed to draw a pixel with intensity $(0.5, 0.5, 0.5)$, the desired DAC value on an 8-bit graphics card is 127.5, half-way between two integer values: with noisy-bit dithering on, each color channel will then have a 50% chance of being rounded down to DAC value 127 and a 50% chance of being rounded up to DAC value 128. Similarly, every time Shady is instructed to draw a pixel with intensity 0.25, the target value is 63.75, so the pixel will have a 25% chance of being rounded down to 63 and a 75% chance of being rounded up to 64. This probabilistic conversion is done independently for every color channel in every pixel in every frame, and the resulting noise causes the luminance values to perceptually average to the desired between-DAC value. Noisy-bit dithering is enabled by default. The only property needed to control it is ditheringDenominator, which will be automatically set to the highest DAC value your monitor can produce (usually 255). Other positive values will cause levels of rounding granularity that are not suited to your hardware, and should be avoided. You can negate the value, or set it to 0, to turn dithering off.

Additive noise follows a similar principle to “noisy bit” dithering, but simply adds random noise to the floating-point value of each pixel before it is linearized, looked up in a look-up table, or converted to a discrete DAC value. The resulting noise should again cause the luminance values to perceptually average to the desired luminance and/or color. You can control the strength of this noise by setting the noiseAmplitude property (or its alias, noise). Use negative values for uniform noise, or positive values for Gaussian noise. Noise is computed once for all color channels of the same pixel, but may be scaled separately per channel, so you can set noiseAmplitude to an RGB triplet if you want to tint the noise (or a single value to set all three channels’ noise amplitude the same). Additive noise is useful if you want to perform “bit stealing” (Tyler 1997), which can be accomplished using a look-up table: the bit-stealing technique introduces small-amplitude step changes in chroma which can sometimes become perceptible if their spatial extent is large: noise can effectively break these areas up.

The differences between the two properties are summarized in the table below:
Noise Amplitude...  Dithering Denominator...

- is added before gamma correction (or look-up table lookup);
- may be scaled differently in different color channels;
- creates noise that is otherwise perfectly correlated across color channels;
- creates independent noise in each color channel;
- is useful in combination with a bit-stealing look-up table, or when you actually want visible noise;
- can be scaled arbitrarily, and may be uniform (when property value is negative) or Gaussian (when positive).

- is applied after gamma correction;
- has the same amplitude on average in all color channels;
- creates independent noise in each color channel;
- is recommended for most purposes (including when visible noise is used) but is disabled automatically when a look-up table is in use;
- only dithers between two nearest DAC values; the correct property value (which will be found automatically) is $2^{\text{bits}} - 1$ where \text{bits} is the bit depth of your graphics card (usually 8).

The following demos may provide further insight:

- \texttt{examples/dynamic-range.py} allows you to visualize and interactively explore various dynamic-range-enhancement options.
- \texttt{examples/dithering.py} performs a numerical sanity-check of our noisy-bit dithering implementation.
- \texttt{examples/precision.py} performs a quantitative analysis of the effective precision achieved by noisy-bit dithering.
- \texttt{examples/noise.py} allows you to examine the distribution of random values created by the additive noise effect.
- \texttt{examples/fancy-hardware.py} illustrates Shady’s support for rendering on specialized vision-science hardware, such as the Bits# or ViewPixx, that can achieve high dynamic range without dithering (see also the \texttt{bitCombiningMode} property of the \texttt{Shady.World} class).

Look-up Table

Instead of using the \texttt{gamma} property to perform automatic gamma-correction, and allowing the \texttt{ditheringDenominator} to perform automatic noisy-bit dithering, you can disable both of these features...
and take control of linearization and dynamic-range enhancement issues directly yourself, by specifying a look-up 
table (LUT).

A look-up table is a discrete series of entries corresponding to a discrete (usually large, like 65536) number of ideal-
luminance ranges that equally divide up the complete range from 0 to 1. Each entry is a triplet of integers, correspond-
ing to the red, green and blue DACs (for most graphics cards, these will be 8-bit integers).

This is useful only for stimuli whose intensity is one-dimensional (e.g. monochromatic stimuli). In fact, Shady only 
uses the first color channel (red) to compute indices into the LUT. The output of the LUT will be RGB or RGBA, 
however. This means that using a LUT is a form of “indexed color” image rendering.

Here is a trivially small example of a 2-bit LUT (i.e. 4 entries) for an 8-bit graphics card (i.e. DAC values go up to 
255):

```python
stim.lut = [
    [ 0, 0, 0 ], # ideal luminances 0 - 0.25 map to black
    [ 255, 0, 0 ], # 0.25 - 0.5 map to red
    [ 255, 255, 0 ], # 0.5 - 0.75 map to yellow
    [ 255, 255, 255 ], # 0.75 - 1.0 map to white
]
```

To attach a LUT to a Stimulus, the easiest way is to call the SetLUT() method or, equivalently, assign to the lut property. You can assign either a Shady.LookupTable instance, or a valid argument to the Shady. 
LookupTable class constructor (in which case, such an instance will be constructed automatically). This means 
that in practice you can assign:

- an existing Shady.LookupTable instance
- an n-by-3 (or m-by-n-by-3) array of integers (or a nested list that numpy can automatically convert into such an 
array, as in the example above)
- a filename of a npy, npz or png file in which you have previously saved a LUT array with Shady. 
  Linearization.SaveLUT()

When you then query the lut property, you will see that its value is a Shady.LookupTable instance. Note that 
creation of such an instance allocates a texture in OpenGL, so the most efficient use of resources would be to re-use 
Shady.LookupTable instances wherever appropriate.

Remember that assigning a look-up table disables automatic gamma-correction and noisy-bit dithering. Assign-
ing stim.lut = None or calling stim.SetLUT(None) removes the look-up table and re-enables automatic 
gamma-correction and noisy-bit dithering.

It is up to you to specify appropriate values for the LUT entries, although Shady does provides a utility for computing 
them according to one particular strategy: Shady.Linearization.BitStealingLUT() which implements a 
version of the “bit-stealing” technique (after Tyler 1997).

Bit-stealing allows monochromatic stimuli to be rendered at higher effective dynamic range, by allowing very small 
chromatic variations: these create luminance levels between the existing strictly-gray levels, while hopefully keeping 
the chromatic information itself well below the subject’s threshold. The latter point can fail in some circumstances 
where there is a very gradual change as a function of distance (such as at the outer edges of a Hann window): then 
it is sometimes possible to see a small step-change in color between large adjacent areas. To break up this effect, it 
is sometimes useful to add a little noise to the signal (as in the dithering approach, the effect of this noise will be 
perceptually averaged away over small spatial and temporal scales). We’ve found noiseAmplitude=1e-4 works 
well.

The examples/dynamic-range.py demo has look-up-table and additive-noise options, and illustrates some of these 
points.
Canvas

If you create a World:

```python
world = Shady.World()
```

it starts off filled with a uniform color. You can specify this color in in the constructor call, or manipulate it after construction, via the `clearColor` attribute:

```python
world.clearColor = [ 1, 0.3, 0.5 ]
```

Yeesh. Now, this may be sufficient for some purposes. But `clearColor` is a very simple property that does not change according to your linearization or dynamic-range-enhancement parameters: it is never gamma-corrected, and is always applied completely uniformly, so there can be no dithering.

However, if you’re doing vision science, you’ll probably want both gamma-correction and dynamic-range enhancement in your stimuli. And if you have those things in your `stimuli`, you’ll probably need them in the `backdrop` as well—for example, you may need to eliminate the risk that a keen-eyed subject can detect the edge of your stimulus bounding-box because of a just-visible artifact at the boundary between dithered and un-dithered gray regions.

The solution is to create a “canvas”, which is simply a rectangular `Stimulus` that fills your World. This can be done during `World` construction:

```python
world = Shady.World( canvas=True )
```

...or after the fact:

```python
world = Shady.World()
world.MakeCanvas()
```

Either way, what you get is a `Stimulus` object with no foreground color, the name `canvas`, and a `z` value of +1 (i.e. as far as possible away from the camera). In addition, various properties of the canvas are linked to those of the `World` itself. So if you specify or change any of the following properties of `world`:

- `world.backgroundColor`
- `world.gamma`
- `world.ditheringDenominator`
- `world.noiseAmplitude`
- `world.lut`
- `world.outOfRangeColor`
- `world.outOfRangeAlpha`

you will actually be affecting the corresponding properties of `world.stimuli['canvas']`. Indeed, these properties of the World are only placeholders and are ignored during rendering of the empty `World` itself at the start of each frame. Changes in their values only cause visible effects to the extent that they change `Stimulus` instances, such as the canvas, that are linked in this way.

Avoiding image artifacts

For all stimuli:

Be aware that you may introduce artifacts due to your graphics card’s linear interpolation between pixel values, whenever you use:
• `envelopeRotation` values that are not divisible by 90,

• `envelopeScaling` values other than 1.0,

• or non-integer values in the first two coordinates of `envelopeOrigin` (but if you stick to using `envelopeTranslation` instead of `envelopeOrigin`, your stimulus position on screen will always be rounded to an integer number of pixels, avoiding this pitfall).

For similar reasons, you should always run your display screen at its native resolution.

For textured stimuli:

Transformations of the carrier signal (via `carrierTranslation`, `carrierRotation` and `carrierScaling`) will also lead to interpolation artifacts as above, if the carrier content comes from a texture, i.e. it is defined by a discrete array of pixels.

For untextured (functionally-generated) stimuli:

If, on the other hand, the carrier content is entirely functionally generated on the GPU functions using the `signalFunction`, `modulationFunction` and `windowingFunction` properties, then you do not need to worry about interpolation artifacts from carrier transformations, because the carrier transformations are applied to the coordinate system before the functions are even evaluated.

You should also check whether a carrier transformation pushes your signal beyond any spatial or spatio-temporal aliasing limits. For example, if you have created an antialiased square-wave signal function as in the `examples/custom-functions.py` demo, you may think that the function automatically avoids components with fewer than 2 pixels per cycle. But if you then shrink it with a `carrierScaling` factor < 1.0, you may be back in trouble.

For moving stimuli:

Remember that speed (pixels per second or degrees per second) multiplied by spatial frequency (cycles per pixel or cycles per degree) gives you the flicker frequency of a pixel in Hz (cycles per second). If that is greater than half your screen’s refresh rate (i.e. > 30Hz, for most commercial screens) then you’re into spatio-temporal aliasing territory (that parallel universe where helicopter blades slow down to a standstill and car wheels spin backwards).

### 5.4.4 Luminance and Contrast

**Definitions**

**Ideal Luminance:**

This means the intensity of a pixel expressed on a scale from 0 to 1, where 0 is the lowest intensity the visual display hardware can produce, and 1 is the highest. It is often convenient to work with ideal luminances because they allow you to express stimulus characteristics in a hardware-independent way. However you would need to convert them to physical luminances (perhaps using the function `IdealToPhysicalLuminance()`) before reporting them in a publication.

In Shady, the carrier content of a `Stimulus` is generally expressed on this ideal 0-to-1 scale: this includes its `signalAmplitude`, the pixel values in its floating-point texture array (if any), its `noiseAmplitude` and its `backgroundColor`. The only exception is when texture content is expressed as an 8-bit integer array (values ranging from 0 to 255) - for example when it is loaded directly from an image file.

**Physical Luminance:**

This means the intensity of a stimulus patch in real physical units such as candela / m². Given accurate gamma-correction, we assume that physical luminance is proportional to ideal luminance, with an additive offset. The offset is a consequence of the fact that a screen’s minimum intensity is never truly zero (either
due to reflected ambient light, or the imperfect black level of most current screen technologies). An OLED screen in an otherwise perfectly dark room may achieve close to 0 - this is currently the only setup in which you can ask how much more black the screen could be and the answer is none. None more black.

Ideal Contrast Ratio:

By “ideal contrast ratio”, we mean “a contrast ratio computed from ideal luminance values”, independent of whether this is computed by the RMS method or the Michelson method. Since ideal luminances go down to zero, ideal contrast ratios can reach 1.0, unlike physical contrast ratios.

It is sometimes convenient to work with an ideal contrast ratio because it can be computed without having to perform actual photometer measurements, and can provide an approximation to the corresponding true physical contrast ratio. But you would need to convert it to a physical contrast ratio (perhaps with the function `IdealToPhysicalContrastRatio()` ) before reporting it in a publication.

Physical Contrast Ratio:

By “physical contrast ratio”, we mean “a contrast ratio computed from physical luminance values”, independent of whether this is computed by the RMS method or the Michelson method. Since the physical luminance at the location of a “black” pixel is never actually zero, a physical contrast ratio will never reach 1.0 (though it may come very close, depending on the screen technology and ambient light control).

RMS Contrast Ratio:

This is a method of computing a contrast ratio according to $\sqrt{\frac{1}{N} \sum_{x} \sum_{y} (L(x,y) - \mu)^2}$, or in plain text:

```
average_over_x_and_y( (luminance(x,y) - background_luminance) ** 2 ) ** 0.5
```

If $L(x,y)$ and $background_luminance \ L_\mu$ are both ideal luminances, then the result is an ideal contrast ratio. If they are both physical luminances, then the result is a physical contrast ratio.

Michelson Contrast Ratio:

This is a method of computing a contrast ratio according to $\frac{L_{\text{max}} - L_{\text{min}}}{L_{\text{max}} + L_{\text{min}}}$, or in plain text:

```
L_max - L_min
-------------
L_max + L_min
```

If $L_{\text{max}}$ and $L_{\text{min}}$ are both ideal luminances, then the result is an ideal contrast ratio. If they are both physical luminances, then the result is a physical contrast ratio.

The Michelson method is best suited to simple periodic stimuli whose luminance varies equally above and below the background luminance. For visual noise or natural stimuli you would probably want to use an RMS contrast ratio instead.

Normalized Contrast:

This may be a somewhat misleading name, but we use it anyway, as do other psychophysical software packages. To mitigate confusion, we are careful always to include the word “ratio” when talking about RMS and Michelson contrast ratios, and to omit it here when talking about normalized contrast, which is not a ratio.

By “normalized contrast”, as in the property `Stimulus.normalizedContrast`, we mean a straightforward scaling factor that acts as a multiplier on deviations of luminance from the background, regardless of absolute luminance level:

```
luminance = background_luminance + signal * normalized_contrast
```
So, for example, if our background_luminance is 0.5, in ideal units, and our signal is $0.5 \times \sin(x)$, a normalized_contrast of 1.0 would allow this sine-wave signal to span the full luminance range of the screen (note that the background_luminance could not be anything other than 0.5, otherwise the signal would go out of range). A normalized_contrast of 0.2 would mean that the same signal spans one fifth of the full intensity range of the screen (so now you could set the ideal background_luminance to anything from 0.1 to 0.9).

**Conversion utility functions**

- `Shady.Contrast.IdealToPhysicalLuminance()`
- `Shady.Contrast.PhysicalToIdealLuminance()`
- `Shady.Contrast.IdealToPhysicalContrastRatio()`
- `Shady.Contrast.PhysicalToIdealContrastRatio()`
- `Shady.Contrast.IdealContrastRatioToNormalizedContrast()`
- `Shady.Contrast.NormalizedContrastToIdealContrastRatio()`

**Contrast-ratio computation functions (ideal or physical)**

- `Shady.Contrast.RMSContrastRatio()`
- `Shady.Contrast.MichelsonContrastRatio()`

### 5.4.5 Managed Properties and Managed Shortcuts

#### Managed Properties

Most properties of a Shady World or Shady Stimulus are managed properties, which allows them to be manipulated in a number of powerful ways. Many of them are simply containers for values that are automatically transferred to the GPU on each frame, where the shader handles all of the computations required for drawing. Managed properties are mediated by descriptors of class `ManagedProperty`.

Managed properties support intelligent assignment. They are all technically arrays, and you can always assign a tuple, list or `numpy.ndarray`, or any other sequence of numeric values. For the sake of syntactic laziness you can also just assign a single scalar value, if you want that value to be copied automatically to all elements. For example:

```
# Let's assume you have a running World, and that `stim` is a reference to
# one of the Stimulus instances that is being rendered by that World.

stim.scale = (3, 1)  # anisometric scaling
print( stim.scale )  # --> 3, 1

stim.scale = 4       # isometric scaling in x and y
print( stim.scale )  # --> 4, 4

stim.color = 0.5     # a simple way to set color to mid-gray
print( stim.color )  # --> 0.5, 0.5, 0.5
```
A property may be addressed by its canonical name, but also by aliases. These are provided to allow code to be less verbose, and to ease the burden on one’s memory for arbitrary names. For example, the `envelopeTranslation` property answers to any of the following names:

```
stim.envelopeTranslation
stim.position
stim.pos
stim.xy
```

Similarly, you don’t have to remember which of the following synonyms is correct, because they all work:

```
stim.envelopeRotation
stim.orientation
stim.rotation
stim.angle
```

Some properties also allow their individual *elements* to be addressed by name (see the subsection on Managed Shortcuts below).

The `Set()` method can be used to set multiple properties or shortcuts at once:

```
stim.Set( scale=3, color=0.5 )
```

The class method `SetDefault()` allows you to configure the default values of managed properties for all future `World` or `Stimulus` instances until the end of the current session:

```
Shady.World.SetDefault( anchor=[ -1, -1 ] )
# Now, all future `World` instances will have the origin of their coordinate systems in the bottom left corner

Shady.Stimulus.SetDefault( color=[ 1, 0, 1 ], angle=30 )
# Now, all future Stimulus instances will be tinted pink and lean thirty degrees to the left by default when created. How useful.
```

Managed properties also support dynamic value assignment. This means that, rather than simply assigning a static numeric value to the property, you can tell that property how to change its value as a function of time:

```python
# ...
def dynamic_scale( t):
    return ( t % 1, t % 2 )  # ( x, y )
stim.scale = dynamic_scale

# or equivalently:
stim.scale = lambda t: ( t % 1, t % 2 )
```

Note that a callable function object, rather than a static numeric value, is being assigned to the Stimulus’s managed `scale` property. Shady will evaluate this function on every frame to determine the final property value for drawing. When you query the property value, for example by saying `print( stim.scale )`, you will still get a numeric result—whatever the current value is at the time the command is issued. If you actually wanted to retrieve the function object, you would say `stim.GetDynamic( 'scale' )` instead.

Finally, managed properties can be intelligently copied, shared, and inherited across stimuli. The simplest way to share a property between two stimuli is to assign the stimulus itself:

```python
# ...
stim1.color = lambda t: ( t % 10 ) / 10  # dynamic: slowly change from black to white every ten seconds
stim2.color = stim1  # assign Stimulus instance to share color with
```

(continues on next page)
stim1.ResetClock(); time.sleep( 5 ); print( stim2.color )
# --> 0.5, 0.5, 0.5  (because five seconds have passed)

Note that sharing is bi-directional, because it causes the properties of both stimuli to point to the same array of numbers in memory:

```python
stim2.color = stim1  # first, forge the link
stim1.color = 1, 0, 1  # then, change the value here...
print( stim2.color )  # ...and the change is felt here
# --> 1, 0, 1
stim2.color = 0, 0, 1  # ...and vice versa:
print( stim1.color )  # turnabout is fair play.
# --> 0, 0, 1
```

See Property Sharing for a more in-depth explanation.

**Managed Shortcuts**

Some managed properties provide subscripting shortcuts that allow each element to be accessed by name. These are mediated by descriptors of class ManagedShortcut. Examples:

```python
stimulus.Set( xscale=10, red=1, blue=0.5 )  # change horizontal scale, red color, channel, and blue color channel
print( stimulus.scaling )  # y scaling remains at its previous (default) value
# --> 10, 1
print( stimulus.color )  # green channel remains at its previous (default) value
# --> 1, -1, 0.5
stim.x = 100
print( stim.x )  # .x is a shortcut for .envelopeTranslation[0]
# --> 100
print( stim.envelopeTranslation )
# --> 100, 0  (y remains at its default value)
```

The Set() instance method and SetDefault() class method support managed shortcuts just as they do for managed properties. Also, managed shortcuts support dynamic value assignment in the same way that full managed property arrays do:

```python
import time
stim = lambda t: t ** 2
stim.ResetClock(); time.sleep( 5 ); print( stim.position )
# --> 100, 25
```

On each frame, dynamics are evaluated for full ManagedProperty arrays first, and then for ManagedShortcut values. This allows you to (for example) dynamically control the color property and then, independently and also dynamically, override just the red channel value.

Note that shortcuts cannot be shared directly between instances in the way that full property arrays can. This is because sharing is accomplished by sharing the memory segment for an entire property array or not at all (See Property Sharing for more details). If you need to work around this limitation, one way to do so (at the expense of a few extra CPU cycles per frame) is to use a dynamic value:
Unmanaged Dynamic Properties

Some properties, despite not being managed themselves, support *dynamic value assignment*. Many such properties affect managed properties *indirectly*. For example, the following properties of *Stimulus* support dynamics, and indirectly manipulate managed properties:

- **frame**: Changing the value of the *frame* property causes *carrierTranslation[0]* to change in discrete steps, thereby showing different parts of a texture at different times. This is one way to *animate multi-frame images*.

- **page**: This property allows indirect manipulation of multiple properties that affect the stimulus carrier texture. This is another way to *animate*.

- **scaledSize**, **scaledWidth** and **scaledHeight**: These properties allow indirect manipulation of the managed property *envelopeScaling*, dependent on the base *envelopeSize*, to achieve a target size expressed in pixels on screen.

- **points** and **pointsComplex**: These properties allow simultaneous manipulation of *nPoints* and *pointsXY*, providing a view into the array of points either as a two-column array (*points*) or as a one-dimensional array of complex numbers (*pointsComplex*).

By contrast:

- **text** supports dynamic value assignment (as a shortcut for assigning *text.string*), but this does not work via indirect manipulation of a managed property.

Finally, it’s worth noting that a dynamic can be associated with any attribute name at all: it will still be evaluated, and the result assigned, once per video frame. However, newly-created attributes will not support the lazy syntax of dynamic value *assignment*, so if you do this:

```python
stim.foo = lambda t: t * 2
print( stim.foo )
```

you can see that *stim.foo* really is a lambda object, just as you would expect from Python’s default behavior. The way to create a custom dynamic is with *SetDynamic()*:

```python
stim.SetDynamic( 'bar', lambda t: t * 2 )
```

Then *stim.bar = t * 2* will be performed automatically once per frame while the *World* is running.

**List of Managed Properties for the *World* Class**

In each case the first name given is the “canonical” name. Subsequent names, if any, are aliases. Names in brackets, if any, indicate managed shortcuts.

- *size* = [width, height]
- *clearColor* = [red, green, blue]
- *anchor* = *origin* = [ax_n, ay_n]
- *timeInSeconds* = *t*
- *visible* = *on*
- *backgroundColor* = *bg* = *bgcolor* = [bgred, bggreen, bgblue]
- `noiseAmplitude = noise = [rednoise, greennoise, bluenoise]`
- `gamma = [redgamma, greengamma, bluegamma]`
- `ditheringDenominator = dd`
- `OutOfRangeColor`
- `OutOfRangeAlpha`

**List of Managed Properties for the Stimulus Class**

In each case the first name given is the “canonical” name. Subsequent names, if any, are aliases. Names in brackets, if any, indicate managed shortcuts.

- `envelopeSize = size = [width, height]`
- `textureChannels`
- `textureSize`
- `visible = on`
- `z = depth = depthPlane`
- `envelopeTranslation = envelopePosition = pos = position = xy = [x, y]`
- `envelopeOrigin = [ox, oy, oz]`
- `envelopeRotation = angle = orientation = rotation`
- `envelopeScaling = scale = scaling = [xscaling, yscaling]`
- `anchor = [ax_n, ay_n]`
- `useTexture`
- `carrierRotation = cr`
- `carrierScaling = cscale = cscaling = [cxscale, cyscale]`
- `carrierTranslation = [cx, cy]`
- `offset = [addr, addg, addb]`
- `normalizedContrast = contrast`
- `plateauProportion = pp = [ppx, ppy]`
- `signalFunction = sigfunc`
- `signalParameters = [signalAmplitude, signalFrequency, signalOrientation, signalPhase]`
- `modulationFunction = modfunc`
- `modulationParameters = [modulationDepth, modulationFrequency, modulationOrientation, modulationPhase]`
- `windowingFunction = windowFunction = winfunc`
- `backgroundAlpha = bgalpha`
- `backgroundColor = bg = bgcolor = [bgred, bggreen, bgblue]`
- `noiseAmplitude = noise = [rednoise, greennoise, bluenoise]`
- `gamma = [redgamma, greengamma, bluegamma]`
• \textit{ditheringDenominator} = \textit{dd}
• \textit{color} = \textit{fg} = \textit{fgcolor} = \textit{foregroundColor} = [\textit{red}, \textit{green}, \textit{blue}]
• \textit{alpha} = \textit{fgalpha} = \textit{opacity}
• \textit{outOfRangeColor}
• \textit{outOfRangeAlpha}
• \textit{penThickness}
• \textit{smoothing}
• \textit{drawMode}

5.4.6 Making Properties Dynamic

Individual Properties

Many attributes of a \textit{Shady.World} or \textit{Shady.Stimulus} instance are what we call “managed properties”. A key feature of these properties is the ability to be made dynamic. This can be done simply by assigning a function object to the property instead of a static value or array. The function should take one argument, \textit{t} (for time, in seconds), and return whatever value (or array of values) you want that property to have at time \textit{t}. On every frame callback, \textit{Shady} will run any dynamic property functions you have assigned using the current \textit{World} time and update the value of the corresponding properties.

Note that there are multiple ways to create callable objects in Python. The standard way is to define a function using \texttt{def}:

```python
import Shady
world = Shady.World( 700, top=100, frame=True )
stim = world.Patch()
def simple_spin( t ):
    return t * 45
stim.rotation = simple_spin  # note no parentheses – we are assigning the
                          # function object itself, not its output
```

Another way is to specify an anonymous function in-line using \texttt{lambda}:

```python
stim.rotation = lambda t: t * 45
```

Both methods are valid. Just remember that \texttt{lambda} functions are restricted to containing just the single expression you want to return from the function.

Note that \textit{t} is measured in seconds, and by default it is the number of seconds elapsed since the \textit{World} first started rendering stimuli. So if it has been a long time since the \textit{World} started, your \textit{Stimulus} will likely be off the screen in the above example. One option is to give the \textit{Stimulus} its own independent “time zero”. This can be reset to the current time using the call:

```python
stim.ResetClock()
```

There are no unusual restrictions on your dynamic functions, provided that they take exactly one argument and return a value or sequence that is appropriate for the property with which they are associated. (It is also legal for them to return \texttt{None}, in which case the property value is not changed.) Any Python variables or objects that are accessible in the same namespace can be used and modified:
Dynamic functions are free to ignore the time variable. You can make the properties of your stimulus dependent on whatever variables you want:

```
import Shady
world = Shady.World( 700, top=100, frame=True )
stiml1 = world.Patch(
    pp = 1,
    x = lambda t: ( t % 1.0 ) * 300,
)
stiml2 = world.Patch(
    pp = 1,
    color = [ 1, 0, 0 ],
    y = lambda _ : stiml1.x, # ignores the time input
)
```

Note that we still have to define our dynamic function with exactly one argument so that Shady can pass in the stimulus’s clock, but name it _ as a convention to indicate that this argument is not used.

Note also that the function references stim1.x which is itself dynamic. Whenever you access a managed property, its current static value (or array of values) will be returned, even if the property is dynamic. If you want to retrieve the actual function object being used to calculate its dynamics, use the `Shady.Stimulus.GetDynamic` method:

```
print( stim1.x )
# --> 294.8838
print( stim1.GetDynamic( 'x' ) )
# --> <function __main__.<lambda>(t)>
```

More generally, you can set a Shady property to any callable object that takes exactly one argument. This includes any instance of a class with a __call__ method defined, provided the call takes one argument. The optional `Shady.Dynamics` submodule offers several useful classes designed to be used as dynamic properties in Shady, such as the `Integral` for integrating arbitrary functions over time or the `Transition` for smoothly transitioning between a start and end value.

NOTE: Be wary when using the same public variable to define multiple dynamics functions in a row. Because of how functions interact with their namespace in Python, the current (i.e. last set) value of that variable will be used when the dynamics are evaluated on each Shady frame callback. This includes simple looping variables! If you want to ‘freeze’ the value of a public variable when defining a dynamic function, you will need to separate it from that variable’s namespace, e.g. by using a nested function or by passing the variable to a lambda as a keyword argument:

```
### WRONG ###
import Shady, math
world = Shady.World( 700, top=100, frame=True )
stimuli = []
amplitudes = [100, 200, 300]
for amplitude in amplitudes:
    stim = world.Stimulus()
    stim.x = lambda t: amplitude * math.sin( 2 * math.pi * t )
    stimuli.append( stim )
# all three stimuli will use amplitude == 300 when their dynamics are evaluated!
```

(continues on next page)
### ALSO WRONG ###

```python
import Shady, math
world = Shady.World( 700, top=100, frame=True )
stimuli = []
amplitudes = [100, 200, 300]
for i in range(3):
    stim = world.Stimulus()
    stim.x = lambda t: amplitudes[i] * math.sin(2 * math.pi * t)
stimuli.append( stim )
# all three stimuli will use i == 2, i.e. amplitudes[2]!
```

### RIGHT (nested function) ###

```python
import Shady, math

def create_oscillation_dynamic( amplitude )
    # the argument `amplitude` is retrieved from a frozen
    # version of the namespace of this function
    return lambda t: amplitude * math.sin(2 * math.pi * t)

world = Shady.World( 700, top=100, frame=True )
stimuli = []
amplitudes = [100, 200, 300]
for amplitude in amplitudes:
    stim = world.Stimulus()
    stim.x = create_oscillation_dynamic( amplitude )
    stimuli.append( stim )
```

### ALSO RIGHT (lambda keyword) ###

```python
import Shady, math

world = Shady.World( 700, top=100, frame=True )
stimuli = []
amplitudes = [100, 200, 300]
for amplitude in amplitudes:
    stim = world.Stimulus()
    stim.x = lambda t, a=amplitude: a * math.sin(2 * math.pi * t)
    stimuli.append( stim )
```

Also note that properties of your `World` instance can be made dynamic using all of the methods described above. For example, to create a world whose background color oscillates between black and white:

```python
import math
import Shady
world = Shady.World( clearColor=lambda t: 0.5 + 0.5 * math.sin(2 * math.pi * t) )
```

The world’s dynamics will be updated before any of the stimuli it contains, and its stimuli are updated according to their draw order (i.e. `z`). Stimuli with the same `z`-value will be drawn in the order they were created.

**The Animate Method**

As the behavior of your stimulus grows more complex and its properties become more interdependent, you may begin to find that relying on individual property dynamics becomes unwieldy. In this case, you will likely want to use the stimulus’s `Animate()` method, which is evaluated before any property dynamics on each Shady frame callback.

The only practical difference between the `Animate()` method and any dynamic properties is that `Animate()` takes
a `self` argument, which makes it easier to refer to the stimulus in your logic (e.g. for checking and modifying its state). The function does not need to return any value, which means that you will most likely want to create it using the standard `def`. Once created, pass the function object to the `SetAnimationCallback()` method to properly bind it to the stimulus:

```python
import Shady, math, time
world = Shady.World( 700, top=100, frame=True )
bike = world.Patch( color=[1, 0, 0], pp=1 )

bike.is_bouncing = False
bike.bounce_t0 = None

def bounce( self, t ):
    if self.is_bouncing:
        if self.bounce_t0 is None:
            self.bounce_t0 = t
            # Note use of `_t` in the lambda to distinguish it from the bounce() Argument 't'.
        self.y = lambda _t: 100 * abs( math.sin( 2 * math.pi * (_t - self.bounce_t0 ) ) )
    else:
        if self.bounce_t0 is not None:
            self.bounce_t0 = None
            self.y = 0

bike.SetAnimationCallback( bounce )  # again, note that function object is assigned
bike.is_bouncing = True  # set it back to False to stop the bounce
```

This example is a little more complex than any of the examples in the previous section, but that’s exactly why the `Animate()` method is useful. The `bounce()` function assigns a bouncing dynamic to the stimulus’s y-coordinate whenever `is_bouncing` is set to True, making sure that the stimulus only starts bouncing at that moment. It abruptly resets the y-coordinate to zero whenever `is_bouncing` is set to False. (The optional `Shady.Dynamics` submodule contains a `StateMachine` class that makes it easier to switch your stimuli between different modes of behavior like this.)

If your animation callback has two arguments (i.e. a `self` as well as just a `t`) then you must use the `SetAnimationCallback()` helper to properly bind your function as the `Animate()` method of the instance, so that Python knows that the Stimulus instance should be passed in as the `self` argument. The following will not work:

```python
### WRONG ###
# ...
stim.Animate = bounce
```

If your callback has only one argument, it is interpreted as time `t`—in this case, you can use `SetAnimationCallback()` or just directly assign `stim.Animate = func`.

As with dynamics, instances of the `World` class can have an `Animate()` method set in the exact same way as instances of the `Stimulus` class.

Note that that `Stimulus` and `World` instances provide have an attribute `AnimationCallback` which can be used as a decorator, as a syntactic alternative to calling `SetAnimationCallback()`:

```python
@stim.AnimationCallback
def bounce( self, t ):
    # ...
```
Order of Dynamic Evaluations

Shady evaluates property dynamics and `Animate()` methods in the following order on each frame:

1. `World.Animate()`
2. `World` dynamic properties
3. Each `Stimulus` (sorted first by \( z \) and second by time of creation):
   a. `Stimulus.Animate()`
   b. `Stimulus` dynamic properties

For each `World` or `Stimulus` instance, the dynamics are evaluated in a fixed order relative to each other. The order may seem arbitrary. It is not recommended to make dynamic properties that use the values of other dynamic properties, thereby relying on an assumption that certain dynamics are evaluated before others in a given frame. If you need to do this, a clearer approach would be to use the `Animate()` method to set the properties procedurally in the order you need them calculated.

5.4.7 Property Sharing

Many attributes of a `Shady.World` or `Shady.Stimulus` instance are what we call “managed properties”. These are stored in arrays (even if they only contain a single value and otherwise behave like scalars). Many of them are transferred to the rendering engine on every frame for drawing. By default, every new `Stimulus` object you create has a fresh new set of managed property arrays created for it, initialized to their default values. However, it is possible to force multiple stimuli to share the memory space for their properties, linking their behavior together at with no extra runtime computational cost. Sharing properties in this way will cause those stimuli to be linked until you explicitly unlink them. This allows your program to be less complex and more CPU-efficient, since you will only have to change the property value of one stimulus and the change will affect all the others.

The simplest way to share a property between two stimuli is to use the shorthand convention of assigning an actual `Stimulus` instance to the relevant property value of another stimulus:

```python
import Shady
world = Shady.World( 700, top=100, frame=True )
stim1 = world.Patch( x=-200, rotation=30 )
stim2 = world.Stimulus( x=+200, rotation=stim1 )
# now, any change to either stimulus's rotation will affect both
stim1.rotation = 45
print( stim2.rotation )  # --> 45
```

This technique is in fact a syntactic shorthand for the `ShareProperties` method of the first stimulus:

```python
stim1.ShareProperties( stim2, 'rotation' )
# same effect as stim1.rotation=stim2
```

This more powerful method can be used to share multiple properties at a time between multiple stimuli:

```python
import Shady
world = Shady.World( 700, top=100, frame=True )
master = world.Stimulus( size=50, y=100 )
followers = [ world.Stimulus( size=50, x=x ) for x in range( -400, 400, 100 ) ]
master.ShareProperties( followers, 'rotation', 'scale' )
```

Note that the name `master` here may be slightly misleading, because sharing is fully symmetric: any change to the `rotation`, or `scale` of any of the `Stimulus` instances in `followers` will affect the remaining contents...
of followers and our master. That said, it may be useful to designate one stimulus as the master to indicate a convention that this stimulus should be used to control the others, especially if you are going to use dynamic property assignment:

```python
master.rotation = lambda t: t * 20
```

`ShareProperties()` also allows you to set property values at the same time as sharing them:

```python
import Shady
world = Shady.World( 700, top=100, frame=True )
stim1 = world.Patch( x=-200 )
stim2 = world.Patch( x=+200 )
# share color and set that shared color to red
stim1.ShareProperties( stim2, color=( 1, 0, 0 ) )
```

If you want a stimulus to stop sharing properties, you can again use the shorthand of `Stimulus`-instance assignment:

```python
stim2.color = stim2    # tell `stim2` to "be yourself"
```

...which is really a shortcut for `stim2.MakePropertiesIndependent( 'color' )`. The `MakePropertiesIndependent` method can also simultaneously change the value(s) of one or more properties as it unlinks them - here's another example:

```python
# ...
stim1.ShareProperties( stim2, position=( -100, 200 ), alpha=0.5, scale=2.5 )
stim1.Set( position=600 )
print( stim2.position )
# --> [ 600, 600 ]

print( ( stim1.scale, stim2.scale ) )
# --> ( 2.5, 2.5 )
stim2.MakePropertiesIndependent( scale=7 )
print( ( stim1.scale, stim2.scale ) )
# --> ( 2.5, 7.0 )
stim2.alpha = 0.3    # still linked
print( ( stim1.alpha, stim2.alpha ) )
# --> ( 0.3, 0.3 )
```

One final warning: property sharing does not work with property index shortcuts, as two stimuli cannot share just part of a full property array. If you want to share specific property dimensions such as `x` or `red`, but not the other dimensions of that property, you should use a dynamic function instead to ensure it is continually updated:

```python
### WRONG ###
# ...
stim2.x = stim1
# --> ValueError: x is the name of a shortcut, not a fully-fledged property - cannot link it across objects

### RIGHT ###
# ...
stim2.x = lambda t: stim1.x    # (although it comes at a small CPU cost)
```

See the demo script `examples/sharing.py` for more.
5.5 Shady API Reference

5.5.1 Shady Package

- The World Class
- The Stimulus Class
- Global Functions and Constants

The World Class

class Shady.World(width=None, height=None, left=None, top=None, screen=None, threaded=True, canvas=False, frame=False, fullScreenMode=None, visible=True, openglContextVersion=None, legacy=None, backend=None, acceleration=None, debugTiming=False, profile=False, syncType=-1, logfile=None, reportVersions=False, window=None, **kwargs)

Bases: Shady.Rendering.LinkGL

A World instance encapsulates the window and rendering environment in which you draw stimuli. By default, a World will fill one screen, but its size, offset and decoration can also be tailored explicitly if necessary. When you initialize a World, Shady creates an OpenGL program and compiles and links a vertex shader and a fragment shader to it: this is what allows signal generation, contrast modulation, windowing, gamma correction and dithering to be performed on the graphics processor.

Once you have created the World, you will probably want to call need to call the Stimulus() method one or more times, to configure the things that should be drawn in it.

Parameters

- **width (int)** – width of drawable area in “screen coordinates” (which usually means pixels, but see the note below).

- **height (int)** – height of drawable area in “screen coordinates” (which usually means pixels, but see the note below).

- **size (int, tuple or list)** – width and height of drawable area in “screen coordinates” (see note below). If this is a single number, it is used for both width and height. If it is a tuple or list, it is interpreted as [width, height]. However, the separate width and/or height arguments take precedence, if supplied.

- **left (int)** – horizontal offset from the edge of the screen, in “screen coordinates” (which usually means pixels, but see the note below).

- **top (int)** – vertical offset from the top of the screen, in “screen coordinates” (which usually means pixels, but see the note below).

- **screen (int)** – Screen number. 0 or None (default) means use whichever is designated as the primary screen. A positive integer explicitly selects a screen number. The output of the global Screens() function may help you choose the screen number you want.

- **threaded (bool)** – If you specify threaded=False, the World’s main rendering/event-processing loop will not be started automatically: you will have to start it yourself, from the appropriate thread, using the Run() method. In this case, the best way to perform initial World configuration and Stimulus creation is to put the code in the body of a Prepare() method that you specify by subclassing World.
With `threaded=True`, a new thread will be created to perform all the work of `World` construction and then, automatically, to run the main loop. Any subsequent call to the `Run()` method will do nothing except sleep until the thread ends. This is the easiest way to use Shady: you can then create and manipulate stimuli either from a `Prepare` method, or from wherever else you want. This appears to work well on Windows, but will have problems (which can only partially be worked-around) on other operating systems: see the Shady.Documentation.Concurrency docstring or click here.

- **canvas (bool)** – If you set this to True a “canvas” `Stimulus` instance will be created automatically, filling the screen behind other stimuli. If you do not create one automatically like this, you can do it later by calling the `MakeCanvas()` method. A canvas allows gamma-correction and dynamic-range enhancement tricks to be performed on the backdrop as well as on your foreground stimuli, and it allows the World’s “atmosphere” properties (`backgroundColor`, `gamma`, `noiseAmplitude` and friends) to take effect: see the Shady.Documentation.PreciseControlOfLuminance docstring or click here.

- **frame (bool)** – Whether or not to draw a frame and title-/drag- bar around the window.

- **fullScreenMode (bool)** – Default behavior is to create a window that exactly covers one screen. This can be done by creating an ordinary window that just happens to be the same size as the screen (fullscreenmode=False) or by actually asking the system to change to full-screen mode (fullscreenmode=True). The default on Windows is False, since it allows you to switch windows (e.g. with alt-Tab) and still have the Shady window visible in the background (note however, that background windows have poor timing precision - to render precisely without skipping frames you will need to keep the window in the foreground). With fullscreenmode=True this is impossible: the window will disappear when it loses focus. On the Mac, the default setting for full-sized windows is fullscreenmode=True, because this seems to be the only way to hide the menu bar at the top of the screen.

For non-full-sized windows, the default is fullscreenmode=False. If you set it to True while also explicitly designating the size of the World, the OS will attempt to change resolution. This will probably be a bad idea for psychophysical applications on most modern screens: for rendering accuracy, you should address the screen at its native (maximum) resolution.

- **visible (bool)** – If you set this to False, the window will be created off-screen and will only become visible when you set its visible property to True.

- **debugTiming (bool)** – Every World records the timing intervals between its frame callbacks, to aid in analyzing timing performance. If you set debugTiming=True, it will record additional information that breaks down the allocation of this time. By default the setting will be propagated to every Stimulus instance as well (set each stim. debugTiming=False if this is not what you want—if there are many stimuli, then the timing debug calls themselves can start to have a measurable impact on performance).

- **logfile (str)** – Optionally specify the name of a text file which will log various pieces of useful diagnostic information. If your filename includes the substring (), this will be replaced by a yyyymmdd-HHMMSS local timestamp. If the file stem ends with ~full then self. logger.logSystemInfoOnClose will be set to True by default which means that a third-party program will be run when the World closes, to record extensive system information (NB: on Windows the program is dxdiag.exe which is time-consuming and produces lengthy logs). You can write to the log file yourself with self.logger.Log()

- **reportVersions (bool)** – If this is True, the World instance will call its ReportVersions() method to report version information to the console, as soon as it is set up.
• **kwargs – Managed property values can also be specified as optional keyword arguments during construction, for example:

\[
\begin{align*}
w &= \text{World}(\ldots, \text{clearColor}=\{0,0,0.5\}, \ldots)
\end{align*}
\]

Note: The easiest way to create a World is by omitting all geometry arguments. Then it will fill the display screen (the primary screen by default, but you can also specify the screen number explicitly). However, if you choose to use explicit geometry arguments (\textit{width, height, size, left, top}) note that they are all in “screen coordinates”. Screen coordinates usually correspond to pixels, but in some systems (Mac with Retina screens) you may need to specify some fixed smaller proportion of the number of addressable pixels you actually want: for example, on a Late-2013 Macbook with 13-inch Retina screen, \(w = \text{World}(1280, 800)\) opens a window that actually has double that number of addressable pixels \((2560 \times 1600)\). After construction, \(w\text{.size}\) will indicate the correct (larger) number of pixels. Unfortunately, \textit{before} construction, I have not yet found a general way of predicting the relationship between screen coordinates and pixels.

\textbf{classmethod AddCustomUniform}(name=None, defaultValue=None, **kwargs)

Modifies the class (World or Stimulus) so that it possesses one or more new managed properties, whose values are then accessible from inside the fragment shader. This must be performed \textit{before} World construction.

Example:

\[
\begin{align*}
\text{Shady.World.AddCustomUniform('spam', [1,2,3])} \\
\text{Shady.Stimulus.AddCustomUniform(eggs=4, beans=[5,6])}
\end{align*}
\]

Either syntax can be used in either class. The keyword-argument syntax has the advantage of being able to define multiple new properties in one call.

The default values you supply dictate whether the new property is 1-, 2-, 3- or 4-dimensional. For a 1-dimensional property, the type of your default value also determines whether the property gets defined as an integer or floating-point variable. (2-, 3- or 4-dimensional properties are always re-cast as floating-point).

The corresponding uniform variables are then automatically made available in the fragment shader code, with the first letter of the property name getting capitalized and a ‘u’ prepended. So, as a consequence of the two lines in the example above, the modified shader would then contain these definitions:

\[
\begin{align*}
\text{uniform vec3 uSpam; } \\
\text{uniform int uEggs; } \\
\text{uniform vec2 uBeans;}
\end{align*}
\]

…all of which is useless unless you actually write some custom shader functions that access the new variables.

\textbf{See also:}

AddCustomSignalFunction, AddCustomModulationFunction, AddCustomWindowingFunction

\textbf{AddForeignStimulus}(stimulus, name=None, z=0, **kwargs)

\textit{stim} is either a class (or callable factory function) or an instance of a class. That class should have a \texttt{draw()} method. It is drawn on every frame with the Shady shader pipeline disabled (so, no automatic linearization or dithering, etc).

The intention is to allow the use of one’s own custom OpenGL primitives in drawing unusual stimuli.

\textbf{AfterClose}(func, *pargs, **kwargs)

This method registers a callable \texttt{func} (and optionally the \*\texttt{pargs} and **\texttt{kwargs} that should be passed to it) that will be called just \textit{after} the World closes. The method is otherwise identical to \texttt{BeforeClose()}. 5.5. Shady API Reference
**AnimationCallback** *(func=None)*  
Decorator version of `SetAnimationCallback()`

Examples:
```
w = Shady.World()
@w.AnimationCallback
def anim( self, t):
    print( t )
```

**BeforeClose** *(func, *pargs, **kwargs)*
This method registers a callable `func` (and optionally the *pargs and **kwargs that should be passed to it) that will be called just before the World closes. One way to use this method is as a decorator—for example:

```
w = Shady.World()
@w.BeforeClose
def Goodbye():
    print( "goodbye" )
```

Returns  (1) when `func` is called, its return value will be inserted into this list; (2) the `id` of the list instance uniquely identifies the callback you have just registered, so it can be used as a handle for cancelling the function with `CancelOnClose()`

Return type  an empty list. This has two uses

**BoundingBox** *(worldCoordinates=False)*
This method returns the World's bounding box, in pixels.

Parameters  `worldCoordinates` *(bool)* – If True, then the left and bottom coordinates are computed relative to the World's own anchor (so that gives you the bounding box within which you can draw stimuli, in coordinates a Stimulus would understand). If False, then left = bottom = 0.

Returns  `[left, bottom], [width, height]` pixel coordinates for the World.

**CancelOnClose** *(container)*
Cancels a callback that had previously been scheduled to run at closing time by either `BeforeClose()` or `AfterClose()`, either of which will have returned the `container` that you need to pass as the input argument here.

**Capture** *(pil=False, fullscreen=False, saveas="", size=None, origin=None, normalize='auto')*
Takes a screenshot of the World and return the RGBA image data as a numpy array (pil=False) or as a PIL.Image.Image instance (pil=True).

Parameters

- **pil** *(bool)* – If True, and PIL or pillow is installed, an Image object is returned. Otherwise, return a numpy array (if numpy is installed) or a buffer containing the raw pixel information (if not).

- **fullscreen** *(bool)* – Normally, with fullscreen=False, we capture just the World content. But if we specify fullscreen=True, and PIL or pillow is installed, and we’re on Windows, then the ImageGrab module will be used to grab a shot of the whole screen as-is (including other windows and desktop content if the World fills only part of the screen or is partly obscured).

- **saveas** *(str)* – If a filename is specified here, and PIL or pillow is installed, then the image is automatically saved under the specified filename.
• **size** *(tuple, list)* – This is a sequence (tuple, list, or 1-D `numpy` array) containing 2 integers: width and height, in pixels. If unspecified (None), the size of the `World` is assumed.

• **origin** *(tuple, list)* – This is a sequence (tuple, list, or 1-D `numpy` array) containing 2 integers: x and y, in pixels, indicating the offset in pixels between the lower left corner of the `World` and the lower left corner of the capture area. If unspecified (None), [0, 0] is assumed.

• **normalize** *(bool or 'auto')* – If False, return raw RGBA values as integers. If True, return floating-point values normalized in the range 0 to 1, and furthermore undo the effects of the current `bitCombiningMode` if any. If 'auto', the default is False except when all the following conditions are met: `numpy` is installed, `pil` is False, and `self.bitCombiningMode` is non-zero.

**Returns** A `PIL.Image.Image` object (with `pil=True`, provided `PIL` or `pillow` is installed) or a numpy array (with `pil=False`) containing 8-bit RGBA pixel values.

**ClearDynamics()**
Remove all property dynamics from the instance.

**See also:** `GetDynamic()`, `GetDynamics()`, `SetDynamic()`

**Close()**
Close the `World`.

**CreatePropertyArray**(propertyName, *stimuli)
This method returns a `PropertyArray` object which contains a numpy array. Each row of the array is the storage area for the named property of one of the specified stimuli. You can still address the property of each individual `Stimulus` in the usual way, but now you also have the option of addressing them all at once in a single array operation, which may be much more efficient if there are many stimuli.

**Parameters**

• **propertyName** *(str)* – The name (or alias) of a fully-fledged `ManagedProperty` of the `Stimulus` class (for example, 'color' or 'position').

• ***stimuli** – This is flexible. You can pass the (string) names of `Stimulus` instances, or you can pass the instances themselves. You can pass them as separate arguments, and/or in tuples or lists. If you’re using names, you can even pass them as a single space-delimited string if you want.

**Returns** a `PropertyArray` instance whose `A` attribute contains the `numpy` array.

**Example:**

```python
import numpy, Shady
w = Shady.World()
p = numpy.linspace(-1.0, +1.0, 20, endpoint=True)
stims = [ w.Stimulus( xy=w.Place( p_i, 0 ), bgalpha=0, pp=1 ) for p_i in p ]
position = w.CreatePropertyArray( 'xy', stims )
color = w.CreatePropertyArray( 'color', stims )
@w.AnimationCallback
def Animate( self, t ):
    s = Shady.Sinusoid( t, p * 180 )
    position.A[ :, 1 ] = 200 * s
    color.A[ :, 0 ] = 0.5 + 0.5 * s
    color.A[ :, 1 ] = 0.5 + 0.5 * s[::-1]
    color.A[ :, 2 ] = s ** 2
    # one call with a few simple efficient numpy array operations
    # instead of 20 stimuli x 4 shortcuts = 80 dynamic function calls
```

---

5.5. Shady API Reference 151
**Culling** *(enable, alphaThreshold=0.25)*

Depth culling is disabled by default. This means that where two stimuli overlap, every pixel is rendered in both stimuli. Depending on your stimulus arrangement, this may be a significant waste of resources.

Turn culling on with `Culling(True)` instead of drawing all pixels of all stimuli in furthest-to-nearest order (the painter’s algorithm), Shady will now draw them in reverse painter’s order, nearest-to-furthest, omitting calculations that would affect already-drawn-to pixels.

Depth culling is fine if all your stimuli are opaque. The disadvantage to depth culling is that it affects alpha blending and composition: when a stimulus with some transparent parts is overlapped on another stimulus, depth culling alone would cause the clearColor of the World to be drawn where there should be transparency. A partial countermeasure to this is alpha culling, which will be turned on concurrently with the depth test unless you explicitly specify a negative alphaThreshold. With alpha culling, any pixel in any stimulus whose alpha is equal to or less than alphaThreshold is simply omitted, and stimuli behind it will show through at that point even with depth culling enabled. The quality of some stimuli (e.g. antialiased text on a transparent background) may be rendered with inaccurate colors: stimulus colors get alpha-blended with the clearColor, not with the colors of the stimuli behind them.

Use `Culling(False)` to disable both depth and alpha culling.

**Defer**(func, *pargs, **kwargs)

Any method that actually makes GL calls, such as `NewPage()` or `LoadTexture()`, must be called from the same thread/context as the one where all other GL operations happen, otherwise you may fail to see any result and/or the program may crash. This is a workaround: if you call (for example) `self.Defer(func, arg1, arg2, ...)` then `func(arg1, arg2, ...)` will be called at the end of the next frame.

This function is already used, under-the-hood, by the @DeferredCall method decorator, to make methods like `World.Stimulus` or `Stimulus.NewPage` safe. So there may be relatively few cases in which you need to use it directly.

**Parameters**

- **func** – this is the callable object to be called at the end of the next frame. Optionally, you may influence the order in which these pending tasks are performed by supplying a tuple `(priority, func)` instead of just `func`. The numeric value `priority` defaults to 0. The higher its value, the earlier the task will be performed, relative to other pending tasks.

- ***pargs** and ****kwargs** – any additional positional arguments, and any keyword arguments, are simply passed through to `func` when it is called.

**Returns** An empty list. When `func` is finally called, its output will be inserted into this list. If `func` causes an exception to be raised, the stack trace information will be wrapped in a `DeferredException` instance and returned in the list. The list also serves as a handle by which you can `Undefer()` a deferred task.

**EventHandler**(slot=0)

Decorator version of `SetEventHandler()`

Examples:

```python
w = Shady.World()

@w.EventHandler # overwrites default slot=0
def handler( self, event):
    print( 'handler got %s event' % event.type )
```

(continues on next page)
@w.EventHandler( slot=1 )  # uses an explicit slot

def handler2( self, event ):
    print( 'handler2 got $s event' % event.type )

GetDynamic(name)
For dynamic properties, return the actual callable object that generates property values, rather than the
current static value.

  Parameters name (str) – Name of the property

  Returns Callable object responsible for generating values for the named property (or None if
there is no such dynamic).

  See also: GetDynamics(), SetDynamic(), ClearDynamics()

GetDynamics()
Get an ordered list of (name, callable) tuples detailing all the dynamics of this instance.

  See also: GetDynamic(), SetDynamic(), ClearDynamics()

HandleEvent(event)
This method is called every time an event happens. Its argument event is a standardized instance of class
Event, containing details of what happened. The default (superclass) implementation responds when
event.type is 'key_release' and the released key is either the Q key or the escape key - this
causes the window to close (and hence the drawing thread will terminate, if this World is threaded).

You can overshadow this method in your World subclass, or you can replace it on an instance-by-instance
basis using the method SetEventHandler() or the decorator EventHandler()—for example:

```python
def handler( self, event ):
    print( event )

w.SetEventHandler( handler )
```

Note that, for each event, it is possible to have more than one handler, occupying different “slots” in the
cascade that is applied to each event. By default, the HandleEvent method occupies slot 0.

Inherit(other)
Set the values of all managed properties of this instance to match those of another instance. Only properties
common to both objects will be considered. Dynamic property values will be copied as dynamic properties.

  Parameters other – instance whose property values will be copied to the properties of self

  Returns self

LinkPropertiesWithMaster(master,*pargs,**kwargs)
See ShareProperties()

  Returns self

LookupTable(*pargs,**kwargs)
Creates a LookupTable instance using LookupTable( world=self, ... )

MakeCanvas(**kwargs)
Create a canvas stimulus that covers the World and thereby allows the World’s backgroundColor,
gamma, ’.ditheringDenominator’ and other “atmosphere” parameters to be put into effect.

This is called automatically if you say canvas=True in your World constructor call.

MakePropertiesIndependent(*pargs,**kwargs)
Undoes the effect of ShareProperties(). Optionally, set the values of the properties after unsharing.
Example:

```python
a.ShareProperties(b, c, d, alpha=1, beta=2, gamma=3)
b.MakePropertiesIndependent(alpha=4)
# Now `a` and `b` share properties `.beta` and `.gamma`, but
# `.b.alpha` is independent and already has the new value 4;
# by contrast `a`, `c` and `d` still share all three properties.

c.MakePropertiesIndependent('beta', 'gamma')
# Now `c` does not share anything except `.alpha`, although the
# property values themselves have not yet been changed.

c.Set(beta=c, gamma=c)
# a syntactic shorthand for the same operation as in the previous
# line
```

Returns self

Patch(**kwargs)
A convenience wrapper around the Stimulus() method, for creating patches of solid color (supply pp=1 for oval, pp=-1 for rectangular).

Place(xp, yp=None, worldCoordinates=True, polar=False)
Convert 2-D normalized coordinates (relative to the instance, -1 to +1 in each dimension), into 2-D pixel coordinates, either relative to the World's current anchor, or relative to the World’s bottom left corner irrespective of anchor.

Input coordinates may be given as one scalar argument, two scalar arguments, or one argument that is a sequence of two numbers. Depending on the polar argument, these will be interpreted as x, y Cartesian coordinates (where, if y omitted, it defaults to y=x) or theta, r polar coordinates (where, if r is omitted, it defaults to r=1).

Parameters

- worldCoordinates (bool) – If True, return pixel coordinates relative to the World’s own anchor position. If False, return pixel coordinates relative to the World’s bottom left corner irrespective of its anchor.
- polar (bool) – If True, input coordinates are interpreted as an angle (in degrees) and an optional radius (0 denoting the center, 1 denoting the edge).

Examples:

```python
instance.Place([-1, 1])  # top left corner
instance.Place(-1, 1)    # likewise, top left corner
instance.Place(90, polar=True)  # middle of top edge (radius 1 assumed)
instance.Place([90, 0.5], polar=True)  # halfway between center and top
instance.Place([90, 0.5, polar=True])  # likewise, halfway between center and top
```

Prepare(**kwargs)
This method is called after the World is initialized and the shader programs are set up, but before the first frame is actually rendered. It is carried out in the same thread as initialization and rendering.

In the World base class, this method does nothing. However, if you create a subclass of World, you can overshadow this method. It is then a good place to perform initial Stimulus creation and other configuration specific to your particular application. You can use any prototype you like for the method, as long as it has a self argument.
When you first construct the `World`, any keyword arguments that are not recognized by the constructor will be automatically passed through to `Prepare()`.

### classmethod Properties

#### Parameters
- **includeShortcuts** – Determines whether `ManagedShortcut` instances should be included (True) or not (False; default) along with `ManagedProperty` instances.

#### Returns
A list of `ManagedProperty` (and optionally also `ManagedShortcut`) descriptor instances.

### ReportVersions

This method calls the global function `ReportVersions` with the `World` instance as its first argument.

### Run

If the `World` was created with `threaded=True` and is already rendering, then this method does nothing except sleep in the current thread until the `World` has finished. If the `World` was not created threaded, then this method is required to actually execute its main rendering loop. Either way, the method does not return until the `World` closes.

### RunDeferred

#### Parameters
- `func`, `*pargs`, `**kwargs` – Defer() a function, then wait for it to run, and then return the output. NB: you’ll wait forever if the `World` is not running...

#### Note
Note that you do not need to do this for most `World` methods: they have already been wrapped so that they run deferred when necessary and appropriate.

### Set

Set the values of multiple managed properties in one call. An error will be raised if you try to set the value of a non-existent, or non-managed, property.

#### Returns
self

**Example:**

```python
instance.Set( rotation=90, color=(0, 0, 1), contrast=0.5 )
```

### SetAnimationCallback

#### Parameters
- `callback` – Bind the callable object `callback` as the instance’s animation callback. Each object, whether it is a `World` or a `Stimulus`, may optionally have a single animation callback which is called on every frame.

#### Note
The animation callback is installed as the attribute `self.Animate`. By default, this attribute is `None`.

#### Example

```python
def animate( self, t ):
    print( t )
my_world = Shady.World().SetAnimationCallback( animate )
```

There is also a decorator version of the same operation, simply called `AnimationCallback()`:

```python
stim = my_world.Stimulus()
@stim.AnimationCallback
def Animate( self, t ):
    print( t )
```
**SetBitCombiningMode** *(mode, verticalGrouping='iso')*

This method supports high-dynamic-range rendering on specialized vision-science hardware—see documentation for the `bitCombiningMode` property for full details. Calling `w.SetBitCombiningMode(mode)` is the same as simply assigning `w.bitCombiningMode = mode`. The only difference is that, when `mode=2` (sacrificing horizontal resolution to achieve 16-bit-per-channel full-color rendering), the method allows you to specify separately whether vertical resolution should also be sacrificed. Shady’s default behavior is to throw away vertical resolution at the same time, so that logical pixels remain physically square, as follows:

```
w.SetBitCombiningMode( mode=2, verticalGrouping=2 )
```

However you can override this, changing the aspect ratio of your pixels and retaining full vertical resolution, as follows:

```
w.SetBitCombiningMode( mode=2, verticalGrouping=1 )
```

classmethod `SetDefault` (**kwargs)**

Affects the class. Sets the default values of named properties. In instances of the class created subsequently, managed properties will have the new default values.

Example:

```
cls.SetDefault( color=(1,0,0), rotation=90 )
```

**SetDynamic** *(name, func, order=-1, canonicalized=False)*

Associate a “dynamic” (i.e. a function that can be called repeatedly to set an attribute) with the name of an attribute.

For example:

```
foo.Set( 'bar', lambda t: t ** 2 )
```

This will set `foo.bar` to `t ** 2` every time the method `foo._RunDynamics(t)` is called (this call will happen automatically from the API user’s point of view, in the object’s infrastructure implementation).

A dynamic can be attached to any attribute name. If the `bar` attribute happens to be a `ManagedProperty` or `ManagedShortcut`, then it will also automatically support “dynamic value assignment”, i.e. you can do:

```
foo.bar = lambda t: t ** 2
```

as a syntactic shorthand for `SetDynamic()`—the setter will detect the fact that the value is callable, and divert it to the register of dynamics rather than assigning it directly (so the actual static value you get from querying `foo.bar` will not immediately change).

**Parameters**

- **name** *(str)* — name of the attribute
- **func** — callable object, or None to remove any dynamic that may already be associated with name
- **order** *(int, float)* — optional numeric ordering key that allows you to control the serial order in which dynamics are evaluated
- **canonicalized** *(bool)* — for internal use only. `ManagedProperty` and `ManagedShortcut` descriptors can have multiple aliases. The default settings, `canonicalized=False` says that name has not necessarily been translated to the descriptor’s canonical name, so this should be attempted
Returns self

See also: `GetDynamic()`, `GetDynamics()`, `ClearDynamics()`

**SetEventHandler** *(handler, slot=0)*

Bind the callable handler as part of the instance’s cascade of event handlers. The slot argument determines the serial order in which handlers are run on each event (negative numbers before positive). If handler is `None`, any handler currently occupying the specified slot is removed.

By default, the class’s `HandleEvent()` method is registered in slot 0.

The prototype for an event handler can be `handler(self, event)` or just `handler(event)`. If the handler returns anything that evaluates to `True` in the boolean sense, that’s the signal to abort the cascade of handlers for the event in question (i.e., skip the handlers in higher-numbered slots, this time around).

Example:

```python
def handler( self, event ):
    print( event )

w = Shady.World().SetEventHandler( handler, slot=-1 )
```

There is also a decorator version of the same operation, simply called `EventHandler()`:

```python
w = Shady.World()
@w.EventHandler( slot=-1 )
def handler( self, event ):
    print( event )
```

**SetLUT** *(value)*

Sets or unsets a `LookupTable` for a `World` or `Stimulus`. (For `World`’s, this will only be effective to the extent that the `World`’s `atmosphere` properties are shared with `Stimulus` instances—for example, the canvas.)

Calling this method is the functional equivalent of setting the `lut` property:

```python
stim.SetLUT( value )  # These are equivalent
stim.lut = value  
```

Setting a look-up table disables automatic linearization via `gamma` and automatic dynamic-range enhancement via `ditheringDenominator`, and allows you to take direct control of these issues (although only for one dimension of luminance per pixel: using a look-up table is a form of indexed-color rendering).

See the `Shady.Documentation.PreciseControlOfLuminance` docstring or `Gamma Correction, Dynamic Range Enhancement, and the Canvas` for more details.

**Parameters** `value` *(None, str, list, numpy.ndarray, LookupTable)* — A pre-existing `LookupTable` instance may be used here. Alternatively, any valid constructor argument for a `LookupTable` may be used, and the instance will be constructed implicitly. That means you can use:

- a `numpy.ndarray` of integers in n-by-3 or m-by-n-by-3 arrangement, or
- a list of lists that can be implicitly converted into such an array by `numpy`, or
- the filename of a `npy` or `npz` file that contains such an array (under the variable name `lut` in the latter case) to be loaded by `Shady.Linearization.LoadLUT()`, or
- the filename of a `png` file containing the look-up table entries as RGB pixel values in column-first order, again to be loaded by `Shady.Linearization.LoadLUT()`.

Finally you have the option of setting `None`, to disable the usage of look-up tables.
Returns A LookupTable instance.

SetSwapInterval (value)
By default, Shady attempts to update the screen on every physical frame the display hardware can produce. This corresponds to a “swap interval” of 1 and on a typical modern LCD display this usually means 60 frames per second. If you set the swap interval to 2, updates will happen on every second frame (hence typically 30 fps) allowing you more time to perform computations between frames.

To synchronize frames with the hardware at all, Shady relies on the “vertical synch” option which may have to be enabled explicitly in the control panel of your graphics card driver. Without synchronization, you may observe “tearing” artifacts.

The ability to change the swap interval is hit-and-miss, and back-end- and platform-dependent. This call may have no effect.

ShareProperties (*pargs, **kwargs)
Share the underlying array storage of the named managed properties with other objects, so that a change to one affects all. Optionally, set the values at the same time. Calling syntax is flexible - the following examples all take certain property arrays associated with instance a and share them with instances b, c and d:

```python
a.ShareProperties( 'gamma noiseAmplitude backgroundColor', b, c, d )
```

A similar effect can be obtained via b.LinkPropertiesWithMaster( a, ... )—although there you are limited to one sharee at a time—and a syntactic shorthand for this is to pretend you are assigning the master object instance itself to the target property:

```python
b.gamma = a
```

Undo with MakePropertiesIndependent() (or b.gamma = b)

Returns self

Sine(**kwargs)
A convenience wrapper around the Stimulus() method, for creating a functionally-generated sine-wave patch, with the linearization and dynamic-range enhancement parameters (“atmosphere” parameters) yoked to those of the World by default.

Stimulus(*pargs, **kwargs)
Creates a Stimulus instance using Stimulus( world=self, ... )

Tick()
Wait until after the next frame has been rendered. (Will have no effect, and return immediately, if your are calling this from the same thread in which your World runs.)

Undefer(target_container)
Cancels a pending task that has been scheduled by Defer() `to run at the end of the next frame. Its input argument is the output argument of `Defer().

Wait()
Wait until after the next frame has been rendered. (Will have no effect, and return immediately, if your are calling this from the same thread in which your World runs.)

WaitFor(condition)
This function blocks the current thread until a specified condition becomes True. The condition will be checked repeatedly in between 1-millisecond sleeps. It can be one of two things:

• a callable (function) which returns True to signal that the wait is over - e.g.
stim.WaitFor(lambda: stim.y < 0)

• a string that names a dynamic attribute belonging to the current instance. In this case, the wait ends when there is no longer a dynamic attached the the property or shortcut in question. A dynamic may be removed due to explicit action of another thread in your program, or explicit action in this instance’s AnimationCallback (or indeed any instance’s AnimationCallback). Alternatively, a dynamic may automatically remove itself, by raising a StopIteration exception (the Function object returned by Shady.Dynamics.Transition is an example of something that does this):

stim.scaling = Shady.Transition(stim.scaling, 2, duration=5)
# the stimulus will now double in size over the course of 5 sec
stim.WaitFor('scaling')
# wait until it's finished

All of this assumes that you are operating in a different thread from the World—if you call WaitFor from inside one of the World or Stimulus callbacks, then it will simply sleep indefinitely because nothing will get the chance to change.

anchor
This is a managed property. It is a pair of numbers specifying where, in normalized coordinates within the rendering area of the window, pixel coordinate (0,0) should be considered to be for Stimulus positioning. An origin of [-1,-1] means the bottom left corner; [0,0] means the center; [+1,+1] means the top right corner. Translations resulting from a change in anchor are automatically rounded down to an integer number of pixels, to avoid anchor becoming an unexpected source of interpolation artifacts.

• Default value: [0.0, 0.0]
• Canonical name: anchor
• Other aliases: origin

• Subscripting shortcuts:
  – ax_n is a shortcut for anchor[0]
  – ay_n is a shortcut for anchor[1]

anchor_x
anchor_x is an alias for ax_n

anchor_y
anchor_y is an alias for ay_n

atmosphere
This property actually encompasses multiple managed properties, all related to linearization and dynamic-range enhancement.

If you query this property, you will get a dict of the relevant property names and values. You can also assign such a dictionary to it.

More usefully, you can use it to link all the properties between instances in one go:

stim.atmosphere = world  # hard-links all the "atmosphere" properties # at once

For more information, see the Shady.Documentation.PreciseControlOfLuminance docstring or click here.

ax_n
This is a managed shortcut. It is a shortcut for anchor[0]
• Canonical name: ax_n
• Other aliases: anchor_x origin_x ox_n

ay_n
This is a managed shortcut. It is a shortcut for anchor[1]
• Canonical name: ay_n
• Other aliases: anchor_y origin_y oy_n

backgroundColor
This is a managed property. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. It specifies the backgroundColor of the background canvas Stimulus, if present (the World needs to be constructed with the canvas=True, or you otherwise need to call MakeCanvas()). When a canvas is created, this property of the World is automatically linked to the corresponding property of the canvas Stimulus instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with ShareProperties() or LinkPropertiesWithMaster().

• Default value: [0.5, 0.5, 0.5]
• Canonical name: backgroundColor
• Other aliases: bg bgcolor

• Subscripting shortcuts:
  – bgred is a shortcut for backgroundColor[0]
  – bggreen is a shortcut for backgroundColor[1]
  – bgblue is a shortcut for backgroundColor[2]

bg
bg is an alias for backgroundColor

bgblue
This is a managed shortcut. It is a shortcut for backgroundColor[2]

bgcolor
bgcolor is an alias for backgroundColor

bggreen
This is a managed shortcut. It is a shortcut for backgroundColor[1]

bgred
This is a managed shortcut. It is a shortcut for backgroundColor[0]

bitCombiningMode
This non-managed property allows high-dynamic-range rendering on specialized vision-science hardware. It can be set to:

0, or equivalently 'C24': denotes standard 24-bit full-color mode (8 bits per channel, with dithering by default).

1, or equivalently 'M16' or 'monoPlusPlus': denotes 16-bit monochrome reinterpretation of frame-buffer contents (each pixel’s red channel is the more-significant byte, and its green channel is the less-significant byte, of a 16 bit value). As with look-up tables, Shady reads just the red channel to determine the monochrome target value. Dithering is disabled.

2, or equivalently 'C48' or 'colorPlusPlus': denotes 48-bit color mode in which horizontal resolution is sacrificed. Dithering is disabled. Each pixel in the frame buffer is paired with its horizontal neighbor, and together they specify a 16-bit-per-channel value for the corresponding yoked
pair of physical pixels. Shady also throws away vertical resolution at the same time, so that you can continue working with square pixels and sane geometry—if you do not want this, then the `SetBitCombiningMode()` method allows more control.

Values 3 and 4 are also reserved for debugging C48 mode (they will not generate correct C48 stimuli).

**blue**
This is a managed shortcut. It is a shortcut for `clearColor[2]`

**bluegamma**
This is a managed shortcut. It is a shortcut for `gamma[2]`

**buenoise**
This is a managed shortcut. It is a shortcut for `noiseAmplitude[2]`

**clearColor**
This is a managed property. It is a triplet of numbers in the range 0 to 1. It specifies the color of the empty screen. Note that these values are never linearized or dithered. For more precise control over the background, construct your `World` with the argument `canvas=True` and then you can manipulate `backgroundColor`, `gamma`, `.ditheringDenominator` and `noiseAmplitude`.

- Default value: varies according to `BackEnd()` settings
- Subscripting shortcuts:
  - `red` is a shortcut for `clearColor[0]`
  - `green` is a shortcut for `clearColor[1]`
  - `blue` is a shortcut for `clearColor[2]`

**dd**
`dd` is an alias for `ditheringDenominator`

**ditheringDenominator**
This is a managed property. It is a floating-point number. It should be 0 or negative to disable dithering, or otherwise equal to the maximum DAC value (255 for most video cards). It specifies the `ditheringDenominator` for the background canvas Stimulus, if present (the `World` needs to be constructed with the argument `canvas=True`, or you otherwise need to call `MakeCanvas()`). When a canvas is created, this property of the `World` is automatically linked to the corresponding property of the canvas `Stimulus` instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with `ShareProperties()` or `LinkPropertiesWithMaster()`.

- Default value: dithering enabled (value determined automatically)
- Canonical name: `ditheringDenominator`
- Other aliases: `dd`

**fakeFrameRate**
By default, with `fakeFrameRate` equal to `None`, a `World` will try to update itself at the frame rate of the display hardware (though of course it may end up being slower if you are doing too much computation between frames, or if there are other tasks that are using too many CPU or GPU resources). The time argument `t` that gets passed into animation callbacks and dynamic property evaluations will reflect the real wall-time at which each frame is drawn.

However, if you set `fakeFrameRate` to a positive value, now frames can take as long as they like in real time, and the `t` argument will reflect the theoretical amount of time passed based on the number of frames completed, assuming the specified frame rate. This allows you to run animations in slower-than-real time. One of the main applications would be for capturing stills or movies of `World` animation (the capture operation itself tends to be slow).
If you do not fully understand the explanation above, do not change this property. It should be left as None whenever you want to display time-accurate animation (which should be the case under nearly all circumstances).

**gamma**

This is a *managed property*. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. It specifies the *gamma* of the background canvas Stimulus, if present (the World needs to be constructed with the canvas=True, or you otherwise need to call MakeCanvas()). When a canvas is created, this property of the World is automatically linked to the corresponding property of the canvas Stimulus instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with ShareProperties() or LinkPropertiesWithMaster().

* Default value: [1.0, 1.0, 1.0]
* **Subscripting shortcuts:**
  - redgamma is a shortcut for gamma[0]
  - greengamma is a shortcut for gamma[1]
  - bluegamma is a shortcut for gamma[2]

**green**

This is a *managed shortcut*. It is a shortcut for clearColor[1]

**greengamma**

This is a *managed shortcut*. It is a shortcut for gamma[1]

**greennoise**

This is a *managed shortcut*. It is a shortcut for noiseAmplitude[1]

**height**

This is a *managed shortcut*. It is a shortcut for size[1]

**lut**

The value of this property will either be None, or an instance of LookupTable. Assigning to this property is equivalent to calling the SetLUT() method—so you can assign any of the valid argument types accepted by that function. Assigning None disables look-up.

**noise**

*noise* is an alias for *noiseAmplitude*

**noiseAmplitude**

This is a *managed property*. It is a triplet of floating-point numbers corresponding to red, green and blue channels. It specifies the noiseAmplitude for the background canvas Stimulus, if present (the World needs to be constructed with the canvas=True, or you otherwise need to call MakeCanvas()). When a canvas is created, this property of the World is automatically linked to the corresponding property of the canvas Stimulus instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with ShareProperties() or LinkPropertiesWithMaster().

* Default value: [0.0, 0.0, 0.0]
* Canonical name: noiseAmplitude
* Other aliases: noise
* **Subscripting shortcuts:**
  - rednoise is a shortcut for noiseAmplitude[0]
– **greennoise** is a shortcut for **noiseAmplitude**[1]
– **bluenoise** is a shortcut for **noiseAmplitude**[2]

**on**

* on is an alias for visible

**origin**

* origin is an alias for anchor

**origin_x**

* origin_x is an alias for ax_n

**origin_y**

* origin_y is an alias for ay_n

### outOfRangeAlpha

This is a managed property. It is a number in the range 0 to 1. It specifies the outOfRangeAlpha for the background canvas Stimulus, if present (the World needs to be constructed with the canvas=True, or you otherwise need to call MakeCanvas() ). When a canvas is created, this property of the World is automatically linked to the corresponding property of the canvas Stimulus instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with ShareProperties() or LinkPropertiesWithMaster().

* Default value: 1.0

### outOfRangeColor

This is a managed property. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. It specifies the outOfRangeColor for the background canvas Stimulus, if present (the World needs to be constructed with the canvas=True, or you otherwise need to call MakeCanvas() ). When a canvas is created, this property of the World is automatically linked to the corresponding property of the canvas Stimulus instance; if there is no canvas then this property is unused, although you may wish to link it explicitly to other stimuli with ShareProperties() or LinkPropertiesWithMaster().

* Default value: [1.0, 0.0, 1.0]

#### ox_n

* ox_n is an alias for ax_n

#### oy_n

* oy_n is an alias for ay_n

### pixelGrouping

Explicit rounding, in the shader, to the nearest screen pixel location, is disabled by default. However you can see the tiny rounding errors this creates if you create, for example, a signal function that hits exactly 0 at multiple places on top of a background value of 0.5, such as:

```python
Shady.AddCustomSignalFunction(''
    float Tartan(vec2 p) { p = mod(p, 2.0); return 0.25 * (p.x - p.y); }
''
)w=Shady.World(fullScreenMode=0,bitCombiningMode=1);s=w.Stimulus(sigfunc=Shady.˓
SIGFUNC.Tartan)w.pixelGrouping = 0 with w.pixelGrouping = 1 but be aware that (a) the differences are tiny, rounding to values 1/65535th apart and only visible because of the red-green split, (b) pixel rounding is not computationally trivial on the GPU - it requires a per-fragment matrix multiplication. You can change the default with Shady.World.SetDefault( pixelGrouping=1 ) - then this
```
setting will be respected automatically whenever you change to `w.bitCombiningMode=0` or `w.bitCombiningMode=1`.

**Type**  Developer note on pixelGrouping

red

This is a managed shortcut. It is a shortcut for `clearColor[0]`

redgamma

This is a managed shortcut. It is a shortcut for `gamma[0]`

rednoise

This is a managed shortcut. It is a shortcut for `noiseAmplitude[0]`

size

This is a managed property. It is a pair of integers denoting the width and height of the World in pixels. Do not attempt to change these values - it will not alter the size of the window and may have unexpected side effects.

- Default value: `[-1, -1]`
- Subscripting shortcuts:
  - `width` is a shortcut for `size[0]`
  - `height` is a shortcut for `size[1]`

**t**

t is an alias for `timeInSeconds`

timeInSeconds

This is a managed property. It is a floating-point scalar value indicating the time in seconds since the World started rendering.

- Default value: `0.0`
- Canonical name: `timeInSeconds`
- Other aliases: t

visible

This is a managed property. It is a scalar boolean value indicating whether or not the World should be visible or not. On Windows the transition from visible to invisible and back is reasonably instantaneous. On the Mac you may have to endure the minimize/restore animation (although this is backend-dependent).

- Default value: `1`
- Canonical name: `visible`
- Other aliases: on

width

This is a managed shortcut. It is a shortcut for `size[0]`

The **Stimulus Class**

class Shady.Stimulus:

A Stimulus is an entity that is drawn automatically within its parent World on every frame. It has a number of properties whose values govern its appearance independent of other stimuli.

Parameters
• **world** (*World*) – The required first argument is a *World* instance. To make this more readable you can alternatively call the *Stimulus* constructor as a *World* method:

```python
w = World(...)  
s = w.Stimulus(source, ...)  
```

• **source** (*str, list, numpy.ndarray, None*) – The second argument is the *source* of the carrier texture. This may be omitted (or equivalently set to *None*) if your stimulus is just a blank patch, or if its carrier signal is defined purely by a function in the shader (see the *signalFunction* property). Alternatively *source* may be a string denoting an image filename or glob pattern for image filenames, a *numpy* array specifying texture data explicitly, or a list of strings and/or *numpy* arrays - see *LoadTexture()* for details.

• **name** (*str*) – This is a string that will identify this *Stimulus* in the container *w.stimuli* of the *World* to which it belongs. To ensure uniqueness of the names in this dict, you may include a single numeric printf-style pattern in the name (the default *name*, for example, is 'stim%02d').

• **page** (*int*) – You may optionally initialize the *page* property here.

• **multipage** (*bool*) – If you set this to *True*, multiple image frames will loaded using the *LoadPages()* method: this transfers each frame to the graphics card as a separate texture, which you switch between using the *page* property.

By contrast, if you leave it at the default value *False*, multiple image frames are handled by concatenating them horizontally into a single texture, and you switch between them using the *frame* property, which indirectly manipulates *carrierTranslation*.

You may need to use *multipage=True* for animated images that have a large width and/or high number of frames, because the normal concatenation method may lead to the texture exceeding the maximum allowable width.

• ****kwargs** – Managed property values can also be specified as optional keyword arguments during construction—for example:

```python
s = w.Stimulus(..., foregroundColor=[1, 0, 0], ...)  
```

The *width*, *height* and/or *size* arguments (all shortcuts/aliases into the *envelopeSize* managed property) are worth mentioning specifically: these can be used to specify the dimensions of the envelope in pixels. For texture stimuli these can usually be omitted, since usually they will be dictated by the dimensions of the underlying *source* material. However, they can be specified explicitly if you want to crop or repeat an image texture. Note that you should specify dimensions in the same units as the underlying texture (i.e. *unscaled* pixels). If you want to change the physical size of the rendered stimulus by magnifying or shrinking the texture, manipulate the *envelopeScaling* property either directly, or indirectly via *scaledSize* (but remember this will produce interpolation artifacts).

**classmethod AddCustomUniform**( *name=None, defaultValue=None, **kwargs* )

Modifies the class (*World* or *Stimulus*) so that it possesses one or more new managed properties, whose values are then accessible from inside the fragment shader. This must be performed before *World* construction.

Example:

```python
Shady.World.AddCustomUniform( 'spam', [1,2,3] )  
Shady.Stimulus.AddCustomUniform( eggs=4, beans=[5,6] )  
```
Either syntax can be used in either class. The keyword-argument syntax has the advantage of being able to define multiple new properties in one call.

The default values you supply dictate whether the new property is 1-, 2-, 3- or 4-dimensional. For a 1-dimensional property, the type of your default value also determines whether the property gets defined as an integer or floating-point variable. (2-, 3- or 4-dimensional properties are always re-cast as floating-point).

The corresponding uniform variables are then automatically made available in the fragment shader code, with the first letter of the property name getting capitalized and a ‘u’ prepended. So, as a consequence of the two lines in the example above, the modified shader would then contain these definitions:

```glsl
uniform vec3 uSpam;
uniform int uEggs;
uniform vec2 uBeans;
```

...all of which is useless unless you actually write some custom shader functions that access the new variables.

See also:

- `AddCustomSignalFunction`
- `AddCustomModulationFunction`
- `AddCustomWindowingFunction`

**AnimationCallback** *(func=None)*

Decorator version of `SetAnimationCallback()`

Examples:

```python
w = Shady.World()

@w.AnimationCallback
def anim( self, t ):
    print( t )
```

**BoundingBox** *(worldCoordinates=False)*

This method returns the bounding box of the `Stimulus`, in pixels.

- **Parameters**
  - `worldCoordinates (bool)` – If True, then the left and bottom coordinates are computed relative to the `World`'s anchor. If False, then [0, 0] is considered to be the bottom-left corner of the `World`, regardless of its anchor.

- **Returns**
  - [left, bottom], [width, height] pixel coordinates for the `Stimulus`.

**Known Issues:** The method takes account of scaling (due to `envelopeScaling`) and translation (due to `envelopeOrigin, envelopeTranslation, Stimulus.anchor` and `World.anchor`). But it does not take account of `envelopeRotation`.

**Capture** *(pil=False, saveas='', normalize='auto')*

This captures the pixel data from a particular `Stimulus`. Note that it accomplishes this simply by calling `World.Capture()` with the appropriate bounding box. Therefore, two caveats: (1) if other stimuli overlap this one, the capture will contain image data composited from all of them; (2) the bounding-box method is smart enough to compensate for `envelopeTranslation` and `envelopeScaling`, but not `envelopeRotation`.

- **Parameters**
  - `pil (bool)` – If True, and if PIL or pillow is installed, return the image data as a PIL. Image.Image instance. If False, return the data as a numpy array.
  - `saveas (str)` – If PIL or pillow is installed, you can use this argument to specify an optional filename for immediately saving the image data.
• **normalize** *(bool or ‘auto’)* – If False, return raw RGBA values as integers. If True, return floating-point values normalized in the range 0 to 1, and furthermore undo the effects of the current `bitCombiningMode` if any. If ‘auto’, the default is False except when all the following conditions are met: numpy is installed, pil is False, and self.bitCombiningMode is non-zero.

**Returns** Either a numpy.ndarray or a PIL.Image.Image instance, depending on the pil argument.

**ClearDynamics**()
Remove all property dynamics from the instance.

See also: `GetDynamic()`, `GetDynamics()`, `SetDynamic()`

**Enter(****props**)
If a Stimulus instance has previously left the stage with `Leave()`, the `Enter()` method allows it to come back.

You may simultaneously change its properties, using keyword arguments.

**GetDynamic**(*name*)
For dynamic properties, return the actual callable object that generates property values, rather than the current static value.

**Parameters** name *(str)* – Name of the property

**Returns** Callable object responsible for generating values for the named property (or None if there is no such dynamic).

See also: `GetDynamics()`, `SetDynamic()`, `ClearDynamics()`

**GetDynamics**()
Get an ordered list of (name, callable) tuples detailing all the dynamics of this instance.

See also: `GetDynamic()`, `SetDynamic()`, `ClearDynamics()`

**Inherit**(*other*)
Set the values of all managed properties of this instance to match those of another instance. Only properties common to both objects will be considered. Dynamic property values will be copied as dynamic properties.

**Parameters** other – instance whose property values will be copied to the properties of self

**Returns** self

**Leave**(deferAfterAdditionalFrames=0)
If a Stimulus instance stim is made invisible with stim.visible=False, it is not rendered on screen. However, it is still a member of the stimuli dictionary of the World to which it belongs, and its AnimationCallback (if any), and any dynamics attached to its individual properties, are still evaluated on every frame.

On the other hand, you tell a Stimulus instance to `Leave()` the stage entirely, it is removed from the stimuli dictionary of its World, it is not rendered (regardless of its visible setting), and none of its callbacks and dynamics are called—not until you tell it to `Enter()` again.

**Returns** the Stimulus instance itself.

**LinkPropertiesWithMaster**(*master, *pargs, **kwargs*)
See `ShareProperties()`

**Returns** self

**LinkTextureWithMaster**(*master*)
This is a wrapper around `LinkPropertiesWithMaster()` that allows Stimulus instances to share a texture—it shares the textureID, textureSlotNumber and useTexture managed properties.
**LoadPages** (*sources*, *keys=*, *updateEnvelopeSize=*, *page=*, **kwargs*)

This method prepares a *Stimulus* instance for animation using the page mechanism (rather than the default frame mechanism). It loads multiple textures in multiple pages, indexed by the specified keys (or by integers starting at keys, if keys is an integer). Subsequently, you can switch between pages by setting the page property or equivalently calling the SwitchTo() method.

This method is called when constructing a *Stimulus* instance with the multipage constructor option. It can also be called explicitly, which is especially useful if you want to re-use texture buffers that have already been allocated on the graphics card, for new stimulus content.

**LoadSubTexture** (*source*, *x=*, *y=* )

This is similar to LoadTexture() in its interpretation of the source argument. The difference is that the *Stimulus* must already have an existing texture, and the new source is pasted over the old one (or over part of it, depending on the new source’s size).

*x* and *y* are pixel coordinates relative to the existing texture’s lower-left corner. They specify the position of the lower-left corner of the incoming piece of the texture.

**LoadTexture** (*source*, *updateEnvelopeSize=*, *useTexture=*)

Loads texture data from *source* and associates it with this Stimulus.

**Parameters**

- **source** – the source of the carrier texture data. This may be:
  - omitted or set to None if there is no texture (just a constant carrier signal, or one defined by a function in the shader)
  - a string (possibly including glob characters '*' and/or '?') denoting one or more image files to be used as animation frames
  - a numpy array specifying the pixel values of a texture image, in which case:
    - **source.dtype** must be one of:
      - `numpy.dtype('uint8')`: 8-bit pixel values in the range 0 to 255
      - `numpy.dtype('float32')`: pixel value in the range 0.0 to 1.0
      - `numpy.dtype('float64')`: pixel value in the range 0.0 to 1.0 (will be converted to float32 automatically)
    - **source.shape** must be one of:
      - [height, width]: LUMINANCE image
      - [height, width, 1]: LUMINANCE image
      - [height, width, 2]: LUMINANCE_ALPHA image
      - [height, width, 3]: RGB image
      - [height, width, 4]: RGBA image
  - a list or tuple containing filenames and/or numpy arrays as above, to be used as multiple frames

- **updateEnvelopeSize** (*bool*) – whether or not to update the envelope size to match the dimensions of the new texture

- **useTexture** (*bool*) – new value for the useTexture property attribute that enables or disables the use of this texture

**MakePropertiesIndependent** (*pargs*, **kwargs*)

Undoes the effect of ShareProperties(). Optionally, set the values of the properties after unsharing.
Example:

```python
a.ShareProperties(b, c, d, alpha=1, beta=2, gamma=3)
b.MakePropertiesIndependent(alpha=4)
# Now 'a' and 'b' share properties '.beta' and '.gamma', but
# 'b.alpha' is independent and already has the new value 4;
# by contrast 'a', 'c' and 'd' still share all three properties.
c.MakePropertiesIndependent('beta', 'gamma')
# Now 'c' does not share anything except '.alpha', although the
# property values themselves have not yet been changed.
c.Set(beta=c, gamma=c)
# a syntactic shorthand for the same operation as in the previous
# line
```

Returns `self`

**NewPage** *(source, key=None, updateEnvelopeSize=True, **kwargs)*

Create a new page from the specified `source`. A page is a bundle of properties that determine texture, envelope size, and (via `drawMode` and `points`) envelope shape.

Note that this can be automated, to load multiple sources as multiple pages, either by specifying `multipage=True` when you construct the `Stimulus` instance, or by explicitly calling `LoadPages()`.

**Parameters**

- `source` – Any valid input to `LoadTexture()`, including `None`.
- `key` – By default, a new page is unlabelled, but if you specify a `key` here, `SavePage(key)` will be called automatically to store and label the new texture settings. This will enable you to `SwitchTo()` this page in future.
- `**kwargs` – Additional keyword arguments can be used to set property values at the same time, if you want. (NB: you can set any property at all this way, but remember that not all properties get paged in and out—a part from the ones whose values get inferred from `source` automatically, the others that are most meaningful here are `drawMode` and `points`)

See also: `page`, `SavePage()`, `SwitchTo()` and `LoadPages()`

**Place** *(xp, yp=None, worldCoordinates=True, polar=False)*

Convert 2-D normalized coordinates (relative to the instance, -1 to +1 in each dimension), into 2-D pixel coordinates, either relative to the `World`'s current `anchor`, or relative to the `World`'s bottom left corner irrespective of `anchor`.

Input coordinates may be given as one scalar argument, two scalar arguments, or one argument that is a sequence of two numbers. Depending on the `polar` argument, these will be interpreted as `x`, `y` Cartesian coordinates (where, if `y` omitted, it defaults to `y=x`) or `theta`, `r` polar coordinates (where, if `r` is omitted, it defaults to `r=1`).

**Parameters**

- `worldCoordinates` *(bool)* – If `True`, return pixel coordinates relative to the `World`'s own `anchor` position. If `False`, return pixel coordinates relative to the `World`'s bottom left corner irrespective of its `anchor`.
- `polar` *(bool)* – If `True`, input coordinates are interpreted as an angle (in degrees) and an optional radius (0 denoting the center, 1 denoting the edge).
Examples:

```python
instance.Place( [ -1, 1 ] )  # top left corner
instance.Place( -1, 1 )     # likewise, top left corner
instance.Place( 90, polar=True )  # middle of top edge (radius 1 assumed)
instance.Place( [ 90, 0.5 ], polar=True )  # halfway between center and top
instance.Place( 90, 0.5, polar=True )  # likewise, halfway between center and...
```

classmethod Properties (includeShortcuts=False)

Return a list of all managed properties of this class. (Class method)

Parameters includeShortcuts — Determines whether ManagedShortcut instances should be included (True) or not (False; default) along with ManagedProperty instances.

Returns A list of ManagedProperty (and optionally also ManagedShortcut) descriptor instances.

ResetClock (other=None)

A Stimulus may have callback functions that are called on every frame—for example, a function that you have registered as its AnimationCallback, or functions that you assign to its managed properties (dynamic value assignment). In all cases, the argument that is passed to these callbacks on each frame is t, time in seconds. By default, this t is the same as the World's t, i.e. the number of seconds have elapsed since the World began rendering. However, if you wish, you can alter the t0 from which each Stimulus instance’s clock counts:

```python
stim.ResetClock()          # resets the clock to 0
stim.ResetClock( otherStim )  # synchronizes with `otherStim`
```

SavePage (key)

Save the current “page” (a bundle of properties determining texture, envelope size and envelope shape) in the dictionary self.pages under the specified key.

See also: page, NewPage(), and SwitchTo()

Set (**kwargs)

Set the values of multiple managed properties in one call. An error will be raised if you try to set the value of a non-existent, or non-managed, property.

Returns self

Example:

```python
instance.Set( rotation=90, color=(0, 0, 1), contrast=0.5 )
```

SetAnimationCallback (callback)

Bind the callable object callback as the instance’s animation callback. Each object, whether it is a World or a Stimulus, may optionally have a single animation callback which is called on every frame. If the callback argument is None, any existing callback is removed.

The animation callback is installed as the attribute self.Animate. By default, this attribute is None.

The prototype for an animation callback can be callback(self, t) or just callback(t) (if it's the latter then you can, alternatively, simply assign it as self.Animate = callback).

Example:

```python
def animate( self, t ):
    print( t )
my_world = Shady.World().SetAnimationCallback( animate )
```
There is also a decorator version of the same operation, simply called `AnimationCallback()`:

```python
stim = my_world.Stimulus()
@stim.AnimationCallback
def Animate( self, t ):
    print( t )
```

**classmethod SetDefault(**kwargs)**
Affects the class. Sets the default values of named properties. In instances of the class created subsequently, managed properties will have the new default values.

Example:

```python
cls.SetDefault( color=(1,0,0), rotation=90 )
```

**SetDynamic(name, func, order=-1, canonicalized=False)**
Associate a “dynamic” (i.e. a function that can be called repeatedly to set an attribute) with the name of an attribute.

For example:

```python
foo.Set( 'bar', lambda t: t ** 2 )
```

This will set `foo.bar` to `t ** 2` every time the method `foo._RunDynamics(t)` is called (this call will happen automatically from the API user’s point of view, in the object’s infrastructure implementation).

A dynamic can be attached to any attribute name. If the `bar` attribute happens to be a ManagedProperty or ManagedShortcut, then it will also automatically support “dynamic value assignment”, i.e. you can do:

```python
foo.bar = lambda t: t ** 2
```

as a syntactic shorthand for `SetDynamic()`—the setter will detect the fact that the value is callable, and divert it to the register of dynamics rather than assigning it directly (so the actual static value you get from querying `foo.bar` will not immediately change).

**Parameters**

- **name** (str) – name of the attribute
- **func** – callable object, or `None` to remove any dynamic that may already be associated with name
- **order** (int, float) – optional numeric ordering key that allows you to control the serial order in which dynamics are evaluated
- **canonicalized** (bool) – for internal use only. ManagedProperty and ManagedShortcut descriptors can have multiple aliases. The default settings, `canonicalized=False` says that name hasn not necessarily been translated to the descriptor’s canonical name, so this should be attempted

**Returns** self

See also: `GetDynamic()`, `GetDynamics()`, `ClearDynamics()`

**SetLUT(value)**
Sets or unsets a LookupTable for a `World` or `Stimulus`. (For `World`s, this will only be effective to the extent that the `World`s atmosphere properties are shared with `Stimulus` instances—for example, the canvas.)
Calling this method is the functional equivalent of setting the `lut` property:

```python
stim.SetLUT( value )  # These are equivalent
stim.lut = value      #
```

Setting a look-up table disables automatic linearization via `gamma` and automatic dynamic-range enhancement via `ditheringDenominator`, and allows you to take direct control of these issues (although only for one dimension of luminance per pixel: using a look-up table is a form of indexed-color rendering).

See the Shady.Documentation.PreciseControlOfLuminance docstring or Gamma Correction, Dynamic Range Enhancement, and the Canvas for more details.

**Parameters**

- **value** *(None, str, list, numpy.ndarray, LookupTable)* – A pre-existing LookupTable instance may be used here. Alternatively, any valid constructor argument for a LookupTable may be used, and the instance will be constructed implicitly. That means you can use:
  - a `numpy.ndarray` of integers in n-by-3 or m-by-n-by-3 arrangement, or
  - a list of lists that can be implicitly converted into such an array by `numpy`, or
  - the filename of a `npy` or `npz` file that contains such an array (under the variable name `lut` in the latter case) to be loaded by `Shady.Linearization.LoadLUT()`, or
  - the filename of a `png` file containing the look-up table entries as RGB pixel values in column-first order, again to be loaded by `Shady.Linearization.LoadLUT()`.

Finally you have the option of setting `None`, to disable the usage of look-up tables.

**Returns**

A LookupTable instance.

**ShareProperties** *(pargs, **kwargs)*

Share the underlying array storage of the named managed properties with other objects, so that a change to one affects all. Optionally, set the values at the same time. Calling syntax is flexible - the following examples all take certain property arrays associated with instance `a` and share them with instances `b`, `c` and `d`:

```python
a.ShareProperties( 'gamma noiseAmplitude backgroundColor', b, c, d )
a.ShareProperties( [ 'gamma', 'noiseAmplitude', 'backgroundColor' ], [ b, c, d ] )
a.ShareProperties( b, c, d, gamma=-1, noiseAmplitude=0.01, backgroundColor=0.5 ) # sets values at the same time
```

A similar effect can be obtained via `b.LinkPropertiesWithMaster( a, ... )`—although there you are limited to one sharee at a time—and a syntactic shorthand for this is to pretend you are assigning the master object instance itself to the target property:

```python
b.gamma = a
```

Undo with `MakePropertiesIndependent()` (or `b.gamma = b`)

**Returns** `self`

**ShareTexture** *(others)*

This is a wrapper around `ShareProperties()` that allows `Stimulus` instances to share a texture—it shares the `textureID, textureSlotNumber, textureSize` and `useTexture` managed properties.

**SwitchTo** *(key)*

Switch to the “page” associated with the given `key`. A page is a bundle of properties that determine texture, envelope size and envelope shape. The `key` argument must be one of the keys in the dictionary `self.pages`. Note that if the current settings have not been stored (via `SavePage()`), via the explicit

---

Chapter 5. To cite Shady:
specification of a key during `NewPage()`, or via the automated loop of `NewPage()` calls provided by `LoadPages()` then this method will cause the current settings to be lost.

See also: `page`, `NewPage()`, `LoadPages()` and `SavePage()`

**WaitFor** *(condition)*

This function blocks the current thread until a specified condition becomes `True`. The condition will be checked repeatedly in between 1-millisecond sleeps. It can be one of two things:

- a callable (function) which returns `True` to signal that the wait is over - e.g.:

  ```python
  stim.WaitFor( lambda: stim.y < 0 )
  ```

- a string that names a dynamic attribute belonging to the current instance. In this case, the wait ends when there is no longer a dynamic attached the the property or shortcut in question. A dynamic may be removed due to explicit action of another thread in your program, or explicit action in this instance’s `AnimationCallback` (or indeed any instance’s `AnimationCallback`). Alternatively, a dynamic may automatically remove itself, by raising a `StopIteration` exception (the `Function` object returned by `Shady.Dynamics.Transition` is an example of something that does this):

  ```python
  stim.scaling = Shady.Transition( stim.scaling, 2, duration=5 )
  # the stimulus will now double in size over the course of 5 sec
  stim.WaitFor( 'scaling' )
  # wait until it's finished
  ```

All of this assumes that you are operating in a different thread from the `World`—if you call `WaitFor` from inside one of the `World` or `Stimulus` callbacks, then it will simply sleep indefinitely because nothing will get the chance to change.

**addb**

This is a **managed shortcut**. It is a shortcut for `offset[2]`

**addg**

This is a **managed shortcut**. It is a shortcut for `offset[1]`

**addr**

This is a **managed shortcut**. It is a shortcut for `offset[0]`

**alpha**

This is a **managed property**. It is a floating-point number from 0 to 1. It specifies the opacity of the `Stimulus` as a whole. Note that with psychophysical stimuli you should always ensure `alpha == 1`, and manipulate `backgroundColor` and `normalizedContrast` instead: this is because alpha-blending is carried out by the graphics card AFTER our linearization (via the `gamma` property or via a look-up table) is applied. Therefore a blended result will no longer be linearized.

- Default value: 1.0
- Canonical name: `alpha`
- Other aliases: `fgalphaopacity`

**anchor**

This is a **managed property**. It is a sequence of two floating-point numbers expressed in normalized coordinates (from -1 to +1). It denotes where, on the surface of the envelope, the anchor point will be. This anchor point is the point whose coordinates are manipulated directly by the other properties, and it also serves as the origin of any scaling and rotation of the envelope. The default value is `[0.0, 0.0]` denoting the center, whereas `[-1.0, -1.0]` would be the bottom left corner.

The translation caused by changes to `anchor` is always rounded down to an integer number of pixels, to avoid it becoming an unforeseen cause of interpolation artifacts.
• Default value: \( [0.0, 0.0] \)

• Subscripting shortcuts:
  
  – \( ax_n \) is a shortcut for \( anchor[0] \)
  
  – \( ay_n \) is a shortcut for \( anchor[1] \)

**anchor\_x**

\( anchor\_x \) is an alias for \( ax_n \)

**anchor\_y**

\( anchor\_y \) is an alias for \( ay_n \)

**angle**

\( angle \) is an alias for \( envelopeRotation \)

**atmosphere**

This property actually encompasses multiple managed properties, all related to linearization and dynamic-range enhancement.

If you query this property, you will get a `dict` of the relevant property names and values. You can also assign such a dictionary to it.

More usefully, you can use it to link all the properties between instances in one go:

```python
stim.atmosphere = world  # hard-links all the "atmosphere" properties
```

For more information, see the `Shady.Documentation.PreciseControlOfLuminance` documentation or click here.

**ax\_n**

This is a **managed shortcut**. It is a shortcut for \( anchor[0] \)

  • Canonical name: \( ax_n \)
  
  • Other aliases: \( anchor\_x \)

**ay\_n**

This is a **managed shortcut**. It is a shortcut for \( anchor[1] \)

  • Canonical name: \( ay_n \)
  
  • Other aliases: \( anchor\_y \)

**backgroundAlpha**

This is a **managed property**. It is a floating-point number from 0 to 1, indicating the opacity at locations where the signal has been attenuated away completely (by windowing via `plateauProportion`, by a custom `modulationFunction`, or by manipulation of overall `normalizedContrast`).

For psychophysical stimuli, ensure `backgroundAlpha` is 1.0 and manipulate `backgroundColor` instead: although alpha can be used for windowing in this way, alpha-blending is applied post-linearization so the result will not be well linearized, except in very fragile special cases.

  • Default value: 1.0
  
  • Canonical name: `backgroundAlpha`
  
  • Other aliases: `bgalpha`

**backgroundColor**

This is a **managed property**. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. It specifies the color at locations in which the carrier signal (texture and/or
foreground color and/or functionally generated carrier signal) has been attenuated away completely (by contrast scaling, windowing, or custom contrast modulation pattern).

- Default value: \[0.5, 0.5, 0.5\]
- Canonical name: \textit{backgroundColor}
- Other aliases: \textit{bg bgcolor}

Subscripting shortcuts:

- \texttt{bgred} is a shortcut for \textit{backgroundColor}[0]
- \texttt{bggreen} is a shortcut for \textit{backgroundColor}[1]
- \texttt{bgblue} is a shortcut for \textit{backgroundColor}[2]

\texttt{bg}

\texttt{bg} is an alias for \textit{backgroundColor}

\texttt{bgalpha}

\texttt{bgalpha} is an alias for \textit{backgroundAlpha}

\texttt{bgblue}

This is a \textit{managed shortcut}. It is a shortcut for \textit{backgroundColor}[2]

\texttt{bgcolor}

\texttt{bgcolor} is an alias for \textit{backgroundColor}

\texttt{bggreen}

This is a \textit{managed shortcut}. It is a shortcut for \textit{backgroundColor}[1]

\texttt{bgred}

This is a \textit{managed shortcut}. It is a shortcut for \textit{backgroundColor}[0]

\texttt{blue}

This is a \textit{managed shortcut}. It is a shortcut for \textit{color}[2]

- Canonical name: \textit{blue}
- Other aliases: \textit{fgblue}

\texttt{bluegamma}

This is a \textit{managed shortcut}. It is a shortcut for \textit{gamma}[2]

\texttt{bluenoise}

This is a \textit{managed shortcut}. It is a shortcut for \textit{noiseAmplitude}[2]

\texttt{carrierRotation}

This is a \textit{managed property}. It is a scalar number, expressed in degrees. The carrier will be rotated counterclockwise by this number of degrees around the center of the envelope. Note that if your rotation values is not divisible by 90, this will introduce interpolation artifacts into stimuli that use textures. (Unlike \textit{envelopeRotation}, however, this will not compromise pure functionally-generated stimuli.)

- Default value: 0.0
- Canonical name: \textit{carrierRotation}
- Other aliases: \textit{cr}

\texttt{carrierScaling}

This is a \textit{managed property}. It is a sequence of two floating-point numbers denoting horizontal and vertical scaling factors. The carrier will be magnified by these factors relative to an origin the center of the envelope. Note that scaling values != 1.0 will introduce interpolation artifacts into stimuli that use textures (but unlike \textit{envelopeScaling}, this will not compromise pure functionally-generated stimuli).
Shady Documentation, Release 1.9.1

• Default value: [1.0, 1.0]
• Canonical name: carrierScaling
• Other aliases: cscale cscaling

• Subscripting shortcuts:
  – cxscale is a shortcut for carrierScaling[0]
  – cyscale is a shortcut for carrierScaling[1]

carrierTranslation
This is a managed property. It is a sequence of two numbers, expressed in pixels, corresponding to x and y dimensions. It shifts the carrier (texture stimulus and/or shader function) relative to the envelope. Note that non-integer translation values will introduce interpolation artifacts into stimuli that use textures (but unlike envelopeTranslation, this should not compromise pure functionally-generated stimuli).

• Default value: [0.0, 0.0]

• Subscripting shortcuts:
  – cx is a shortcut for carrierTranslation[0]
  – cy is a shortcut for carrierTranslation[1]

color
This is a managed property. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. Values may also be negative, in which case no colorization is applied in the corresponding channel.

Where non-negative, foreground color plays slightly different roles depending on other parameters:

If the Stimulus uses a texture, the pixel values from the texture are tinted via multiplication with the color values.

If the Stimulus uses a signalFunction then the signal is also multiplied by the color before being added.

If there is no texture and no signalFunction, the carrier image consists of just the specified uniform solid color. The Stimulus color may still be attenuated towards the backgroundColor—uniformly by setting normalizedContrast < 1.0, and/or as a function of space by setting plateauProportion >= 0.0 or by using a modulationFunction).

• Default value: [-1.0, -1.0, -1.0]

• Canonical name: color

• Other aliases: fg fgcolor foregroundColor

• Subscripting shortcuts:
  – red is a shortcut for color[0]
  – green is a shortcut for color[1]
  – blue is a shortcut for color[2]

contrast
  contrast is an alias for normalizedContrast

cr
cr is an alias for carrierRotation

cscale
cscale is an alias for carrierScaling
**cscaling**

*cscaling* is an alias for *carrierScaling*

**cx**

This is a *managed shortcut*. It is a shortcut for *carrierTranslation[0]*

**cxscale**

This is a *managed shortcut*. It is a shortcut for *carrierScaling[0]*

- Canonical name: *cxscale*
- Other aliases: *cxscaling*

**cxscaling**

*cxscaling* is an alias for *cxscale*

**cy**

This is a *managed shortcut*. It is a shortcut for *carrierTranslation[1]*

**cyscale**

This is a *managed shortcut*. It is a shortcut for *carrierScaling[1]*

- Canonical name: *cyscale*
- Other aliases: *cyscaling*

**cyscaling**

*cy scaling* is an alias for *cyscale*

**dd**

*dd* is an alias for *ditheringDenominator*

**depth**

*depth* is an alias for *z*

**depthPlane**

*depthPlane* is an alias for *z*

**ditheringDenominator**

This is a *managed property*. It is a floating-point number. It should be 0 or negative to disable dithering, or otherwise equal to the maximum DAC value (255 for most video cards). It allows implementation of the “noisy-bit” dithering approach (Allard & Faubert, 2008) for increasing effective dynamic range.

This is distinct from the *noiseAmplitude* property, which specifies noise that is (a) applied pre-gamma-correction, (b) can be scaled differently in different color channels, but (c) is otherwise perfectly correlated across color channels. Noisy-bit dithering, on the other hand (a) is applied post-gamma-correction, (b) has the same amplitude in all color channels but (c) is independent in each color channel.

Note that, like gamma-correction, noisy-bit dithering is disabled for stimuli that use a *lut*.

- Default value: dithering enabled (value determined automatically)
- Canonical name: *ditheringDenominator*
- Other aliases: *dd*

**drawMode**

This is a *managed property*. It is an integer that selects between different drawing behaviors. The different values are given meaningful names in the namespace *DRAWMODE*, whose documentation also explains the behavior of the different draw modes.

- Default value: 1

**envelopeOrigin**

This is a *managed property*. It is a triplet of numbers, expressed in pixels, denoting the starting-point, in the
coordinate system of the parent World, of the offsets envelopeTranslation (which is composed of x and y) and depth coordinate z. The actual rendered position of the anchor point of Stimulus s will be:

\[
\text{int}(s.x) + s.ox, \quad \text{int}(s.y) + s.oy, \quad s.z + s.oz
\]
relative to the World’s own origin.

You can manipulate envelopeOrigin exclusively and leave envelopeTranslation at 0, as an alternative way of specifying Stimulus position. This is the way to go if you prefer to work in 3D floating-point coordinates instead of 2D integers: unlike envelopeTranslation, this property gives you the opportunity to represent non-integer coordinate values. With that flexibility comes a caveat: non-whole-number values of ox and oy may result in artifacts in any kind of Stimulus (textured or not) due to linear interpolation during rendering. You may therefore wish to take care to round any values you assign, if you choose to use this property. If you exclusively use envelopeTranslation instead, this pitfall is avoided (as you can see in the formula above, its components x and y are rounded down to integer values when they are applied).

Note also that for all stimuli, you should ensure that the total depth coordinate, \( s.z + s.oz \), is in the range \((-1, +1)\).

- **Default value:** \([0.0, 0.0, 0.0]\)
- **Subscripting shortcuts:**
  - ox is a shortcut for envelopeOrigin[0]
  - oy is a shortcut for envelopeOrigin[1]
  - oz is a shortcut for envelopeOrigin[2]

**envelopePosition**

envelopePosition is an alias for envelopeTranslation

**envelopeRotation**

This is a managed property. It is a scalar number, expressed in degrees. The envelope will be rotated counter-clockwise by this number of degrees around its anchor. Note that such transformations of the envelope (except at multiples of 90 degrees) will introduce small artifacts into any stimulus, due to linear interpolation.

- **Default value:** 0.0
- **Canonical name:** envelopeRotation
- **Other aliases:** angle orientation rotation

**envelopeScaling**

This is a managed property. It is a sequence of two floating-point numbers denoting horizontal and vertical scaling factors. The actual rendered size, in pixels, of the scaled Stimulus s will be \( s.\text{envelopeScaling} \times s.\text{envelopeSize} \). Note that such transformations of the envelope will introduce small artifacts into any stimulus, due to linear interpolation.

- **Default value:** \([1.0, 1.0]\)
- **Canonical name:** envelopeScaling
- **Other aliases:** scale scaling
- **Subscripting shortcuts:**
  - xscaling is a shortcut for envelopeScaling[0]
  - yscaling is a shortcut for envelopeScaling[1]
envelopeSize

This is a managed property. It is a sequence of two numbers denoting the unscaled width and height of the envelope (i.e. width and height in texel units). Change these numbers if, for example, you want to crop or repeat a texture, or to load new differently-shaped texture (the easiest way to do the latter is to call LoadTexture() method with argument updateEnvelopeSize=True). To change the size of the envelope by stretching the image content to fit, manipulate envelopeScaling or scaledSize instead.

- Default value: [200, 200]
- Canonical name: envelopeSize
- Other aliases: size
- Subscripting shortcuts:
  - width is a shortcut for envelopeSize[0]
  - height is a shortcut for envelopeSize[1]

envelopeTranslation

This is a managed property. It is a pair of numbers, expressed in pixels. It dictates the two-dimensional coordinates of the Stimulus location within the drawing area. The values are rounded down to integer values when they are applied, to avoid artifacts that might otherwise be introduced inadvertently due to linear interpolation during rendering.

See also: envelopeOrigin

- Default value: [0.0, 0.0]
- Canonical name: envelopeTranslation
- Other aliases: envelopePosition pos position xy
- Subscripting shortcuts:
  - x is a shortcut for envelopeTranslation[0]
  - y is a shortcut for envelopeTranslation[1]

fg

fg is an alias for color

fgalpha

fgalpha is an alias for alpha

fgblue

fgblue is an alias for blue

fgcolor

fgcolor is an alias for color

fggreen

fggreen is an alias for green

fgred

fgred is an alias for red

foregroundColor

foregroundColor is an alias for color

frame

This non-managed property is an integer denoting the index of the current frame of a multi-frame stimulus. Note that frames are concatenated horizontally in the underlying carrier texture, so a change in frame is actually achieved by manipulating cx, otherwise known as carrierTranslation[0].
gamma
This is a managed property. It is a triplet of values denoting the screen gamma that should be corrected-for in each of the red, green and blue channels. A gamma value of 1 corresponds to the assumption that your screen is already linear. Setting gamma values other than 1 is an alternative to using a pre-linearized lookup-table or lut.

Any value less than or equal to 0.0 is interpreted to denote the sRGB function, which is a standard piecewise function that follows the $\gamma=2.2$ curve quite closely (although the exponent it uses is actually slightly higher).

Note that gamma is ignored for stimuli that use a lut

- Default value: [1.0, 1.0, 1.0]
- Subscripting shortcuts:
  - redgamma is a shortcut for $\text{gamma}[0]$
  - greengamma is a shortcut for $\text{gamma}[1]$
  - bluegamma is a shortcut for $\text{gamma}[2]$

green
This is a managed shortcut. It is a shortcut for color[1]

- Canonical name: green
- Other aliases: fggreen
greengamma
This is a managed shortcut. It is a shortcut for $\text{gamma}[1]$
greennoise
This is a managed shortcut. It is a shortcut for noiseAmplitude[1]

height
This is a managed shortcut. It is a shortcut for envelopeSize[1]

linearMagnification
This property governs the interpolation behavior applied when a textured Stimulus is enlarged (i.e. when its envelopeScaling or carrierScaling are greater than 1).

If True (default), interpolate pixel values linearly when enlarged.

If False, take the “nearest pixel” method when enlarged.

lut
The value of this property will either be None, or an instance of LookupTable. Assigning to this property is equivalent to calling the SetLUT() method—so you can assign any of the valid argument types accepted by that function. Assigning None disables look-up.

moda
moda is an alias for modulationDepth

modd
modd is an alias for modulationDepth

modf
modf is an alias for modulationFrequency

modfunc
modfunc is an alias for modulationFunction

modo
modo is an alias for modulationOrientation
modp

`modp` is an alias for `modulationPhase`

**modulationAmplitude**

`modulationAmplitude` is an alias for `modulationDepth`

**modulationDepth**

This is a managed shortcut. It is a shortcut for `modulationParameters[0]` (amplitude, i.e. modulation depth, from 0 to 1)

- Canonical name: `modulationDepth`
- Other aliases: `moda modd modulationAmplitude`

**modulationFrequency**

This is a managed shortcut. It is a shortcut for `modulationParameters[1]` (frequency in cycles/pixel)

- Canonical name: `modulationFrequency`
- Other aliases: `modf`

**modulationFunction**

This is a managed property. It is an integer specifying the index of a shader-side contrast modulation function. If it is left at 0, no function is used: the stimulus contrast is then dependent only on the overall `normalizedContrast` and on the window applied according to `plateauProportion`.

A value of 1 (which can also be referenced as the constant MODFUNC.SinewaveModulation) corresponds to the one and only shader-side modulation function that we provide out of the box, namely `SinewaveModulation` which performs sinusoidal contrast modulation. The parameters of this modulation pattern are determined by the `modulationParameters` property.

Further values, and hence further functions, may be supported if you add them yourself using `AddCustomModulationFunction()`.

In the shader, the return value of a modulation function is a float. This return value is used as a multiplier for stimulus contrast.

See also: `modulationParameters`, `signalFunction`, `signalParameters`, `windowingFunction`

- Default value: 0
- Canonical name: `modulationFunction`
- Other aliases: `modfunc`

**modulationOrientation**

This is a managed shortcut. It is a shortcut for `modulationParameters[2]` (orientation in degrees)

- Canonical name: `modulationOrientation`
- Other aliases: `modo`

**modulationParameters**

This is a managed property. It is a 4-element vector that can be used to pass parameters to the shader-side contrast-modulation function chosen by `modulationFunction`.

If `modulationFunction` is left at its default value of 0, these four values are ignored. For the one built-in shader modulation function (`modulationFunction=1` corresponding to the shader function `SinewaveModulation`), the values are interpreted as depth, frequency, orientation and phase of the desired sinusoidal modulation pattern.
Shady Documentation, Release 1.9.1

If you’re adding your own custom shader function via `AddCustomModulationFunction()`, your implementation of that function may choose to ignore or reinterpret this property as you wish. If you choose to use it, your shader code can access it as the uniform `vec4` variable `uModulationParameters`.

See also: `modulationFunction`, `signalFunction`, `signalParameters`

• Default value: `[0.0, 0.005, 0.0, 90.0]`

• **Subscripting shortcuts:**
  
  – `modulationDepth` is a shortcut for `modulationParameters[0]` (amplitude, i.e. modulation depth, from 0 to 1)
  
  – `modulationFrequency` is a shortcut for `modulationParameters[1]` (frequency in cycles/pixel)
  
  – `modulationOrientation` is a shortcut for `modulationParameters[2]` (orientation in degrees)
  
  – `modulationPhase` is a shortcut for `modulationParameters[3]` (phase in degrees)

`modulationPhase`

This is a **managed shortcut**. It is a shortcut for `modulationParameters[3]` (phase in degrees)

• Canonical name: `modulationPhase`

• Other aliases: `modp`

`nFrames`

Read-only non-managed property that returns the width of each frame, if the texture pixel data is divided up horizontally into multiple frames.

`noise`

`noise` is an alias for `noiseAmplitude`

`noiseAmplitude`

This is a **managed property**. It is a triplet of floating-point numbers corresponding to red, green and blue channels. Negative values lead to uniform noise in the range `[s.noiseAmplitude[i], -s.noiseAmplitude[i]]`. Positive values lead to Gaussian noise with standard deviation equal to the `noiseAmplitude` value.

The noise specified in this way is applied before gamma-correction, and is monochromatic (i.e. perfectly correlated across color channels, though it may have different amplitudes in each channel). It is particularly useful for applying very-low-amplitude noise prior to look-up in a high-dynamic-range bit-stealing look-up table. It can also be useful at higher amplitudes, to create visible noise effects.

It is not to be confused with noisy-bit dithering, which is applied post-gamma-correction (and only when a look-up table is not used), and which is controlled independently by the `ditheringDenominator` property.

• Default value: `[0.0, 0.0, 0.0]`

• Canonical name: `noiseAmplitude`

• Other aliases: `noise`

• **Subscripting shortcuts:**
  
  – `rednoise` is a shortcut for `noiseAmplitude[0]`
  
  – `greennoise` is a shortcut for `noiseAmplitude[1]`
  
  – `bluenoise` is a shortcut for `noiseAmplitude[2]`
normalizedContrast

This is a managed property. It is a scalar floating-point value in the range 0.0 to 1.0 which scales the overall contrast of the Stimulus. At a contrast of 0, the Stimulus is reduced to its backgroundColor.

- Default value: 1.0
- Canonical name: normalizedContrast
- Other aliases: contrast

offset

This is a managed property. It is a triplet of numbers, each in the range 0.0 to 1.0, corresponding to red, green and blue channels. This is added uniformly to all pixel values before they are scaled by normalizedContrast or by windowing (via plateauProportion and windowingFunction) and/or custom modulation (via modulationFunction).

- Default value: [0.0, 0.0, 0.0]
- Subscripting shortcuts:
  - addr is a shortcut for offset[0]
  - addg is a shortcut for offset[1]
  - addb is a shortcut for offset[2]

on

on is an alias for visible

opacity

opacity is an alias for alpha

orientation

orientation is an alias for envelopeRotation

outOfRangeAlpha

This is a managed property. It is a floating-point number from 0 to 1. It specifies the color that should be used for pixels whose values go out of range. A negative value disables this feature (the alpha value of an out-of-range pixel is left unchanged).

- Default value: 1.0

outOfRangeColor

This is a managed property. It is a triplet of numbers, each in the range 0 to 1, corresponding to red, green and blue channels. It specifies the color that should be used for pixels whose values go out of range. A negative value means “do not flag out-of-range values in this color channel” (values are merely clipped in the range 0 to 1 in that case).

- Default value: [1.0, 0.0, 1.0]

ox

This is a managed shortcut. It is a shortcut for envelopeOrigin[0]

oy

This is a managed shortcut. It is a shortcut for envelopeOrigin[1]

oz

This is a managed shortcut. It is a shortcut for envelopeOrigin[2]

page

A Stimulus may optionally have more than one “page”. A page is a label attached to a particular set of property values that determine texture, envelope size and envelope shape.
A page may have any hashable label—integers and strings are usually the most meaningful. The `page` property allows you to query the label of the current page, or switch to a different page according to its label. It is not a managed property, but it does support dynamic value assignment.

See also: `LoadPages()`, `NewPage()`, `SavePage()` and `SwitchTo()`

### penThickness

This is a **managed property**. It is a scalar floating-point value that determines the width, in pixels, of lines and points drawn when you set `drawMode` to `DRAWMODE.POINTS`, `DRAWMODE.LINES`, `DRAWMODE.LINE_STRIP` or `DRAWMODE.LINE_LOOP`. Implementation is driver-dependent however: many graphics cards seem to consider line-thickness to be a legacy feature and have dropped support for it (for lines, but maybe not for points) in their pure modern-OpenGL contexts.

- Default value: 1.0

### plateauProportion

This is a **managed property**. It is sequence of two floating-point values corresponding to the x and y dimensions.

A negative value indicates that no windowing is to be performed in the corresponding dimension.

A value in the range [0, 1] causes windowing to occur: with the default (raised cosine) `windowingFunction`, a `plateauProportion` of 0 gives a Hann window, 1 gives a boxcar window, and intermediate values combine the specified amount of constant plateau with a raised- cosine falloff to the edge of the envelope (i.e., a Tukey window).

A non-zero plateau proportion can also be combined with other (custom self-written) `windowingFunction` implementations.

Note that this means [1, 1] gives a circular or elliptical envelope with sharp edges, in contrast to [-1, -1] which gives a square or rectangular envelope.

- Default value: [-1.0, -1.0]
- Canonical name: `plateauProportion`
- Other aliases: `pp`
- **Subscripting shortcuts:**
  - `ppx` is a shortcut for `plateauProportion[0]`
  - `ppy` is a shortcut for `plateauProportion[1]`

### points

This is a view into the `Stimulus` instance’s optional array of coordinates. If you do not have the third-party package `numpy` installed, then this is fairly limited (although you can assign a list of alternating x and y coordinates to it, you will not be able to do any array arithmetic operations with it, which is where its power lies). With `numpy`, your `points` will appear as an n-by-2 array of coordinates (you can also view the same data as a sequence of complex numbers, by accessing as `pointsComplex`). These coordinates are not used in the default `drawMode`, which is `DRAWMODE.QUAD`. To learn how other draw modes use the coordinates, see the documentation for the `DRAWMODE` namespace.

Assignment to `points` is interchangeable with assignment to `pointsComplex`—in either case, the input can be a sequence of n complex numbers OR an n-by-2 array of real-valued coordinates. Under the hood, assignment to `points` changes the content of the managed property `pointsXY` and also adjusts the managed property `nPoints`.

The coordinate system of the `points` is always in pixels relative to the bottom-left corner of the `Stimulus` bounding-box—i.e. the bottom-left corner of the rectangle you would see if you were to switch the `Stimulus` back to `drawMode=Shady.DRAWMODE.QUAD`. The `World.anchor`,
Stimulus.position, Stimulus.anchor and Stimulus.size determine where this local origin actually lies on screen, but the points themselves are expressed independently of these, and points defined outside the bounding-box (e.g. with negative coordinates) still get drawn.

pointsComplex
This is a view into the Stimulus instance’s optional array of coordinates. The same data are also accessible as points. The only difference is that when you ask for pointsComplex, the data are interpreted as a sequence of complex numbers (requires the third-party package numpy).

Assignment to pointsComplex is interchangeable with assignment to points—in either case, the input can be a sequence of n complex numbers OR an n-by-2 array of real-valued coordinates.

pos
pos is an alias for envelopeTranslation

position
position is an alias for envelopeTranslation

pp
pp is an alias for plateauProportion

ppx
This is a managed shortcut. It is a shortcut for plateauProportion[0]

ppy
This is a managed shortcut. It is a shortcut for plateauProportion[1]

red
This is a managed shortcut. It is a shortcut for color[0]
  • Canonical name: red
  • Other aliases: fgred

redgamma
This is a managed shortcut. It is a shortcut for gamma[0]

rednoise
This is a managed shortcut. It is a shortcut for noiseAmplitude[0]

rotation
rotation is an alias for envelopeRotation

scale
scale is an alias for envelopeScaling

scaledHeight
Non-managed property that reflects the product of envelopeSize[1] and envelopeScaling[1]. Assigning to this property will change envelopeScaling[1] accordingly. Dynamic value assignment is supported.

scaledSize
Non-managed property that reflects the product of envelopeSize and envelopeScaling. Assigning to this property will change envelopeScaling accordingly. Dynamic value assignment is supported. Note that you can also address scaledWidth and scaledHeight separately if you wish.

scaledWidth
Non-managed property that reflects the product of envelopeSize[0] and envelopeScaling[0]. Assigning to this property will change envelopeScaling[0] accordingly. Dynamic value assignment is supported.

scaling
scaling is an alias for envelopeScaling
siga

*siga* is an alias for *signalAmplitude*

sigf

*sigf* is an alias for *signalFrequency*

sigfunc

*sigfunc* is an alias for *signalFunction*

**signalAmplitude**

This is a managed shortcut. It is a shortcut for *signalParameters*[0] (amplitude from 0 to 1)

- Canonical name: *signalAmplitude*
- Other aliases: *siga*

**signalFrequency**

This is a managed shortcut. It is a shortcut for *signalParameters*[1] (frequency in cycles/pixel)

- Canonical name: *signalFrequency*
- Other aliases: *sigf*

**signalFunction**

This is a managed property. It is an integer specifying the index of a shader-side signal function. If it is left at 0, no function is used: the carrier content is then dependent purely on the texture, if any, or is blank if no texture was specified. A value of 1 (which can also be referenced as the constant SIGFUNC. SinewaveSignal) corresponds to the one and only shader-side signal function that we provide out of the box, namely SinewaveSignal which generates a sinusoid. The parameters of the sinusoid are determined by the *signalParameters* property.

Further values, and hence further functions, may be supported if you add them yourself using AddCustomSignalFunction().

In the shader, the return value of a signal functions is either a vec3 or a float. This return value gets multiplied by the color of the Stimulus, and added to its backgroundColor (and/or its texture, if any). If color is negative (i.e. disabled, as it is by default) then the function output is multiplied by vec3(1.0, 1.0, 1.0).

See also: *signalParameters*, modulationFunction, modulationParameters, windowingFunction

- Default value: 0
- Canonical name: *signalFunction*
- Other aliases: *sigfunc*

**signalOrientation**

This is a managed shortcut. It is a shortcut for *signalParameters*[2] (orientation in degrees)

- Canonical name: *signalOrientation*
- Other aliases: *sigo*

**signalParameters**

This is a managed property. It is a 4-element vector that can be used to pass parameters to the shader-side carrier-signal-generating function chosen by *signalFunction*. If *signalFunction* is left at its default value of 0, the *signalParameters* are ignored. For the one built-in shader signal function (signalFunction=1 corresponding to the shader function SinewaveSignal), these parameters are interpreted as amplitude, frequency, orientation and phase of the sinusoidal pattern. Signal functions are additive to your background and/or texture, so if you have no texture and a background color of, for
example, 0.3 or 0.7, a sinewave amplitude greater than 0.3 will go out of range at full contrast. (Beware also the additive effect of noise, if your \texttt{noiseAmplitude} is not 0.)

If you’re adding your own custom shader function via \texttt{AddCustomSignalFunction()}, your implementation of that function may choose to ignore or reinterpret this property as you wish. If you choose to use it, your shader code can access it as the uniform \texttt{vec4} variable \texttt{uSignalParameters}.

See also: \texttt{signalFunction}, \texttt{modulationFunction}, \texttt{modulationParameters}

- Default value: \[1.0, 0.05, 0.0, 0.0\]

Subscripting shortcuts:

- \texttt{signalAmplitude} is a shortcut for \texttt{signalParameters[0]} (amplitude from 0 to 1)
- \texttt{signalFrequency} is a shortcut for \texttt{signalParameters[1]} (frequency in cycles/pixel)
- \texttt{signalOrientation} is a shortcut for \texttt{signalParameters[2]} (orientation in degrees)
- \texttt{signalPhase} is a shortcut for \texttt{signalParameters[3]} (phase in degrees)

\texttt{signalPhase}

This is a \textit{managed shortcut}. It is a shortcut for \texttt{signalParameters[3]} (phase in degrees)

- Canonical name: \texttt{signalPhase}
- Other aliases: \texttt{sigp}

\texttt{sigp}

\texttt{sigp} is an alias for \texttt{signalPhase}

\texttt{size}

\texttt{size} is an alias for \texttt{envelopeSize}

\texttt{smoothing}

This is a \textit{managed property}. It is an integer value that determines whether lines, dots and polygons should be drawn with or without smoothing. The results are unfortunately unpredictable as they vary according to your graphics card and drivers. Example: with \texttt{DRAWMODE.POINTS}, setting \texttt{smoothing=1} causes the dots to be round rather than square—but with some graphics drivers, they will remain square regardless (to guarantee round dots, it’s better to draw tiny many-sided polygons). A value of 0 means no smoothing. A value of 1 means points and lines should be smoothed. A value of 2 means lines, points and polygons are smoothed (note that polygon smoothing causes diagonal “crack” artifacts on many graphics cards/drivers, so this option is not enabled by default).

- Default value: 1

\texttt{text}

To use the \texttt{text} property of the \texttt{Stimulus} class, you must first explicitly import \texttt{Shady.Text} (see \texttt{Shady.Text} for documentation on this property).

\texttt{textureChannels}

This is a \textit{managed property}. It is an integer value denoting whether the texture has 1, 2, 3 or 4 channels. If there is no texture, the value is -1. You should consider this property read-only—do not change the value by hand.

- Default value: -1

\texttt{textureSize}

This is a \textit{managed property}. It is a pair of values denoting the width and height of the texture data, in pixels. If there is no texture, both values are -1. You should consider this property read-only—do not change the values by hand.
• Default value: \([-1, -1]\)

**useTexture**

This is a managed property. It is a boolean value. The default value is `True` if a texture has been specified, and this causes pixel values to be drawn from the texture specified by the constructor argument `source` (or the `source` argument to a subsequent `NewPage()` or `LoadTexture()` call). If `useTexture` is set to `False`, the pixel values are determined by `backgroundColor` and/or `foregroundColor` (as well as `offset`, `normalizedContrast`, and any windowing and shader-side `signalFunction` or `modulationFunction`).

  • Default value: 1

**video**

To use the `video` property of the `Stimulus` class, you must first explicitly import `Shady.Video` (see `Shady.Video` for documentation on this property).

**visible**

This is a managed property. It is a boolean value that determines whether the `Stimulus` is rendered or not.

  • Default value: 1

  • Canonical name: `visible`

  • Other aliases: `on`

**width**

This is a managed shortcut. It is a shortcut for `envelopeSize[0]`

**windowFunction**

`windowFunction` is an alias for `windowingFunction`

**windowingFunction**

This is a managed property. It is an integer specifying the index of a shader-side windowing function. If it is set to 0 (or if the `plateauProportion` property is negative), no spatial windowing is used.

The default value of 1 (which can also be referenced as the constant `WINFUNC.Hann`) corresponds to the one and only shader-side windowing function that we provide out of the box, namely Hann, which causes contrast to fall off according to a raised cosine function of radial distance.

Further values, and hence further functions, may be supported if you add them yourself using `AddCustomWindowingFunction()`.

In the shader, the windowing function takes one input argument (a `float` whose value will range from 0 at peak of the window to 1 at the edge) and return a `float`. This return value is used as a multiplier for stimulus contrast.

See also: `signalFunction`, `modulationFunction`

  • Default value: 1

  • Canonical name: `windowingFunction`

  • Other aliases: `windowFunction winfunc`

**winfunc**

`winfunc` is an alias for `windowingFunction`

**x**

This is a managed shortcut. It is a shortcut for `envelopeTranslation[0]`

**xscale**

`xscale` is an alias for `xscaling`
xscaling
This is a managed shortcut. It is a shortcut for envelopeScaling[0]
• Canonical name: xscaling
• Other aliases: xscale

xy
xy is an alias for envelopeTranslation

yscale
yscale is an alias for yscaling

yscaling
This is a managed shortcut. It is a shortcut for envelopeScaling[1]
• Canonical name: yscaling
• Other aliases: yscale

z
This is a managed property. It is a floating-point number that determines the depth plane of the Stimulus. The convention is that negative values put you closer to the camera, and positive further away; also, you must ensure -1 <= z <= +1. Since the projection is orthographic, the value is purely used for depth-sorting of stimuli: therefore, setting z to a non-integer value will not cause interpolation artifacts.
• Default value: 0.0
• Canonical name: z
• Other aliases: depth depthPlane

Global Functions and Constants

Shady.AddCustomSignalFunction (code)
Defines a new signal function in the fragment shader.
Must be called, before construction of your World. The code argument is a (usually triple-quoted multi-line) string containing the complete definition of a function in GLSL.
The prototype for the function may be one of the following:

```glsl
float MySignalFunction( vec2 xy ) { ... }
vec3 MySignalFunction( vec2 xy ) { ... }
```

where xy are coordinates measured in pixels relative to the center of the stimulus. Obviously we’re using MySignalFunction here as a placeholder—use your own descriptive name for the new function. Your function must return either a floating-point signal value, or three-dimensional (red, green, blue) signal value. If non-negative values have been supplied in a Stimulus instance’s color property, your signal function will get multiplied by these. Then, any desired contrast modulation and windowing will be applied. Then, the result will be added to the backgroundColor.

Your new function can be applied to a Stimulus instance stim as follows:

```glsl
stim.signalFunction = Shady.SIGFUNC.MySignalFunction
```

If you want to parameterize your new function further, you can use the existing uniform vec4 uSignalParameters variable, which is linked to the Stimulus.signalParameters property. You
Shady Documentation, Release 1.9.1

can also define your own additional properties/uniform variables with the AddCustomUniform() class method.

See also:

AddCustomModulationFunction, AddCustomWindowingFunction World.
AddCustomUniform, Stimulus.AddCustomUniform,

Shady.AddCustomModulationFunction(code)
Defines a new contrast-modulation function in the fragment shader.

Must be called before construction of your World. The code argument is a (usually triple-quoted multi-line) string containing the complete definition of a function in GLSL.

The prototype for the function must be:

```glsl
float MyModulationFunction( vec2 xy ) { ... }
```

where xy are coordinates measured in pixels relative to the center of the stimulus. Obviously we’re using MyModulationFunction here as a placeholder—use your own descriptive name for the new function. Your function must return a floating-point contrast multiplier.

Your new function can be applied to a Stimulus instance stim as follows:

```cpp
stim.modulationFunction = Shady.MODFUNC.MyModulationFunction
```

If you want to parameterize your new function further, you can use the existing uniform vec4 uModulationParameters variable, which is linked to the Stimulus.modulationParameters property. You can also define your own additional properties/uniform variables with the AddCustomUniform() class method.

See also:

AddCustomSignalFunction, AddCustomWindowingFunction World.AddCustomUniform, Stimulus.AddCustomUniform,

Shady.AddCustomWindowingFunction(code)
Defines a new spatial windowing function in the fragment shader.

Must be called before construction of your World. The code argument is a (usually triple-quoted multi-line) string containing the complete definition of a function in GLSL.

The prototype for the function must be:

```glsl
float MyWindowingFunction( float r ) { ... }
```

where the domain of r if from 0 (in the center of the Stimulus, or indeed anywhere on its plateau if it has one) to 1 (at outer edge of the largest oval that fits in the Stimulus bounding box). Obviously we’re using MyWindowingFunction here as a placeholder—use your own descriptive name for the new function. Your function must return a floating-point contrast multiplier.

Your new function can be applied to a Stimulus instance stim as follows (bearing in mind that a negative plateauProportion value disables windowing entirely):

```cpp
stim.windowingFunction = Shady.WINFUNC.MyWindowingFunction
stim.plateauProportion = 0
```

If you want to parameterize your new function further, you can define your own additional properties/uniform variables with the AddCustomUniform() class method.

See also:
Shady Documentation, Release 1.9.1

AddCustomSignalFunction, AddCustomModulationFunction World.AddCustomUniform,
Stimulus.AddCustomUniform,

Shady.BackEnd (windowing=None, acceleration=None)

Globally specify the back-end windowing and rendering systems that future World instances should use.

Parameters

• windowing – specifies the windowing system. Possible values are as follows:
  'default': use the ShaDyLib dynamic library if available, else fall back on pyglet.
  'shadylib', 'accel', or 'glfw': use the ShaDyLib dynamic library (windowing is handled via the GLFW library from http://glfw.org).
  'pyglet': use pyglet (a third-party package that you will need to install separately if you want to use this option)
  'pygame': use pygame (a third-party package that you will need to install separately if you want to use this option)

• acceleration – specifies the rendering implementation, i.e. whether to use the ShaDyLib dynamic library (and if so, whether to use the “development” copy of ShaDyLib in cases where you have the entire Shady repository including the C++ sources for ShaDyLib) or whether to fall back on Python code for rendering (not recommended). Possible values are:
  None: leave things as they were (default).
  False: disable ShaDyLib and fall back on the pyglet or PyOpenGL code in the PyEngine submodule (this option is not recommended for time-critical presentation).
  True: if ShaDyLib is already imported, leave things as they are; if not, import either version of ShaDyLib or die trying. Prefer the development version, if available. Print the outcome.
  'bundled': silently import the bundled version of ShaDyLib from the Shady.accel sub-package, or die trying.
  'devel': silently import the development version of ShaDyLib from /../accel-src/release/, or die trying.
  'auto': try to import ShaDyLib. Prefer the development version, if available. Don’t die in the attempt. Whatever happens, print the outcome.

Returns If both input arguments are None, the name of the current windowing back-end is returned. Otherwise, returns None.

Shady.Screens (pretty_print=False)

Get details of any attached screens using the Screens() method of whichever windowing backend is enabled.

Parameters pretty_print (bool) – determines the type of the return value

Returns If pretty_print is True, returns a human-readable string. If pretty_print is False, returns a dict.

class Shady.SIGFUNC

This class is an enum container: a namespace of values that can be assigned to the signalFunction property of a Stimulus instance. Its members are named according to the signal functions available in the shader. The only built-in function is SinewaveSignal which has a value of 1. So, when you set the signalFunction property of a Stimulus to 1, the SinewaveSignal function is selected in the shader, and when you set it to 0, no signal function is selected.

You can define further signal functions, and hence add names and values to this namespace, with AddCustomSignalFunction.

5.5. Shady API Reference 191
See also: MODFUNC, WINFUNC

class Shady.MODFUNC
This class is an enum container: a namespace of values that can be assigned to the modulationFunction property of a Stimulus instance. Its members are named according to the modulation functions available in the shader. The only built-in function is SinewaveModulation which has a value of 1. So, when you set the modulationFunction property of a Stimulus to 1, the SinewaveModulation function is selected in the shader, and when you set it to 0, no modulation function is selected.

You can define further modulation functions, and hence add names and values to this namespace, with AddCustomModulationFunction.

See also: SIGFUNC, WINFUNC

class Shady.WINFUNC
This class is an enum container: a namespace of values that can be assigned to the windowingFunction property of a Stimulus instance. Its members are named according to the windowing functions available in the shader. The only built-in function is Hann which has a value of 1. So, when you set the windowingFunction property of a Stimulus to 1, the Hann function is selected in the shader, and when you set it to 0, no windowing function is selected.

You can define further windowing functions, and hence add names and values to this namespace, with AddCustomWindowingFunction.

See also: SIGFUNC, MODFUNC

class Shady.DRAWMODE
This class is an enum container: a namespace of values that can be assigned to the drawMode property of a Stimulus instance. The default drawMode is DRAWMODE.QUAD: that means that each Stimulus is automatically drawn as a rectangle according to its envelopeSize.

Other modes rely on the points property, which contains a sequence of two-dimensional coordinates:

- DRAWMODE.POINTS draws a disconnected dot at each location.
- DRAWMODE.Lines takes two locations at a time and connects each pair with a line segment, disconnected from previous or subsequent line segments.
- DRAWMODE.LINE_STRIP draws line segments continuously from one location to the next, only taking the pen off the paper if it encounters a NaN coordinate.
- DRAWMODE.LINE_LOOP is like LINE_STRIP, but the last point in each group is joined back to the first (where a “group” is delimited by NaNs in the sequence of coordinates).
- DRAWMODE.POLYGON also connects successive locations continuously, and fills the area bounded by the lines thus drawn (a NaN coordinate is a way to delimit multiple polygons)

5.5.2 Shady.Dynamics Sub-module

The Dynamics module contains a number of objects that are designed to perform discretized real-time processing of arbitrary functions.

The most general-purpose object is the Function which is a type of callable that supports arithmetic and other tricks. Various wrapper functions act as factories for Functions configured with specialized behavior (most of them with memory for previous inputs). For example:

- Integral and Derivative perform discrete calculus on their inputs.
- Smoother smooths its input
- Transition provides a transient (self-removing, once complete) dynamic.
• Oscillator: provides sinusoidally varying output

The other major contribution is StateMachine a different class that provides callable instances in which discrete-time state-machine logic can easily be implemented.

Note that everything exported from this module is also available in the top-level Shady.* namespace.

Shady.Dynamics.Clock(startNow=True, speed=1.0)

This function returns a Function wrapped around a simple linear callable that takes one argument t.

**Parameters**

• startNow (bool) – If this is True, the clock’s time zero starts the first time it is called. If it is False, the clock does not subtract any "time zero", but rather just uses the t argument that is passed to it.

• speed (float) – This specifies the speed at which the clock runs—it’s just a multiplier applied to the input value t.

If startNow=True you can get the same effect with \( f = \text{Integral}(\text{speed}) \) but the implementation in Clock() is simpler and hence a little more efficient. Note that, because of this simplicity, ResetTimeBase will not work on this object.

class Shady.Dynamics.Function(*pargs, **kwargs)

A Function instance is a callable wrapper around another callable and/or around a constant. Function objects support arithmetic operators.

\[ f = \text{Function}(\lambda t: t \times 2) + \text{Function}(\lambda t: t) \]

\[ g = \lambda t: t \times 2 + t \]

f and g are both callable objects and both will return the same result when called with a numeric argument. f is of course less efficient than g. However, it allows functions to be defined and built up in a modular way

**Parameters**

• *pargs – Each of the positional arguments is considered to be a separate additive term*. Terms (or their outputs, if callable) are added together when the Function is called.

• **kwargs – If supplied, any optional keyword arguments are passed through to every callable term, whenever the Function is called.

**Tap (initial=None)**

As you transform a Function object, you develop a pipeline of successively applied operations. What if you want to examine values at an intermediate stage along that pipeline? In this case you can Tap() it. The result is a callable object that you can call to obtain the latest value.

Example:

```python
f = Function( lambda t: t )
f *= 2
f += 2
intermediate = f.Tap()
f.Transform( math.sin )

for input_value in [0.1, 0.2, 0.3, 0.4, 0.5]:
    output_value = f( input_value )
    print( ' t = %r ' % input_value )
    print( ' 2 * t + 2 = %r ' % intermediate() )
    print( 'sin(2 * t + 2) = %r ' % output_value )
    print( '' )
```

**Through (any_callable, *additional_pargs, **kwargs)**

Arithmetic operations on a Function build up a chain of operations. For example:

5.5. Shady API Reference 193
\[ f = \text{Function} ( \lambda \ t: t ) \]
\[ f += 1 \quad \# \text{Now } f(t) \text{ will return } t + 1 \]
\[ f *= 2 \quad \# \text{Now } f(t) \text{ will return } 2 * t + 2 \]

The `Transform` method allows arbitrary operations to be added to the chain. For example:

\[
\begin{align*}
    f &. \text{Transform} ( \text{math.sin} ) \\
    \# \text{Now } f(t) \text{ will return } \text{math.sin}(2 * t + 2)
\end{align*}
\]

Any `*additional_pargs` or `**kwargs` are passed through to the transforming function (any_callable).

The `Transform()` method changes the `Function` instance in-place, in a manner analogous to `+=` and `*=`. By contrast, binary operators `*` and `+` return a new `Function` containing deep copies of the original instance’s terms. As `+` is to `+=`, so the global function `Apply()` is to the `Transform()` method:

\[
\begin{align*}
    f & = \text{Function} ( \lambda \ t: t ) \\
    g & = \text{Apply} ( \text{math.cos}, f ) \quad \# \text{creates a separate } '\text{Function}' \text{ instance} \\
    f &. \text{Transform} ( \text{math.sin} ) \quad \# \text{transforms } 'f' \text{ in-place} \\
    \# \text{now } g(t) \text{ returns } \text{math.cos(t)} \text{ and } f(t) \text{ returns } \text{math.sin(t)}
\end{align*}
\]

**Transform** *(any_callable, *additional_pargs, **kwargs)*

Arithmetic operations on a `Function` build up a chain of operations. For example:

\[
\begin{align*}
    f & = \text{Function} ( \lambda \ t: t ) \\
    f & += 1 \quad \# \text{Now } f(t) \text{ will return } t + 1 \\
    f & *= 2 \quad \# \text{Now } f(t) \text{ will return } 2 * t + 2
\end{align*}
\]

The `Transform` method allows arbitrary operations to be added to the chain. For example:

\[
\begin{align*}
    f &. \text{Transform} ( \text{math.sin} ) \\
    \# \text{Now } f(t) \text{ will return } \text{math.sin}(2 * t + 2)
\end{align*}
\]

Any `*additional_pargs` or `**kwargs` are passed through to the transforming function (any_callable).

The `Transform()` method changes the `Function` instance in-place, in a manner analogous to `+=` and `*=`. By contrast, binary operators `*` and `+` return a new `Function` containing deep copies of the original instance’s terms. As `+` is to `+=`, so the global function `Apply()` is to the `Transform()` method:

\[
\begin{align*}
    f & = \text{Function} ( \lambda \ t: t ) \\
    g & = \text{Apply} ( \text{math.cos}, f ) \quad \# \text{creates a separate } '\text{Function}' \text{ instance} \\
    f &. \text{Transform} ( \text{math.sin} ) \quad \# \text{transforms } 'f' \text{ in-place} \\
    \# \text{now } g(t) \text{ returns } \text{math.cos(t)} \text{ and } f(t) \text{ returns } \text{math.sin(t)}
\end{align*}
\]

**Watch** *(conditional, *additional_pargs, **kwargs)*

Adds a watch condition on the output of a `Function`.

**Parameters**

- **conditional** *(callable)* — This should be a callable whose first input argument is a float, int or numeric numpy array. The return value can be:
  - `None`, in which case nothing happens
  - a dict `d`, in which case the `Function` will raise an `Abort` exception (a subclass of `StopIteration`) containing `d` as the exception’s argument. The `Function` itself
Shady Documentation, Release 1.9.1

will not perform any further processing.

– a numeric value (or numeric array) x, in which case the Function will continue to process the numeric value but, at the very last step in the chain, it will raise a Stop exception (a subclass of StopIteration) containing the final processed value.

• *additional_pargs – If additional positional arguments are supplied, they are simply passed through to conditional.
• **kwargs – If additional keyword arguments are supplied, they are simply passed through to conditional.

Some frameworks (e.g. Shady.PropertyManagement) will automatically catch and deal with StopIteration exceptions appropriately, but if you need to do so manually, you can do so as follows:

```
try:
    y = f(t)
except StopIteration as exc:
    info = exc.args[0]
    if not isinstance(info, dict):
        terminal_value_of_y = info
```

Shady.Dynamics.Integral(*pargs, **kwargs)

Returns a specially-configured Function. Like the Function constructor, the terms wrapped by this call may be numeric constants, and/or callables that take a single numeric argument t. And like any Function instance, the instance returned by Integral is itself a callable object that can be called with t.

Unlike a vanilla Function, however, an Integral has memory for values of t on which it has previously been called, and returns the cumulative area under the sum of its wrapped terms, estimated discretely via the trapezium rule at the distinct values of t for which the object is called.

Like any Function, it can interact with other Functions, with other single-argument callables, with numeric constants, and with numeric numpy objects via the standard arithmetic operators +, -, /, *, **, and %. and may also have other functions applied to its output via Apply.

Integral may naturally be take another Integral output as its input, or indeed any other type of Function.

Example - prints samples from the quadratic $\frac{1}{2}t^2 + 100$:

```
g = Integral( lambda t: t ) + 100.0
print( g(0) )
print( g(0.5) )
print( g(1.0) )
print( g(1.5) )
print( g(2.0) )
```

Shady.Dynamics.Derivative(*pargs, **kwargs)

Like Integral(), but configures its Function to perform discrete-time differentiation instead of integration.

Shady.Dynamics.ResetTimeBase(x)

This method can be applied to Function and StateMachine instances. It will run through the terms of the Function (and recursively through all the terms of any terms that are themselves Function’s) looking for dynamic objects that have memory: `StateMachine, and Function wrappers produced by Integral, Derivative, TimeOut, Transition or Smoother. In any of these cases, it erases their memory of previous calls. They (and hence the Function as a whole) will consider the next t value they receive to be “time zero”.

5.5. Shady API Reference 195
Shady.Dynamics.Impulse (magnitude=1.0, autostop=True)
This function constructs a very simple specially-configured Function instance, which will return magnitude the first time it is called (or when called again with the same t argument as its first call) and then return 0.0 if called with any other value of t.

Shady.Dynamics.Smoother (arg=None, sigma=1.0, exponent='EWA')
This function constructs a Function instance that smooths, with respect to time, the numeric output of whatever callable object it wraps. You could test this by wrapping the output of Impulse() with it.

Parameters
- arg – A Function instance or any other callable that returns a numeric output.
- sigma – Interpreted as the sigma (width) parameter of a Gaussian if exponent is 2.0 (or of the comparable exponential-family function if it is some other positive numeric value). If exponent is not numeric, sigma is interpreted as the half-life of an exponential-weighted-average (EWA) smoother.
- exponent – If this is None or the string 'EWA' then the Smoother uses exponential weighted averaging, with sigma as its half-life. Alternatively, if this is a positive floating-point value, it is treated as the exponent of an exponential-family function for generating finite-impulse-response weights, with sigma as the time-scale parameter. exponent=2.0 gets you Gaussian-weighted FIR coefficients.

Returns A Function instance.

Shady.Dynamics.Sinusoid (cycles, phase_deg=0)
Who enjoys typing 2.0 * numpy.pi * over and over again? This is a wrapper around numpy.sin (or math.sin if numpy is not installed) which returns a sine function of an argument expressed in cycles (0 to 1 around the circle). Heterogeneously, but hopefully intuitively, the optional phase-offset argument is expressed in degrees. If numpy is installed, either argument may be non-scalar (phase_deg=[90,0] is useful for converting an angle into 2-D Cartesian coordinates).

This is a function, but not a Function. You may be interested in Oscillator, which returns a Function wrapper around this.

Shady.Dynamics.Oscillator (freq, phase_deg=0.0)
Returns a Function object with an output that oscillates sinusoidally as a function of time: the result of Apply()-ing Sinusoid to an Integral.

Shady.Dynamics.Transition (start=0.0, end=1.0, duration=1.0, delay=0.0, transform=None, finish=None)
This is a self-stopping dynamic. It uses a Function.Watch() call to ensure that, when the dynamic reaches its end value, a Stop exception (a subclass of StopIteration) is raised. Some frameworks (e.g. Shady.PropertyManagement) will automatically catch and deal with StopIteration exceptions.

Parameters
- start (float, int or numeric numpy.array) – initial value
- end (float, int or numeric numpy.array) – terminal value
- duration (float or int) – duration of the transition, in seconds
- delay (float or int) – delay before the start of the transition, in seconds
- transform (callable) – an optional single-argument function that takes in numeric values in the domain [0, 1] inclusive, and outputs numeric values. If you want the final output to scale correctly between start and end, then the output range of transform should also be [0, 1].
- finish (callable) – an optional zero-argument function that is called when the transition terminates
Example:

```python
from Shady import World, Transition, RaisedCosine, Hann

w = World( canvas=True, gamma=-1 )
gabor = w.Sine( pp=0, atmosphere=w )

@w.EventHandler( slot=-1 )
def ControlContrast( self, event ):
    if event.type == 'key_release' and event.key == 'r':
        gabor.contrast = Transition( transform=RaisedCosine )
    if event.type == 'key_release' and event.key == 'f':
        gabor.contrast = Transition( 1, 0, transform=RaisedCosine )
    if event.type == 'key_release' and event.key == 'h':
        gabor.contrast = Transition( duration=2, transform=Hann )

# press 'f' to see contrast fall from 1 to 0 using a raised-cosine
# press 'r' to see contrast rise from 0 to 1 using a raised-cosine
# press 'h' to see contrast rise and fall using a Hann window in time
```

Shady.Dynamics.RaisedCosine(x)
Maps a linear ramp from 0 to 1 onto a raised-cosine rise from 0 to 1. Half a Hann window.

Shady.Dynamics.Timeout (func, duration)
Returns a wrapped version of callable func that raises a Stop exception when called with a t argument larger than the very first t argument it receives plus duration.

Shady.Dynamics.Sequence (container)
Returns a Function object whose value is defined piecewise by the elements of container. The container may be:

- a dict whose keys are numbers: the keys are interpreted as time-points relative to the first time the Function is called, and they dictate the times at which the Function output should switch to the corresponding value. If any value is itself a callable object, then the overall Function output is computed by calling it, with the relative time-since-first call as its single argument.

- any other iterable: the items are then simply handled in turn, each time the Function is called with a new value for the time t argument. If any item is itself a callable object, then the overall Function output is computed by calling it, with the relative time-since-first call as its single argument—and, we do not advance to the next item until that item has raised a StopIteration exception. This allows you to chain Transition() function objects together—e.g.:

  ```python
  Sequence( [ Transition( 0, 100 ), Transition( 100, 0 ) ] )
  ```

You can also use the constant STITCH to ensure that the terminal value from the preceding callable is ignored—so in the following example, the value 100 would not be repeated:

```python
Sequence( [ Transition( 0, 100 ), STITCH, Transition( 100, 0 ) ] )
```

See also: CallOnce, WaitFor, TimeOut, STITCH

Shady.Dynamics.Forever
alias of itertools.count

Shady.Dynamics.CallOnce (func)
Can be used in a Sequence to create a side effect. For example:

```python
def Foo():
    print( 'The sequence has ended' )
```

(continues on next page)
```python
f = Sequence([ Transition(0, 1), Transition(1, 0), CallOnce(Foo) ])
import numpy
for t in numpy.arange(0, 3, 0.01): print(f(t))
```

See also: `Sequence, WaitUntil, TimeOut, STITCH`

**Shady.Dynamics.WaitUntil(func, ongoingValue=None, finalValue=None)**

Can be used in a `Sequence` to hold until a certain condition is fulfilled (signalled by `func()` returning a truthy value).

See also: `Sequence, CallOnce, TimeOut, STITCH`

**Shady.Dynamics.Apply(any_callable, f, *additional_pargs, **kwargs)**

```python
g = Apply(some_function, f)  # `f` is a `Function` instance
# and now so is `g`
g = copy.deepcopy(f).Transform(some_function)
```

In both cases, `some_function()` is applied to transform the output that `f` would otherwise have given, but whereas the `Transform()` method actually alters the original instance `f`, `Apply` creates a new `Function` instance and leaves the original `f` untouched.

See also: `Function.Transform`

**class Shady.Dynamics.StateMachine(*states)**

This class encapsulates a discrete-time state machine. Instances of this class are callable, with a single argument `t` for time.

You can call a `StateMachine` instance multiple times with the same value of `t`. Its logic will only run when `t` increases.

Example:

```python
sm = StateMachine()
def PrintName(state):
    print(state.name)
sm.AddState('First', next='Second', duration=1.0, onset=PrintName)
sm.AddState('Second', next='First', duration=2.0, onset=PrintName)
import time
while True:
    time.sleep(0.1)
    sm(time.time())
```

The magic of a `StateMachine` depends on how the states are defined. This can be done in a number of ways, the most powerful of which is to define each state as a subclass of `StateMachine.State`.

See the `AddState()` method for more details.

**AddState(state, duration=None, next=Unspecified, onset=None, ongoing=None, offset=None)**

Add a new state definition to a `StateMachine`.

**Parameters**
- **state** – This can be a string, defining the name of a new state. Or it can be a class that inherits from StateMachine.State. Or it can be an actual instance of such a StateMachine.State subclass.

- **duration** – A numeric constant, or None, or a callable that returns either a numeric constant or None. Determines the default duration of the state (None means indefinite).

- **next** – A string, or a callable that returns a string, specifying the state to change to when the duration elapses, or when ChangeState() is called without specifying a destination. May also be None because None is a legal state for a StateMachine to be in. Or it can be left entirely Unspecified in which case it means “the next state, if any, that I add to this StateMachine with AddState()”.

- **onset** – Either None, or a callable routine that will get called whenever we enter this state.

- **ongoing** – Either None, or a callable routine that will get called whenever the StateMachine is called and we are currently in the state. If the callable ongoing() returns a string, the state machine will immediately attempt to change to the state named by that string.

- **offset** – Either None, or a callable routine that will get called whenever we leave this state.

**duration**, **next**, **onset**, **ongoing**, **offset** may be constants, callables that take no arguments, or callables that take one argument. If they accept an argument, that argument will be an instance of StateMachine.State. This means they are effectively methods of your State, and indeed can be defined that way if you prefer.

Since it is legal for the **state** argument to be a class definition, and for all the other arguments to be defined as attributes or methods of that class (instead of as arguments to this method), one valid way to use the **AddState** method is as a class decorator:

```python
import random

sm = StateMachine()

@sm.AddState
class First( StateMachine.State ) :
    next = 'Second'
    duration = 1.0
    def onset( self ) :
        print( self.name )

@sm.AddState
class Second( StateMachine.State ) :
    next = 'First'
    def onset( self ) :
        print( self.name )
    def duration( self ) :
        return random.uniform( 1, 5 )
    def ongoing( self ) :
        if self.elapsed > 3.0:
            print( 'we apologize for the delay...' )
```

Equivalently, you also can do:

```python
class First( StateMachine.State ) :
    ...

class Second( StateMachine.State ) :
    ...

sm = StateMachine( First, Second )
```
Equivalent state-machines can be defined without using the object-oriented class-definition approach: the syntax is simpler for very simple cases, but quickly explodes in complexity for more sophisticated machines. Here is an example, simplified relative to the above:

```python
sm = StateMachine()
def PrintName( state ):
    print( state.name )
sm.AddState( 'First', next='Second', duration=1.0, onset=PrintName )
sm.AddState( 'Second', next='First', duration=2.0, onset=PrintName )
```

**ChangeState** *(newState=StateMachine.NEXT, timeOfChange=StateMachine.PENDING)*

This method manually requests a change of state, the next time the StateMachine is called with a new time t value.

**Parameters**

- **newState** *(str)* – If omitted, change to the state dictated by the current state’s next attribute/method. Otherwise, attempt to change to the state named by this argument.

- **timeOfChange** *(float)* – This is used internally. To ensure accuracy, you should not specify this yourself. When called from outside the stack of an ongoing StateMachine call, this method actually only requests a state change, and the change itself will happen on, and be timed according to, the next occasion on which the the StateMachine instance is called with a novel time t value.

**Elapsed** *(t=None, origin='total')*

Return the amount of time elapsed at time t, as measured either from the very first call to the StateMachine(origin='total') or from the most recent state change (origin='current').

If t is None, then the method returns the time elapsed at the most recent call to the StateMachine, a result that can also be obtained from two special properties:

- `self.elapsed_total` same as `self.Elapsed(t=None, origin='total')`

- `self.elapsed_current` same as `self.Elapsed(t=None, origin='current')`

### 5.5.3 Shady.Linearization Sub-module

This module contains several utility functions related to linearization.

Some of these *(ScreenNonlinearity, Linearize, ApplyLUT)* are only useful in special circumstances where we want to re-create “offline” what the shader is doing for us on every frame.

Others *(LoadLUT, SaveLUT)* are for general management of look-up table arrays.

Others *(BitStealingLUT, ReportBitStealingStats)* are useful for creating and examining a specific type of look-up table that employs a bit-stealing technique (after Tyler 1997).

Note that everything exported from this module is also available in the top-level Shady.* namespace.

**Shady.Linearization.ScreenNonlinearity** *(x, gamma='sRGB')*

Maps normalized DAC values x (in the domain 0 to 1, which corresponds to DAC values 0 to 255 if we assume standard 8-bit DACs) to ideal luminance Y (in the range 0 to 1), given the screen non-linearity parameter gamma.

Generally, gamma is numeric and strictly positive, in which case the relationship is $Y = x^{\gamma}$, which is also used if you pass a gamma value of 0 or less: this uses a slightly different piecewise equation, very close to the gamma=2.2 curve (even though the exponent used in it is actually 2.4).

Inverse of **Linearize()**
Shady.Linearization.Linearize\((Y, \gamma='sRGB')\)
Maps ideal luminance \(Y\) (in the domain 0 to 1) to normalized DAC values \(x\) (in the range 0 to 1, which corresponds to DAC values 0 to 255 if we assume standard 8-bit DACs) given the screen non-linearity parameter \(\gamma\).

Generally, \(\gamma\) is numeric and strictly positive, in which case the relationship is \(x = Y^{(1/\gamma)}\). A special case is \(\gamma='sRGB'\), which is also used if you pass a \(\gamma\) value of 0 or less: this uses a slightly different piecewise equation, very close to the \(\gamma=2.2\) curve (even though the exponent used in it is actually 2.4).

This allows us to emulate, on the CPU and in Python, the linearization operation performed automatically by the GPU in the fragment shader on every frame according to the value of of the Stimulus.gamma property.

Inverse of ScreenNonlinearity()

Shady.Linearization.LoadLUT\((source, DACbits=8)\)
Load or prepare a look-up table array.

Parameters

- **source** (str, :class:`~numpy.ndarray`) – Usually this is a string denoting a filename. The file may be in numpy format - either a :func:`numpy.save` file in which the look-up-table has been written with :func:`numpy.save`, or a :func:`numpy.savez` file into which the look-up table array has been saved with :func:`numpy.savez` as either the sole variable or a variable called :func:`lut`. If the third-party pillow package is installed, the file may alternatively be a :func:`png` image file in which look-up table entries have been saved as R,G,B pixel values in column-first order.

  The source may also be a :func:`numpy.ndarray` already.

- **DACbits** (int) – This is the number of bits per DAC in the graphics card for which the look-up table is intended. In this function it is used to verify that the lookup-table entries are in range and to cast the output as the appropriate numeric data-type.

Returns Whether source is a filename or an array already, in either case, this function ensures that the returned result is a 3-dimensional :class:`~numpy.ndarray`, of the appropriate integer type, with extent 3 (RGB) or 4 (RGBA) in its third dimension.

See also:

- **SaveLUT()**

Shady.Linearization.SaveLUT\(\text{filename, lut, luminance=(), DACbits=8}\)
Save a look-up table array, and optionally also the corresponding sequence of luminance values, in a file.

Parameters

- **filename** (str) – name of the file to save. Determines the file format, and should end in :func:`numpy.save`, :func:`numpy.savez` or :func:`png`

- **lut** (:class:`~numpy.ndarray`) – look-up table array, \(n\)-by-3 or \(n\)-by-1-by-3 as per the output of :func:`LoadLUT()` or :func:`BitStealingLUT()`, where \(n\) is the number of entries. If the format is :func:`numpy.savez` the array will be saved under the variable name :func:`lut`.

- **luminance** (:class:`~numpy.ndarray`, list) – sequence of \(n\) ideal luminance values (i.e. luminance values in the range 0 to 1) corresponding to each of the look-up table entries. Only saved (under the variable name :func:`luminance`) if the file format is :func:`numpy.savez`

- **DACbits** (int) – This is the number of bits per DAC in the graphics card for which the look-up table is intended. In this function it is used to verify that the lookup-table entries are in range and to cast the output as the appropriate numeric data-type.

Returns
• the filename
• the look-up table array in standardized format
• the sequence of luminance values

Return type  a tuple consisting of

See also:
• LoadLUT()

Shady.Linearization.ApplyLUT(image, lut, noiseAmplitude=0, DACbits=8)
Translate an array of image pixel values via a look-up table lut.
This allows us to emulate, on the CPU in Python, the look-up operation performed automatically by the GPU in the shader on every frame if a look-up table has been configured via the Stimulus.lut property.

Parameters

• image (numpy.array) – Source pixel values. If the array data type is floating-point, pixel values are assumed to refer to ideal luminances in the range 0 to 1, and are clipped to this range before scaling and rounding according to the size of the lut. If the array is of some integer type, the values are assumed to be direct indices into the look-up table.
  
  Note that, if the image has more than one color channel (i.e. it has a third dimension with extent > 1) then only the first channel (red) will be used.
• lut (numpy.array) – Look-up table array, e.g. as output by LoadLUT() or BitStealingLUT().
• noiseAmplitude (float) – specifies the amplitude of an optional random signal that can be added to image pixel values before lookup. A negative value indicates a uniform distribution (in the range [noiseAmplitude, -noiseAmplitude]) whereas a positive value is interpreted as the standard deviation of a Gaussian noise distribution.
• DACbits (int) – The number of bits per digital-analog converter in the graphics card for which the look-up table is intended. Floating-point image pixel values will be scaled accordingly.

Returns An array of integer DAC values post-lookup. First two dimensions match those of the input image. Extent in the third dimension will match that of lut.

Shady.Linearization.BitStealingLUT(maxDACDeparture=2, Cmax=3.0, nbits=16, gamma='sRGB', cache_dir=None, DACbits=8)
Create an RGB look-up table that (a) linearizes pixel intensity values according to the specified gamma profile, and (b) increases dynamic range using a “bit-stealing” approach (after Tyler 1997).

Parameters

• maxDACDeparture (int) – Red, green and blue DAC values will be considered only up to +/- this value relative to the R==G==B line
• nbits (int) – The look-up table will have 2 ** nbits entries. It doesn’t hurt to specify a high number here, like 16—however, note that, depending on the values of maxDACDeparture and Cmax you may not get that many distinct or evenly-spaced luminance levels. The actual effective precision can be investigated using ReportBitStealingStats()
• Cmax (float) – [R, G, B] triples will not be considered if the corresponding chroma, in percent (i.e. the third column of RGB_to_YLChab() output) exceeds this.
• gamma (float) – screen non-linearity parameter (see ScreenNonlinearity())
• `cache_dir (str, None)` – optional directory in which to look for a cached copy of the resulting LUT (or save one, after creation, if the appropriately-named file was not found there).

`DACbits (int)`: The number of bits per digital-analog converter in the graphics card for which the look-up table is intended. LUT values will be scaled accordingly.

Returns `numpy` array of shape `[2**nbits, 1, 3]` with the appropriate integer type (usually `uint8`), containing integer RGB triplets.

`Shady.Linearization.ReportBitStealingStats(lut=None, image=None, gamma='sRGB', DACbits=8)`

Prints to the console certain statistics about the look-up table `lut`, if supplied, and optionally also any given `image` that has been through the look-up process (i.e. an output of `ApplyLUT()`).

Make sure that the `gamma` and `DACbits` arguments match the values that were used for creating `lut`.

### 5.5.4 Shady.Contrast Sub-module

This module contains various optional (but useful) utility functions that relate specifically to computation and conversion of contrast and luminance values.

Note that everything exported from this module is also available in the top-level `Shady.*` namespace.

Brief definitions for terms used in this submodule are as follows, but for full details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.

**Ideal luminance**: a value from 0 (corresponding to the screen’s minimum luminance) to 1 (corresponding to the screen’s maximum luminance).

**Physical luminance**: a true physical measure of screen luminance, including the effect of reflected ambient light. Will not ever reach 0 unless your setup is very expensive...

**Ideal contrast ratio**: a contrast ratio (RMS or Michelson) computed from ideal luminance values.

**Physical contrast ratio**: a contrast ratio (RMS or Michelson) computed from physical luminance values.

**Normalized contrast**: not a contrast ratio, but rather a scaling factor from 0 to 1 that is applied to a signal’s amplitude, such that a normalized contrast of 1 allows the signal to fill the screen’s entire luminance range.

`Shady.Contrast.RMSContrastRatio(pixels, background=None)`

Compute the root-mean-square contrast ratio of a pixel array. \( \sqrt{\frac{1}{N} \sum_{x} \sum_{y} (L_{xy} - L_{\mu})^2} \), or in plain text:

\[
\frac{\text{(pixels - background) \^ 2}.mean() \times 0.5}{\text{background}}
\]

where `background (= L_{\mu})` defaults to `pixels.mean()`.

If `pixels` is a three-dimensional array, two preprocessing steps are performed: first, if its extent in the third dimension is 2 or 4, the last layer is assumed to be an alpha channel and is excluded from the computation; second, “luminance” is computed as the mean-across-(remaining)-layers. In reality, red, green and blue channels will not contribute equally to luminance, so for more accurate results, you may wish to perform your own computations to convert the array into a single-channel array of luminances before calling this function.

This function will work on any kind of luminance scale: if you feed it ideal luminances (in the 0 to 1 range), you will get an “ideal” contrast ratio. If you feed it physical luminances (in candelas / m^2) you will get a physical contrast ratio. For more details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.

See also:

• `MichelsonContrastRatio()`
Compute the Michelson contrast ratio of a pixel array, \( \frac{L_{\max} - L_{\min}}{L_{\max} + L_{\min}} \), or in plain text:

\[
\frac{\text{pixels}.max() - \text{pixels}.min()}{\text{pixels}.max() + \text{pixels}.min()}
\]

The input argument `background` is unused but it is included in the prototype for compatibility with `RMSContrastRatio()`.

If `pixels` is a three-dimensional array, two preprocessing steps are performed: first, if its extent in the third dimension is 2 or 4, the last layer is assumed to be an alpha channel and is excluded from the computation; second, “luminance” is computed as the mean-across-(remaining)-layers. In reality, red, green and blue channels will not contribute equally to luminance, so for more accurate results, you may wish to perform your own computations to convert the array into a single-channel array of luminances before calling this function.

This function will work on any kind of luminance scale: if you feed it ideal luminances (in the 0 to 1 range), you will get an “ideal” contrast ratio. If you feed it physical luminances (in candelas / m^2) you will get a physical contrast ratio. For more details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.

See also:

- `RMSContrastRatio()`
- `IdealToPhysicalContrastRatio()`
- `PhysicalToIdealContrastRatio()`

Convert ideal luminance (from 0 to 1) to physical luminance. For more details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.

**Parameters**

- `idealLuminance` – “ideal” luminance value(s) to convert (in the range 0 to 1)
- `physicalScreenRange` – sequence of [min, max] physical luminance values measured from the screen with a photometer

**Returns** Physical luminance value(s) corresponding to the ideal input value(s).

Convert physical luminance to ideal luminance (from 0 to 1). For more details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.

**Parameters**

- `physicalLuminance` – physical luminance value(s) to convert
- `physicalScreenRange` – sequence of [min, max] physical luminance values measured from the screen with a photometer

**Returns** Ideal luminance value(s) (in the range 0 to 1) corresponding to the physical input value(s).

Convert one or more ideal contrast ratios to physical contrast ratios. For more details, see `Luminance and Contrast` in the online documentation or the docstring of `Shady.Documentation.LuminanceAndContrast`.
Parameters

- **idealContrastRatio** – “ideal” contrast ratio(s) to convert
- **idealBackgroundLuminance** – “ideal” background luminance (scalar, in the range 0 to 1)
- **physicalScreenRange** – sequence of [min, max] physical luminance values measured from the screen with a photometer

**Returns**
Physical contrast ratio(s) corresponding to the ideal input value(s).

**Shady.Contrast.PhysicalToIdealContrastRatio**(*physicalContrast*, *idealBackgroundLuminance*, *physicalScreenRange*)
Convert one or more physical contrast ratios to ideal contrast ratios. For more details, see *Luminance and Contrast* in the online documentation or the docstring of *Shady.Documentation*. LuminanceAndContrast.

Parameters

- **physicalContrastRatio** – physical contrast ratio(s) to convert
- **idealBackgroundLuminance** – “ideal” background luminance (scalar, in the range 0 to 1)
- **physicalScreenRange** – sequence of [min, max] physical luminance values measured from the screen with a photometer

**Returns**
Ideal contrast ratio(s) corresponding to the physical input ratio(s).

**Shady.Contrast.IdealContrastRatioToNormalizedContrast**(*idealContrastRatio*, *idealBackgroundLuminance*)
Convert one or more ideal contrast ratios to normalized contrast scaling factors. For more details, see *Luminance and Contrast* in the online documentation or the docstring of *Shady.Documentation*. LuminanceAndContrast.

Parameters

- **idealContrastRatio** – “ideal” contrast ratio(s) to convert
- **idealBackgroundLuminance** – “ideal” background luminance (scalar, in the range 0 to 1)

**Returns**
Normalized contrast scaling factors(s) corresponding to the input ideal contrast ratio(s).

**Shady.Contrast.NormalizedContrastToIdealContrastRatio**(*normalizedContrast*, *idealBackgroundLuminance*)
Convert one or more normalized contrast values to ideal contrast ratios. For more details, see *Luminance and Contrast* in the online documentation or the docstring of *Shady.Documentation*. LuminanceAndContrast.

Parameters

- **normalizedContrast** – normalized contrast scaling factor(s) (in the range 0 to 1) to convert
- **idealBackgroundLuminance** – “ideal” background luminance (scalar, in the range 0 to 1)

**Returns**
Ideal contrast ratio(s) corresponding to the input normalized contrast scaling factor(s).

### 5.5.5 Shady.Utilities Sub-module

This module contains various optional (but useful) utility functions.

Note that everything exported from this module is also available in the top-level *Shady.* namespace.

**Shady.Utilities.Tukey**(*x*, *rise=0.5*, *plateau=0.0*, *fall='rise'* , *start=0.0*)
This is a one-dimensional windowing function consisting of a raised- cosine rise from zero, an optional plateau,
and a raised-cosine fall back to zero. It is a generalization of the Hann function. It requires the \texttt{numpy} package to be installed.

**Parameters**

- \(x\) – The input argument to the function, in units of space or time (let’s assume it’s time from here on).
- \(\text{rise}\) – In the same units as \(x\), the amount of time the function takes to rise from 0.0 to 1.0.
- \(\text{plateau}\) – In the same units as \(x\), the amount of time for which the function remains at 1.0 between the rising and falling phases. The default is 0, which means this function returns a simple Hann or raised-cosine function by default.
- \(\text{fall}\) – In the same units as \(x\), the amount of time the function takes to fall from 1.0 back to 0.0. Alternatively, the string ‘\text{rise}\' may be used (and it is the default) which makes the duration of the fall match the duration of the rise.
- \(\text{start}\) – In the same units as \(x\), the initial amount of time for which the function remains at 0.0 before starting to rise. Defaults to 0.

**Returns** The numeric output of the window function (as a \texttt{float} if the input \(x\) was a scalar numeric; as a \texttt{numpy.array} if \(x\) was a sequence or array of numbers).

```python
Shady.Utilities.Hann(x, rise=0.5, start=0.0)
```
A Hann window (raised cosine) function of \(x\). Wraps around the more general \texttt{Tukey()} function but ensures no plateau, and equal lengths of rise and fall.

```python
Shady.Utilities.EllipticalTukeyWindow(size, plateauProportion=0.0)
```
This function constructs a two-dimensional \texttt{numpy.array} of the specified \texttt{size} containing a discretely sampled one- or two-dimensional spatial windowing function. The function has an optional plateau in the center, outside of which it falls off to 0 according to a raised cosine profile. For two-dimensional windows, all contours of the array are elliptical and have the same aspect ratio as the specified \texttt{size}.

This function replicates the formula that is used for windowing on the GPU, as per the fragment shader program at the heart of Shady.

**Parameters**

- \(\text{size}\) – May be an integer (dictating a square output), a sequence of two integers [\texttt{width}, \texttt{height}], a \texttt{World} or \texttt{Stimulus} instance, or an existing 2- or 3-D \texttt{numpy.array} to use as a template (i.e. one whose first two dimensions are [\texttt{height}, \texttt{width}]).
- \(\text{plateauProportion}\) – This is either a single floating-point number, or a sequence of two numbers corresponding to the horizontal and vertical dimensions. The numbers specify the linear proportion of the cross section of the window function that is at maximum. They can be in the range 0.0 (no plateau => radial Hann window) to 1.0 (all plateau, no skirt => sharp-edged circle or ellipse). Alternatively, negative values can be used to disable windowing (so, for example, \([0.0, -1.0]\) specifies Hann windowing in the horizontal dimension only, and no windowing in the vertical).

**Returns** A two-dimensional \texttt{numpy.array} containing image pixel multipliers, in the range \([0.0, 1.0]\).

```python
Shady.Utilities.LinearizationTestPattern(canvasSize=(1920, 1080), patchSize=(50, 50), amplitudes=(0.95, 0.75, 0.5, 0.25), plateauProportion=0.85, meanZero=True)
```
Construct a 2-dimensional \texttt{numpy.array} containing a special linearization pattern. Blank patches are interspersed in checkerboard fashion with textured patches. Textured patches contain horizontal stripes, vertical stripes, and checkerboard patterns, all of single-pixel granularity. Textured patches vary in contrast. When the screen (or stimulus) is perfectly linearized, textured patches of all contrasts should be indistinguishable from blank patches when viewed from a sufficient distance (or with sufficiently bad uncorrected vision).
Parameters

- **canvasSize** – may be an int (resulting in a square stimulus), or a sequence of two numbers (width, height). May also be a Shady.World instance, Shady.Stimulus instance, a PIL image, or an image represented as a 2- or 3-dimensional numpy.array - in these cases the dimensions of the output match the dimensions of the instance.

- **patchSize** (sequence of 2 ints) – dimensions, in pixels, of the individual patches that make up the pattern in checkerboard fashion.

- **amplitudes** (tuple of floats) – 0.0 means blank, 1.0 means full contrast.

- **plateauProportion** (float) – governs the one-dimensional raised-cosine fading at the edges of striped patches. With plateauProportion=1, bright and dark edge artifacts tend to be visible.

- **meanZero** (bool) – if True, then the mean of the output array is 0 and its range is \([-1, 1]\) when amplitude=1. If False, then the mean of the output array is 0.5 and the range is \([0, 1]\) when amplitude=1.

**Returns** A two-dimensional numpy.array containing the image pixel values.

Examples:

```python
world.Stimulus( LinearizationTestPattern( world ) * 0.5 + 0.5 )
world.Stimulus( LinearizationTestPattern( world, meanZero=False ) ) # same as above
```

Shady.Utilities.CheckPattern (canvasSize=(1920, 1080), amplitude=1.0, checkSize=1, meanZero=True)

Return a 2-dimensional numpy.array containing a checkerboard pattern.

Parameters

- **canvasSize** – may be an int (resulting in a square stimulus), or a sequence of two numbers (width, height). May also be a Shady.World instance, Shady.Stimulus instance, a PIL image, or an image represented as a 2- or 3-dimensional numpy.array - in these cases the dimensions of the output match the dimensions of the instance.

- **amplitude** (float) – 0.0 means blank, 1.0 means full contrast.

- **checkSize** (int) – size, in pixels, of the individual light and dark squares making up the checkerboard pattern.

- **meanZero** (bool) – if True, then the mean of the output array is 0 and its range is \([-1, 1]\) when amplitude=1. If False, then the mean of the output array is 0.5 and the range is \([0, 1]\) when amplitude=1.

**Returns** A two-dimensional numpy.array containing the image pixel values.

Examples:

```python
world.Stimulus( CheckPattern( world ) * 0.5 + 0.5 )
world.Stimulus( CheckPattern( world, meanZero=False ) ) # same as above
```

Shady.Utilities.RelativeLocation (loc, obj, anchor=None, normalized=False)

Translate a two-dimensional location loc...

**From:** World coordinates (i.e. pixels relative to the World’s current anchor position)

**To:** coordinates relative to the current xy position of a Stimulus instance in the same World, or (optionally) relative to a different anchor position of the Stimulus instance.

5.5. Shady API Reference
Parameters

- **loc** – input coordinates, in pixels (single scalar or sequence of two scalars)
- **obj** – `Shady.Stimulus` or `Shady.World` instance
- **anchor** – By default, the return value is expressed as an offset in pixels relative to the anchor position of `obj` (so, if `obj` is a `Stimulus`, the function just subtracts `obj.xy`; or if `obj` is the `World`, the output value will be equal to the input value). However, if you specify an explicit anchor (2 numbers each in the range \([-1, +1]\)) then the return value will be re-expressed as a pixel offset relative to some other part of `obj`. For example, with `anchor=[-1,-1]` you would get pixel coordinates relative to the bottom left corner of `obj`, or with `anchor=[0,+1]` you get pixel coordinates relative to the middle of the top edge of `obj`. The anchor argument is ignored if you specify `normalized=True`.
- **normalized** (bool) – If `False`, return 2-D coordinates in pixels. If `True`, ignore anchor and return 2-D coordinates in the normalized coordinate system of `obj`. This effectively makes this function the inverse of `obj.Place()`.

Returns [x, y] coordinates, in pixels or normalized units depending on the `normalized` argument.

See also:

- `Stimulus.Place()`
- `World.Place()`

**Shady.Utilities.FindGamma** *(world, finish=None, xBlue=True, text=False, **kwargs)*

Interactively use a `LinearizationTestPattern` to estimate the correct gamma.

Blank patches are interspersed in checkerboard fashion with textured patches. Textured patches contain horizontal stripes, vertical stripes, and checkerboard patterns, all of single-pixel granularity. Textured patches vary in contrast. When the screen (or stimulus) is perfectly linearized, textured patches of all contrasts should be indistinguishable from blank patches when viewed from a sufficient distance (or with sufficiently bad uncorrected vision).

The interactive component works best with touch-screens, but you can use the mouse. Up-down movement changes the overall gamma. With the option `xBlue=True`, left-right movement will also adjust the “color temperature” by varying the blue gamma relative to the others (without this, even when the overall gamma is neither too high nor too low, the textured patches can look yellower than the blank patches on some screens).

Supply an optional callback as `finish`. When you press the return key, the adjustment procedure will end and `finish( gamma )` will be called with the final empirical gamma values. If you press the escape key instead, the adjustment will end by calling `finish( None )` instead. Press any letter key to report the current gamma setting on the console.

**Shady.Utilities.PixelRuler** *(base, steps=((10, (0, 0, 1)), (100, (1, 0, 0)), (1000, (0, 1, 0))), alpha=None, topDown=False, world=None, oscillateGamma=False)*

If `base` is an image size specification, or a `World` instance, create a 90%-contrast gray `CheckPattern()` of the appropriate size, to use as a base. Alternatively, `base` may be a ready-made image.

Draw grid lines over the base: the first pixel (pixel 0) is not colored. The 10th, 20th, 30th, . . . pixels (i.e. pixels 9, 19, 29, . . .) are blue. Similarly every 100th pixel is red, and every 1000th pixel is green.

The `topDown` argument changes the definition of “first pixel”: the default is `topDown=False`, which means the first pixel is the bottom row and we work upwards, just like Shady’s normal coordinate system. But if you specify `topDown=True`, the first pixel is on the top row and we work downwards (the way one would index rows of a matrix).
If `base` is a `World` instance, or if `world` is supplied as a separate argument, render the pattern at a depth of \( z=0.9 \) and return the corresponding `Stimulus` instance. Otherwise just return the texture as a `numpy` array.

When rendering as a stimulus, the `oscillateGamma` argument may be useful. Non-zero values cause the stimulus `gamma` property to oscillate between 1.0 and 3.0 as a function of time. The pixel values set by `PixelRuler`, as well as those of the underlying default `CheckPattern`, are all either 1.0 or 0.0—therefore, changes in `gamma` should not be visible. The oscillation becomes visible if there is any spatial interpolation of pixel values, so it is a useful tool for highlighting any unintended geometric anomalies that violate the pixel-for-pixel assumption (sub-pixel shifts, non-90-degree rotations, scaling).

```python
Shady.Utilities.Cross(world=None, size=13, thickness=3, **props)
```

Create an image of a simple cross (horizontal and vertical strokes) and optionally render it as a `Stimulus` object.

**Parameters**

- `world` ([`World`]) – Optional. If supplied, the image will be rendered as a `Stimulus` with \( z=-0.9 \) and `linearMagnification=False`. If omitted, the function will simply return the array of pixel values.
- `size` (`int`) – Desired dimensions of the final image, in pixels. For symmetry, if `thickness` is odd, `size` should also be odd; and if `thickness` is even, `size` should also be even. Otherwise `size` will get reduced by 1.
- `thickness` (`int`) – Thickness of the strokes of the cross, in pixels.
- `**props` – Optional properties to be applied to the `Stimulus` object (ignored if `world` is not supplied).

**Returns** If a `World` instance `world` is supplied, returns a `Stimulus` instance. Otherwise just return the array of pixel values.

```python
Shady.Utilities.Loupe(target, update_period=1.0, **kwargs)
```

Given a `target` instance of class `Stimulus`, create another `Stimulus` instance that presents a magnified, contrast-enhanced, slowed-down (or rather, temporally sub-sampled) view of the pixels rendered by the `target`.

The returned instance has the following additional attributes:

- `target`: a `weakref.ref` to the `target` instance
- `update_period`: a floating-point number of seconds
- `update_now`: a boolean which, if set to `True`, forces an update on the next frame (and which is then automatically set back to `False`).

NB: raw (post-linearization) pixel values are captured from the target and then rendered on the loupe which is itself, necessarily, unlinearized. At extreme values, color-contrast-enhancement may therefore lead to an apparent change in mean luminance even though there is no such change in the target.

```python
Shady.Utilities.TearingTest(world, period_sec=4.0, proportional_width=0.15)
```

Show a high-contrast bar oscillating from side to side, allowing visual inspection of possible “tearing” artifacts.

```python
class Shady.Utilities.FrameIntervalGauge(world, corner=(-1, -1), thickness=49, variable='width', color=(0, 0, 0), useTexture=True)
```

Display an animated gauge that visually records the frame-to-frame interval in milliseconds. Every millisecond is marked in blue; every ten milliseconds in red. To avoid overhead, no text is rendered.

```python
Shady.Utilities.EstimateFrameRate(world, nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1, wait=True)
```

DOC-TODO
Shady.Utilities.PlotTimings 
\( \text{arg, savefig=None, traces=(), axes=None, finish=True, **kwargs} \) 
Plot a graph of the recent timing diagnostics from a World instance or log-file name. 

The information is richer if the World instance, and even more so if individual Stimulus instances, had debugTiming set to True. 

Requires the third-party packages numpy and matplotlib.

Shady.Utilities.Histogram 
\( \text{img, plot=True, DACmax=255, title=None, xlabel='Red, Green or Blue DAC Value', ylabel='Number of Pixels'} \) 
Takes an array of pixel values (for example, a Stimulus.Capture() output) and computes histograms of the luminance values in each channel. Optionally, plots the histograms. 

Requires the third-party package numpy to compute histograms, and matplotlib if you want to plot them.

Shady.Utilities.VDP 
\( \text{*pargs, **kwargs} \) 
Return viewing distance measured in pixels, based on a set of configuration parameters. This allows easy conversion between degrees and pixels.

If a single scalar numeric argument is provided, return it unchanged.

If a string is provided, treat it the name of a Python file which defines the necessary settings as variables, and execute it.

If a Shady World object is provided, infer widthInPixels and heightInPixels from its size. Otherwise, if a dict is provided, take the necessary settings from that.

Use **kwargs to augment / override settings.

The necessary settings are:

- viewingDistanceInMm, and
- EITHER: widthInPixels AND widthInMm OR: heightInPixels AND heightInMm

Flexibility is allowed with these variable names: they are case-invariant and underscore-invariant, the 'In' is optional, and the physical units can be mm, cm or m. So for example:

```python
viewing_distance_cm = 75
```

would give the same result as:

```python
ViewingDistanceInMm = 750
```

Example:

```python
v = VDP( world, heightInMm=169, viewingDistanceInCm=75 )
pixelsPerDegree = DegreesToPixels( 1.0, v )
```

Shady.Utilities.PixelsToDegrees 
\( \text{extentInPixels, screenInfo, eccentricityInPixels=0} \) 
Compute the number of degrees of visual angle subtended by a stimulus of given its extent in pixels, and its distance from the point of fixation in pixels.

Parameters

- extentInPixels – an int, float or sequence/array of numbers, denoting the size of the stimulus in pixels.
- screenInfo – either the input (a filename or a dict) or the output (a floating-point number) of VDP().
- eccentricityInPixels – an int, float or sequence/array of numbers, denoting the distance between fixation and the center of the stimulus, in pixels.
Returns a float (for scalar inputs) or a numpy.array (for array inputs) denoting angle(s) subtended, in degrees of visual arc.

See also:

• DegreesToPixels()
• VDP()

Shady.Utilities.DegreesToPixels (extentInDegrees, screenInfo, eccentricityInDegrees=0)
Compute the extent of a stimulus in pixels, given the number of degrees of visual arc it subtends and its eccentricity in degrees away from the line of sight.

Parameters

• extentInDegrees – an int, float or sequence/array of numbers, denoting the number of degrees subtended at the eye by the stimulus.
• screenInfo – either the input (a filename or a dict) or the output (a floating-point number) of VDP().
• eccentricityInDegrees – an int, float or sequence/array of numbers, denoting the angle in degrees between the line of sight and the line from the eye to the center of the stimulus

Returns a float (for scalar inputs) or a numpy.array (for array inputs) denoting stimulus extent(s) in pixels.

See also:

• PixelsToDegrees()
• VDP()

class Shady.Utilities.CommandLine (argv=None, dashes=2, doc=None)
c = CommandLine() Create the object. Uses argv=sys.argv[1:] by default, as well as doc=__doc__.
c.Option(...) Define an option. By default, the option name and resolved value will be stored as key and value in the dict c.opts. If you specify another container it will be stored there instead (and if you say container=None, it will not be stored anywhere). In any case the resolved value will also be returned from Option()
c.Help() Define and process the --help option: if it was supplied, print documentation and then either sys.exit() or raise CommandLineHelp().
c.Finalize() Issue an error if there are unrecognized options.
c.Delegate() An alternative to Finalize(). Returns argv with all already-recognized options removed, so you can pass it to the next CommandLine parser in the cascade (in cases where you have more than one).

Shady.Utilities.WorldConstructorCommandLine (argv=None, doc=None, **defaults)
This tool is used in our example scripts, and may be useful to you in any applications you write that are similarly launched from the command line. It creates a general-purpose CommandLine object, but then pre-configures various options to match the main arguments that can be passed to the World object constructor.

Example:
import Shady

cmdline = Shady.WorldConstructorCommandLine( canvas=True )
# Add your own custom command-line option definitions here:
hello = cmdline.Option( 'hello', default=False, type=bool, container=None )
cmdline.Help().Finalize()

world = Shady.World( **cmdline.opts )
if hello:
    import Shady.Text
greeting = world.Stimulus( text='Hello World!' )
world.Run()

In the above example, most of the World constructor arguments now have counterparts that can be passed on the command-line when you run foo.py. For example, if you say python foo.py --screen=2 then the World will be opened on your second screen. If you say python foo.py --help then (thanks to the Help() call above) the command-line arguments will be printed and then Python will exit without creating the World. The command-line arguments all have the same default values as their counterparts in the World constructor prototype, with one exception: canvas has had its default value explicitly changed to True.

In the particular cases of width and/or height, they can be passed as named arguments (say, python foo.py --width=1920 --height=1080) or as positional arguments (python foo.py 1920 1080).

If the doc argument is left as None, this function will impolitely ransack the calling namespace for a __doc__ variable, and use that if it finds it. Pass doc='' to suppress this.

Shady.Utilities.AutoFinish( world, shell=False, prefer_ipython=True, plot='auto' )
A useful call at the end of a demo script, to navigate the various ways in which the script, and any Shady.World that was created in it, might be running (single- or dual-threaded by direct interpretation of the script; dual-threaded with the World running in the main thread and the script interpreted in a subsidiary thread via python -m Shady).

Parameters

- world – A World instance, ready to run.
- shell (bool) – Whether or not to spawn a threaded interactive shell while the World is running.
- prefer_ipython (bool) – If a threaded shell is used, this indicates whether to try, to import the third-party package IPython to improve the interactive experience. If this is False, or if IPython is not installed, then any interactive prompt spawned by shell=True will be a plain-vanilla Python prompt.
- plot – This argument can be:
  - a callable function with zero arguments: in this case it is called when world finishes running; then it is assumed that one or more matplotlib figures have been generated by the call and that they should be kept alive and responsive.
  - a boolean specifying whether matplotlib figures already exist that need to be kept alive and responsive when world finishes running.
  - 'auto' (default): If world.debugTiming is True, then call PlotTimings( world ) when world finishes running. If not, infer automatically whether there are matplotlib figures.

Returns None
Shady Documentation, Release 1.9.1

Shady.Utilities.RunShadyScript(*argv, **kwargs)

If a script name is specified, run it. If not, run an interactive prompt. In either case, interpret the commands in a new subsidiary thread—the main thread will be reserved. Construction and running of one Shady.World, if performed in the script or from the prompt, will be diverted to the main thread. This allows Shady applications to run in multi-threaded mode even on non-Windows operating systems, on which the graphical back-end insists on being in the main thread.

With a script, the console option allows the script to run in demo mode, in which sections of the code can be run from an interactive prompt.

This routine is used under the hood when you start Python with −m Shady. Here are some pairs of commands that produce equivalent results starting from outside and from inside Python:

```bash
python −m Shady demo showcase
--> RunShadyScript('showcase', console='IPython')
python −m Shady run showcase
--> RunShadyScript('showcase', console=None)
python −m Shady run showcase --console=Python
--> RunShadyScript('showcase', console='Python')
```

Parameters

- *argv – positional arguments, just as you would use from a system command-line. If the first element is sys.argv then sys.argv[1:] will be used plus any subsequent *argv and **kwargs arguments.
- **kwargs – optional keyword arguments, just as you would specify --key=value options from the command-line.

If you specify neither argv nor kwargs, then sys.argv[1:] will be used. Options are as follows:

- **--script (or positional argument 0)** The name of a Python script to run. If your script argument is not a valid (absolute or relative) path to an existing file, Shady will make a second attempt to find the named script inside the examples subdirectory of the Shady package itself (adding a py extension if necessary). If no script is specified, an interactive prompt will be opened instead.
- **--console** Values can be 'None', 'Python' or 'IPython', specifying a preference for the type of interactive prompt that should be interleaved with script execution. The default is 'IPython' although Shady will fall back to 'Python' if the third-party IPython package is not available. 'None' will only be respected if a script has been specified, and it causes the script to be run without any interaction from the console.

Shady.Utilities.ComplexPolygonBase(nsides, appendNaN=True, joined=False)

Return a 1-by-n numpy.array of complex numbers that describe the vertices of a polygon.

Parameters

- nsides (int) – number of sides (or vertices) of the polygon.
- appendNaN (bool) – whether to append a NaN (interpreted as a break between polygons in the Stimulus.points property).
- joined (bool) – whether to repeat the first vertex explicitly at the end. Only necessary if you want to draw unfilled (i.e. wireframe) closed polygons.

Returns a 1-by-n numpy.array of complex numbers, where n is nsides, plus 1 if appendNaN is True, plus 1 if joined is True.

Example:
from Shady import World, ComplexPolygonBase, Real2DToComplex

w = World(1000)
s = w.Stimulus( drawMode=Shady.DRAWMODE.POLYGON, anchor=-1, color=[1,0,0] )
shape = 50 * ComplexPolygonBase( 12 )
locations = 150 * ComplexPolygonBase( 3, appendNaN=False ).T
s.points = shape + locations

See also:

• Real2DToComplex()
• ComplexToReal2D()

Shady.Utilities.Real2DToComplex(x, add_dims_left=0, add_dims_right='auto')
Given an n-by-2 array of real-valued coordinates, return the coordinates as an array of complex numbers (default shape n-by-1, but depends on add_dims_left and add_dims_right).

Alternatively, given one coordinate expressed as a 2-element numeric sequence (i.e. a list, tuple or 1-D numpy array of length 2), return a single complex number.

Inverse of ComplexToReal2D()

Shady.Utilities.ComplexToReal2D(x)
Given a single complex number x, return [x.real, x.imag]. Or, by extension, given a single scalar real number x, return [float(x), 0.0].

Alternatively, if x is a sequence (i.e. a list, tuple or numpy array) of complex numbers, convert to an n-by-2 array of real-valued coordinates.

Inverse of Real2DToComplex()

class Shady.Utilities.VideoRecording (filename, fps=60.0, codec='mp4v', container='.avi')
This class allows frames (in the form of numpy arrays) to be written to a video file. It wraps the VideoWriter class from OpenCV.

It requires third-party packages numpy and cv2 (the latter being part of opencv-python).

Write frames with WriteFrame(). Remember to Close() when finished.

Example (capture video of one particular Stimulus):

world = Shady.World( canvas=True, gamma=-1, noise=-0.1 )
stim = world.Stimulus( sigfunc=1, siga=0.4, pp=0, cx=lambda t:t*100, atmosphere=world )
world.fakeFrameRate = 60.0 # ensures accurate slower-than-real-time animation # (because .Capture() operations may be slow)
movie = Shady.VideoRecording( 'example_movie', fps=world )
world.OnClose( movie.Close )

@stim.AnimationCallback
def StimFrame( self, t ):
    movie.WriteFrame( self ) # we can pass a frame directly as a `numpy` array,
    # or we can pass a `Stimulus` or `World` instance # in which case its `.Capture()` method will be called automatically
world.Run()
Parameters

- **filename (str)** – stem, or name, or path, to the video file to be saved. If filename includes a file extension, the container format is inferred from that instead of from the container argument.

- **fps (int, float, World)** – frames per second. Can also use a World instance if that instance’s `fakeFrameRate` property has been set.

- **codec (str)** – four characters that specify the FOURCC code of the codec to use. The default ‘mp4v’ is a reasonably safe cross-platform/out-of-the-box option, especially with the .avi container format. But, as with all things OpenCV, and indeed all things video-codec-related, your mileage may vary considerably. Note that ‘mp4v’ is a lossy codec. For lossless compression (and hence perfectly accurate reconstruction of your stimuli) you may need to install a third-party codec—this seems to be quite a technical task so we can’t go into detail here.

- **container (str)** – file extension dictating the default video container format (used only if there is no file extension in filename).

**WriteFrame (frame)**

The frame argument can be a numpy array, height x width x channels, of uint8 pixel values. The number of channels should be either 3 (RGB) or 4 (RGBA). The alpha channel, if any, will be ignored.

Alternatively, frame can be a Stimulus or World instance, in which case its `Capture()` method is called automatically.

Note that, once the first frame is written, you should ensure that all subsequent frames have the same dimensions as the first.

### 5.5.6 Shady.Text Sub-module

To activate Shady’s ability to render text, you have to explicitly:

```python
import Shady.Text
```

This is a break from Shady’s usual behavior of arriving with all batteries included. The reason is that this import can take several seconds, as Shady (via the third-party package matplotlib, if available) collates a list of available fonts. (For some reason, even in the twenty-first century, it is a universal truth that the initialization of any application that wants a few dozen fonts to be available must be ridiculously time-consuming.)

The third-party packages numpy and pillow (or PIL) are required. The matplotlib package is optional, but without it you will be limited to one font (monaco).

Once activated, you probably do not need direct access to anything inside the Shady.Text submodule. Instead you can just work with the `text` property of any Shady.Stimulus object. It can be as simple as:

```python
import Shady.Text
w = Shady.World()
s = w.Stimulus( text='Hello, World!' )
```

You can assign a string to the Stimulus property `text`, but what that really does is create an instance of type TextGenerator which has a number of sub-properties. In fact, assigning a string to s.text is really a syntactic shortcut for creating the TextGenerator instance s.text if it doesn’t already exist, then assigning the string to the sub-property s.text.string. Subproperties are accessible in the Stimulus() constructor, and also as (dynamic-assignment-capable) properties of the Stimulus instance itself, using an underscore-delimited notation:
Note that, in contrast to Shady’s usual way of doing things, any change to the content, style or appearance of a piece of text requires the CPU to do the rendering work and then transfer the resulting image texture from CPU to GPU. This will very quickly cause frame skips if you want to change the content of a large text image frequently.

The subproperties are as follows:

**family, a.k.a. family_name, font, font_family, font_name:**

Assign a string here, and Shady will attempt to find a font whose name contains all the words in that string. If no match can be found, the font_found property will be False.

**bold, italic, black, heavy, light, regular, etc...:**

These style properties will be present to the extent that the operating system has fonts installed whose names end with these words. Each one is a boolean property that allows you to toggle its respective style on or off. They may or may not work for any given font (e.g. the “bold” version of a given font may not exist on your system). You can always query the font_found property to discover whether it has worked.

**style:**

This is an alternative way of controlling font style. You can assign a string containing one or more style words here, for example 'bold italic'. As explained above, this may or may not succeed depending on what your current font provides.

**font_found:**

This read-only property is a boolean value that indicates whether the requested font family name and style can be found. If not, the stimulus will render text in Shady’s built-in default font (monaco). This will always be the case if the third-party package matplotlib is not installed, because matplotlib is required in order to access the operating system’s fonts.

**emWidthInPixels, a.k.a. em:**

By default, this is None, because by default the size of your font is specified by lineHeightInPixels (a.k.a size). If you assign a value here, lineHeightInPixels will become None, and the font size will be chosen such that the lower-case letter “m” is this many pixels wide.

**lineHeightInPixels, a.k.a. size:**

This is the default way of dictating font size, although it will be None if you have assigned a value to emWidthInPixels instead. It dictates the vertical space allocated to each line of text, in pixels (to within a margin of error that depends on the sizes that the current font provides). Note that the lines may appear to occupy additional space if linespacing is not 1.0

**linespacing:**

This dictates where one line of text starts, vertically, relative to the next. It is expressed as a proportion of the line height. The default value is 1.1, which means there will be blank space between lines equal to 10% of the line height. Values less than 1.0 may cause lines of text to overlap.

**string:**

This is the text string to be rendered. It may contain newline characters. It may be a unicode string (however, note that special characters will only be rendered correctly to the extent that the currently selected font provides them).

**wrapping:**

216 Chapter 5. To cite Shady:
This is **None** by default, which means your **string** content will be rendered unchanged. If you give it a positive integer, then your **string** will be re-wrapped to the specified number of pixels, with a ragged edge. If you give it a negative integer, then similarly the wrapped width in pixels is equal to the magnitude of that integer, but now the text is left- and right-justified to remove the ragged edge. Paragraph breaks are retained, as are whitespace characters that indent the beginning of each paragraph. A paragraph break can be indicated by a blank line, or indeed by a newline character that is immediately followed by *any* additional whitespace.

**border:**

This dictates the thickness of the border around the text. It can be a scalar (in which case it is considered to be a proportion of the text size) or it can be a pair of numbers (in which case it is interpreted as horizontal and vertical thickness in pixels).

**pad to:**

This is **None** by default, indicating that the resulting texture image should occupy the minimal space necessary to satisfy the other parameters. However, you can use this property to guarantee a larger minimum size, in pixels. Supply either a single integer (to create a square image) or a pair of integers indicating width and height.

**align:**

Set this to 'left', 'right' or 'center' to dictate the way in which multiple lines of the same text image are aligned relative to each other.

**fill:**

This can be a single number in the range 0 to 1, or it can be a sequence of three or four such numbers specifying an RGB or RGBA color that is used for the text. By default, the text is white (1.0) which means that the text color can also be controlled as expected by the **color** property of the **Stimulus** instance itself.

**bg:**

This can be a single number in the range 0 to 1, or it can be a sequence of three or four such numbers specifying an RGB or RGBA color that is used fill the background of each line of text. This includes the **border**, but does not include space beyond the end of shorter lines, nor does it include space between lines that results from **linespacing** values greater than 1. By default, the background is fully transparent.

**block bg:**

This can be a single number in the range 0 to 1, or it can be a sequence of three or four such numbers specifying an RGB or RGBA color that is used to manipulate independently the background color of the text lines themselves, superimposed on the **block bg** color. By default, the background is fully transparent.

### 5.5.7 Shady.Video Sub-module

This submodule is not imported by default by the parent package. Import it for its side-effect: it enables the **video** property of the **Shady.Stimulus** class.

If **s** is a **Stimulus** instance and **s.video** is equal to its default value of **None**, then saying:

```python
s.video = 'fish.mp4'
```

implicitly creates a **VideoSource** object with the **source** property equal to 'fish.mp4'. (If **s.video** was previously already a **VideoSource** object then its **source** property is simply updated accordingly.)
The third-party package cv2 (installable with `pip install opencv-python`) is required for video support. Any source readable by `cv2.VideoCapture` is acceptable: use an integer to open a live camera stream, or a string to specify a video file name.

If you want to record the video to disk while rendering, there are two approaches. One way is to capture frames in a way that is time- (or frame-) locked to the animation of the `World`: in this case, see the doc for the `VideoRecording` class, which can capture the content of any `Stimulus` regardless of whether its source is a `VideoSource`, or even capture an entire `World`. The other way is to record a `VideoSource` that is already attached to a `Stimulus`, sub-sampling frames at an independent pace in a background thread: in this case, simply call `s.video.Record(filename)`.

If you want to record a video to disk without attaching it to a `Shady.Stimulus`, or even without having a `Shady.World` running at all. Let’s say you want to record from camera 0:

```python
from Shady.Video import VideoSource
s = VideoSource(0).Record('foo')
# ...
s.Close()
```

An acquisition thread will continually read frames from the camera until you call `s.Close()`. A separate recording thread will continually write frames to the specified file (`foo.avi` in this example) until you call either `s.Close()` or `s.StopRecording()`.

Note that reading frames from a camera or file, capturing stimulus content into memory, and recording video frames to disk all use RAM and place a burden on the CPU—they are not intended for applications in which precise timing is critical.

### 5.6 License

This file is part of the Shady project, a Python framework for real-time manipulation of psychophysical stimuli for vision science.

Copyright (C) 2017-18 Jeremy Hill, Scott Mooney

Shady is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program (file COPYING in the main package directory). If not, see [http://www.gnu.org/licenses/](http://www.gnu.org/licenses/).

We know the GNU Public License is restrictive. It may not be suitable for your purposes. If you need something different, contact the authors.
CHAPTER 6

Indices and tables

- genindex
- modindex
- search
S
Shady.Contrast, 203
Shady.Dynamics, 192
Shady.Linearization, 200
Shady.Text, 215
Shady.Utilities, 205
Shady.Video, 217
Index

A
adg (Shady.Stimulus attribute), 173
AddCustomModulationFunction() (in module Shady), 190
AddCustomSignalFunction() (in module Shady), 189
AddCustomUniform() (Shady.Stimulus class method), 165
AddCustomUniform() (Shady.World class method), 149
AddCustomWindowingFunction() (in module Shady), 190
AddForeignStimulus() (Shady.World method), 149
addr (Shady.Stimulus attribute), 173
AfterClose() (Shady.World method), 149
alpha (Shady.Stimulus attribute), 173
anchor (Shady.Stimulus attribute), 173
anchor (Shady.World attribute), 159
anchor_x (Shady.Stimulus attribute), 174
anchor_x (Shady.World attribute), 159
anchor_y (Shady.Stimulus attribute), 174
anchor_y (Shady.World attribute), 159
angle (Shady.Stimulus attribute), 174
AnimationCallback() (Shady.Stimulus method), 166
AnimationCallback() (Shady.World method), 149
Apply() (in module Shady.Dynamics), 198
ApplyLUT() (in module Shady.Linearization), 202
atmosphere (Shady.Stimulus attribute), 174
atmosphere (Shady.World attribute), 159
AutoFinish() (in module Shady.Utilities), 212
ax_n (Shady.Stimulus attribute), 174
ax_n (Shady.World attribute), 159
ay_n (Shady.Stimulus attribute), 174
ay_n (Shady.World attribute), 160

B
BackEnd() (in module Shady), 191
backgroundAlpha (Shady.Stimulus attribute), 174
backgroundColor (Shady.Stimulus attribute), 174
backgroundColor (Shady.World attribute), 160
BeforeClose() (Shady.World method), 150
bg (Shady.Stimulus attribute), 175
bg (Shady.World attribute), 160
bgalpha (Shady.Stimulus attribute), 175
bgblue (Shady.Stimulus attribute), 175
bgblue (Shady.World attribute), 160
bgcolor (Shady.Stimulus attribute), 175
bgcolor (Shady.Stimulus attribute), 160
bgred (Shady.Stimulus attribute), 175
bgred (Shady.World attribute), 160
bitCombiningMode (Shady.World attribute), 160
BitStealingLUT() (in module Shady.Linearization), 202
blue (Shady.Stimulus attribute), 175
blue (Shady.World attribute), 161
bluegamma (Shady.Stimulus attribute), 175
bluegamma (Shady.World attribute), 161
blunoise (Shady.Stimulus attribute), 175
blunoise (Shady.World attribute), 161
BoundingBox() (Shady.Stimulus method), 166
BoundingBox() (Shady.World method), 150

C
CallOnce() (in module Shady.Dynamics), 197
CancelOnClose() (Shady.World method), 150
Capture() (Shady.Stimulus method), 166
Capture() (Shady.World method), 150
carrierRotation (Shady.Stimulus attribute), 175
carrierScaling (Shady.Stimulus attribute), 175
carrierTranslation (Shady.Stimulus attribute), 176
visible (Shady.World attribute), 164

W

Wait () (Shady.World method), 158
WaitFor () (Shady.Stimulus method), 173
WaitFor () (Shady.World method), 158
WaitUntil () (in module Shady.Dynamics), 198
Watch () (Shady.Dynamics.Function method), 194
width (Shady.Stimulus attribute), 188
width (Shady.World attribute), 164
windowFunction (Shady.Stimulus attribute), 188
windowingFunction (Shady.Stimulus attribute), 188
WINFUNC (class in Shady), 192
winfunc (Shady.Stimulus attribute), 188
World (class in Shady), 147
WorldConstructorCommandLine () (in module Shady.Utilities), 211
WriteFrame () (Shady.Utilities.VideoRecording method), 215

X

x (Shady.Stimulus attribute), 188
xscale (Shady.Stimulus attribute), 188
xscaling (Shady.Stimulus attribute), 188
xy (Shady.Stimulus attribute), 189

Y

y (Shady.Stimulus attribute), 189
yscale (Shady.Stimulus attribute), 189
yscaling (Shady.Stimulus attribute), 189

Z

z (Shady.Stimulus attribute), 189