ratcave Documentation

Release 0.5

Nicholas A Del Grosso

Jun 12, 2018
## Table of Contents:

1.1 Introduction ...................................................... 1
1.2 Installation ....................................................... 1
1.3 Features ............................................................ 1
1.4 System Requirements ............................................. 3
1.5 Tutorials ............................................................ 3
1.6 CAVE Virtual Reality ........................................... 31
1.7 API Documentation .............................................. 34

### Python Module Index

51
1.1 Introduction

ratcave is a Python package for displaying 3D Graphics. It was inspired by a Virtual Reality CAVE Setup for rodents in a neuroscience lab in Munich, Germany, and was meant to make the creation of 3D experiments simple and accessible. ratcave has since evolved into a standalone wrapper for modern OpenGL constructs, like programmable shaders, environment mapping, and deferred rendering. Because it wraps these OpenGL features directly, it also works with all popular python OpenGL graphics engines, including Pyglet, PsychoPy, and PyGame.

Finally, ratcave is written to reduce boilerplate code, in order to make writing simple 3D environments easy. It does this using many python features, including dictionary-like uniform assignment and context managers to bind OpenGL objects.

1.2 Installation

ratcave supports both Python 2 and Python 3, and can be installed via pip!

```
    pip install ratcave
```

1.3 Features

ratcave was created to be a graphics package for doing behavioral experiments with animals in a freely-moving virtual reality environment. The goals of this project are:

- **Less Boilerplate, more Coding:** More code means more errors. Many behavioral experiments tend to be complex sets of logic written in a single script, so I tried to make ratcave as low-profile as possible to keep the focus on the experiment, not on the graphics management.

- **Ease of Use:** Moving objects in a scene, displaying a window, and changing objects’ colors should be intuitive.
• *high-temporal performance*: Lag is the enemy of immersive VR, and we wanted to take advantage of our 360 fps display for VR research. Advanced hardware-accelerated algorithms and modern OpenGL constructs are an essential part of doing high-performance graphics in Python.

• *Cubemapping Support* (the essential algorithmic approach for a single-projector CAVE VR system),

• *Free and Open Source*

What I’ve found so far is that ratcave makes for a succinct 3D graphics engine, even for simple 3D scenes, making it a useful candidate package for psychophysics research in general. To that end, I’ve made it very compatible with the PsychoPy package, as a way of extending PsychoPy experiments to 3D scenes. While we are still at an early stage of development with ratcave, we’ve already reached the requirements listed above, with a goal of continually refactoring and adding features to make ratcave the mature backend it has the potential to be. If you are interested in aiding the development of ratcave, either through contributions on GitHub, bug reporting, or even simply testing it out yourself and giving us feedback, we hope you’ll get involved and help us develop this little project into something wonderful!

### 1.3.1 Supplied 3D Primitives

Blender 3D’s built-in primitives (Cone, Sphere, Cube, etc) come packaged with ratcave, making it easier to get started and prototype your 3D application. A reader object for Blender’s .obj Wavefront files is also included.

### 1.3.2 Supplied 3D Shaders

ratcave is “batteries-included”: You get diffuse shading, specular reflections, shadows, and even FXAA antialiasing in the packaged shaders. These shaders are open-source and free to be edited and improved!
1.3.3 Pythonic Interface

FrameBuffer Context Managers

Normally, the OpenGL code to bind a framebuffer involves the following:

```python
glGetIntegerv(GL_VIEWPORT, old_viewport_size)
glBindFramebufferEXT(GL_FRAMEBUFFER, fbo_id)  # Rendering off-screen
glViewport(0, 0, texture_width, texture_height)
<< Draw Scene Here >>
glBindFramebufferEXT(GL_FRAMEBUFFER, 0)
glViewport(old_viewport_size)
```

In ratcave, this is a simple context manager:

```python
with fbo:
    scene.draw()
```

Shader Uniforms

OpenGL Shader Uniform creation and setting is also wrapped in a pythonic way:

```python
sphere.uniforms['diffuse_color'] = [1., 0., 0.]  # RGB values
```

1.3.4 Fast Execution

ratcave uses Numpy arrays, c binaries, and GLSL OpenGL to make rendering detailed scenes fast!

1.4 System Requirements

At the moment, ratcave’s shaders require OpenGL 3.3, though this is planned to change in future releases. If you’d like to use ratcave and don’t have a graphics driver that supports OpenGL 3.3, however, you can already load your own shaders and it will work fine.

1.5 Tutorials

These Tutorials are meant to help you get started!

1.5.1 Tutorial 1: Displaying a 3D Object

This tutorial will show the process of displaying a 3D object onscreen. This will be done in four steps:

1. We’ll open a file containing 3D objects—a Wavefront .obj file containing basic 3D primitives that comes with ratcave (although you can use any .obj file outputted by 3D modeling software), using the `WavefrontReader` class.

2. We then retrieve a `Mesh` object from the file. Mesh objects contain all information about the object, including its position (inside its Local and World attributes, which are `Physical` objects), color (inside its Material attribute, which are of the `Material` class), and even the vertex data itself.
• We’ll put the Mesh inside a Scene object, which is a container class that holds Mesh objects, a Camera object, and a Light object, along with an RGB background color. Multiple Scenes can be created, even ones that contain the same Meshes, and rendering one vs another one is as simple as calling the scene’s Scene.draw() method.

Note: Each tutorial follows from each other, assuming knowledge of the previous. To get the most out of ratcave, then, it is best to do them in order!

Note: Some of the constructs in this tutorial, like decorators and event loops, come from Pyglet. While completely understanding them isn’t necessary, it’ll probably best to start out with a Pyglet tutorial to get your feet wet. This one is good: http://www.natan.termitnjak.net/tutorials/pyglet_basic.html

Starting an OpenGL Context and a Window

ratcave depends on their already being an OpenGL context set up before loading objects. This can be done by any OpenGL manager (Pyglet and PsychoPy are useful, but PyGame and Qt OpenGL windows should work fine as well). So, before doing anything in ratcave, a window must first be created. In these tutorials, I’ll show it with Pyglet:

```python
import pyglet
import ratcave as rc

window = pyglet.window.Window()
```

If you want to verify that Pyglet is working and the window gets created, just start Pyglet’s event loop at the end of your script. This script will create a Pyglet window that closes when you press the escape key:

```python
pyglet.app.run()
```
Warning: Creating a Window automatically starts the OpenGL context, which is important for many aspects of ratcave. As a general rule, it’s good to make your window first, before doing anything else!

Getting Pyglet Actively Drawing

Pyglet’s event loop won’t automatically update the scene, so we’ll stick in a function that does nothing, that gets called every frame, to guarantee that everything appears onscreen. Anything you want done between frames (updating positions, logging events, etc) can go in this function:

```python
def update(dt):
    pass
pyglet.clock.schedule(update)
```

Reading a Wavefront .obj file

To load a 3D object, let’s read in a .obj file! The built-in WavefrontReader can read triangulated, uv-maped .obj files exported from Blender. ratcave comes with some primitive 3D objects in its resoufruities module, so let’s use one of those:
# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Check which meshes can be found inside the Wavefront file, and extract it into a Mesh object for rendering.
print(obj_reader.bodies.keys())
>>> ['Torus', 'Sphere', 'Monkey', 'Cube']

## Loading a Mesh from the WavefrontReader and Positioning it

Loading a mesh can be done through the `WavefrontReader.get_mesh()` method. By default, the mesh will have its position in the same location as in its .obj file, but this can be easily changed. Because the camera is in the -z direction by default per OpenGL convention, let’s set it in front of the camera:

```python
monkey = obj_reader.get_mesh("Monkey")
monkey.position.xyz = 0, 0, -2
```

## Creating a Scene

Scenes consist of meshes, lights, and a camera–everything we need to view and position object in the real world! Let’s put the monkey Mesh into a Scene:

```python
scene = rc.Scene(meshes=[monkey])
```

## Drawing the Scene

To draw the scene, we need a 3D shader (discussed in more detail in the next tutorial). Luckily, ratcave provides one to start with! Simply call the `Scene.draw()` method in your draw loop! In Pyglet, this looks like this:

```python
@window.event
def on_draw():
    with rc.default_shader:
        scene.draw()
pyglet.app.run()
```

## Summary

That’s it! Here’s the final script, in one place. This script will be modified in the next tutorial to animate the scene:

```python
import pyglet
import ratcave as rc

# Create Window
window = pyglet.window.Window()

def update(dt):
    pass
pyglet.clock.schedule(update)
```

(continues on next page)
# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey")
monkey.position.xyz = 0, 0, -2

# Create Scene
scene = rc.Scene(meshes=[monkey])

@window.event
def on_draw():
    with rc.default_shader:
        scene.draw()

pyglet.app.run()

Version using PsychoPy

Alternatively, you can see the same example using a PsychoPy window:

```python
import ratcave as rc
from psychopy import visual, event

# Create Window
window = visual.Window()

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey")
monkey.position.xyz = 0, 0, -2

# Create Scene
scene = rc.Scene(meshes=[monkey])

while 'escape' not in event.getKeys():
    with rc.default_shader:
        scene.draw()
        window.flip()

window.close()
```
1.5.2 Tutorial 2: Animating a Scene with Multiple Meshes and Moving the Camera with the Keyboard

This tutorial will build on the previous one by adding some more interesting elements. We’ll allow the user to move the scene’s camera by pressing the left and right arrow keys, and have multiple meshes in the scene that move.

**Warning:** This tutorial builds on code from Tutorial 1. If you have not yet completed Tutorial 1, it’s best to go and do that, first!

**Scenes Hold Lists of Meshes**

Let’s insert a couple Meshes from our `obj_reader` WavefrontReader object into the scene!

```python
# Create Meshes from WavefrontReader
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5), scale=.6)
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.4)

# Create Scenes with Meshes.
scene = rc.Scene([monkey, torus])
```
Moving a Mesh

Now, we’ll animate the Meshes by changing their rotation attributes in the update function:

```python
def rotate_meshes(dt):
    monkey.rotation.y += 15 * dt  # dt is the time between frames
    torus.rotation.x += 80 * dt
pyglet.clock.schedule(rotate_meshes)
```

Modifying Scene’s Background Color

Scenes also have a background color, saved as an RGB tuple in the Scene.bgColor attribute:

```python
scene.bgColor = 1, 0, 0
```

Moving the Camera with the Keyboard

While we could easily make a new Camera object from scratch, we’ll just grab the scene’s camera and have it accept keyboard inputs for movement:

```python
# This is how to get keyboard input in pyglet:
from pyglet.window import key
keys = key.KeyStateHandler()
window.push_handlers(keys)

def move_camera(dt):
    camera_speed = 3
    if keys[key.LEFT]:
        scene.camera.x -= camera_speed * dt
    if keys[key.RIGHT]:
        scene.camera.x += camera_speed * dt
pyglet.clock.schedule(move_camera)
```

Now you should have an interactive scene! Don’t forget to use the arrow keys to move around!
Summary

Here’s the full code for Tutorial 2:

```python
import pyglet
from pyglet.window import key
import ratcave as rc

# Create Window and Add Keyboard State Handler to it's Event Loop
window = pyglet.window.Window()
keys = key.KeyStateHandler()
window.push_handlers(keys)

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5), scale=.6)
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.4)
```

(continues on next page)
scene = rc.Scene(meshes=[monkey, torus])
scene.bgColor = 1, 0, 0

# Functions to Run in Event Loop

def rotate_meshes(dt):
    monkey.rotation.y += 15 * dt  # dt is the time between frames
    torus.rotation.x += 80 * dt
pyglet.clock.schedule(rotate_meshes)

def move_camera(dt):
    camera_speed = 3
    if keys[key.LEFT]:
        scene.camera.position.x -= camera_speed * dt
    if keys[key.RIGHT]:
        scene.camera.position.x += camera_speed * dt
pyglet.clock.schedule(move_camera)

@window.event
def on_draw():
    with rc.default_shader:
        scene.draw()

pyglet.app.run()

PsychoPy Version

Here’s the same scenario, done in PsychoPy:

```
from psychopy import visual, event
import ratcave as rc

camera_speed = 2

# Create Window and Add Keyboard State Handler to it's Event Loop
window = visual.Window()

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5), scale=.6)
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.4)

# Create Scene
scene = rc.Scene(meshes=[monkey, torus])
scene.bgColor = 1, 0, 0

while True:
    dt = .016
```
keys_pressed = event.getKeys()
if 'escape' in keys_pressed:
    window.close()
    break

# Move Camera
for key in keys_pressed:
    if key == 'left':
        scene.camera.x -= camera_speed * dt
    elif key == 'right':
        scene.camera.x += camera_speed * dt

# Rotate Meshes
monkey.rot_y += 15 * dt  # dt is the time between frames
torus.rot_x += 80 * dt

# Draw Scene and Flip to Window
with rc.default_shader:
    scene.draw()
window.flip()  

1.5.3 Tutorial 3: Custom GLSL Shaders, Sending Data to the Graphics Card

To get the most out of our graphics, many newer graphics engines use programs running on the graphics card called “shaders” to specify how objects should be shown on-screen. While teaching GLSL shaders is beyond the scope of this tutorial, and ratcave allows you to completely skip writing shaders at all by supplying a few useful ones, you’ll likely want to use a shader of your own.

In this tutorial, you’ll learn how to use ratcave to:

- Compile a Shader object and use it in the Scene.draw() function.
- Send data to the shader from Python as a Uniform variable.

Warning: This tutorial builds on the previous tutorials. If you’re just getting started, it’s recommended to go back and do those tutorials first!

Initial Script

Since the previous tutorials have already covered a lot of ratcave methods, let’s just start with the following script:

```python
import pyglet
import ratcave as rc

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()

# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))
scene = rc.Scene(meshes=[torus])
```
# Constantly-Running mesh rotation, for fun

def update(dt):
    torus.rotation.y += 20. * dt
pyglet.clock.schedule(update)

# Draw Function
@window.event
def on_draw():
    with rc.default_shader:
        scene.draw()

# Pyglet’s event loop run function
pyglet.app.run()

This code should display a rotating torus on the window.

Creating a Custom GLSL Shader

Now, one thing ratcave does automatically is use it’s built-in genShader Shader, if none is specified. This is to make it easier to get started. Let’s replace it with our own custom shader program, which simply positions the mesh in 3D
Shader programs come in two types. **Vertex Shaders** tell the graphics card where a vertex will appear on your screen. Our shader here will take data from the meshes, the lights, and the camera to determine where everything goes:

```
vert_shader = ""
#version 120
attribute vec4 vertexPosition;
uniform mat4 projection_matrix, view_matrix, model_matrix;

void main()
{
    gl_Position = projection_matrix * view_matrix * model_matrix * vertexPosition;
}
"
```

**Warning:** This shader requires OpenGL 3.3 drivers to be installed, along with an OpenGL 3.3-compatible graphics card on your system.

The **fragment shader** takes the vertex shader’s position data and determines what color a pixel on the screen will be. These can get quite complex, but we’ll use a fairly simple one here, and just make everything automatically appear red:

```
frag_shader = ""
#version 120
uniform vec3 diffuse;
void main()
{
    gl_FragColor = vec4(diffuse, 1.);
}
"
```

**Note:** Normally, you would just put these shaders in their own files, but here we’ll keep everything together and use them as strings.

Now, to make the **Shader**

```
shader = rc.Shader(vert=vert_shader, frag=frag_shader)
```

Using the shader during drawing is done in a shader keyword argument in `Scene.draw()`:

```
scene.draw(shader=shader)
```

Here is what the code should look like now:

```
import pyglet
import ratcave as rc
import time
import math

vert_shader = ""
#version 120
attribute vec4 vertexPosition;
uniform mat4 projection_matrix, view_matrix, model_matrix;
```
```python
void main()
{
    gl_Position = projection_matrix * view_matrix * model_matrix * vertexPosition;
}

frag_shader = ""
#version 120
uniform vec3 diffuse;
void main()
{
    gl_FragColor = vec4(diffuse, 1.);
}

shader = rc.Shader(vert=vert_shader, frag=frag_shader)

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()

# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))
torus.uniforms['diffuse'] = [.5, .0, .8]
scene = rc.Scene(meshes=[torus])

# Constantly-Running mesh rotation, for fun
def update(dt):
    torus.rotation.y += 20. * dt
pyglet.clock.schedule(update)

def update_color(dt):
    torus.uniforms['diffuse'][0] = 0.5 * math.sin(time.clock() * 30) + .5
pyglet.clock.schedule(update_color)

# Draw Function
@window.event
def on_draw():
    with shader:
        scene.draw()

# Pyglet's event loop run function
pyglet.app.run()
```

If you run it, you should see a flat red torus!
Sending Data to the Shader using Uniforms

Data can be attached to each object and sent to the shaders, to customize their behavior. Here, let’s let the `Mesh.uniforms['diffuse']()` uniform control what color the torus takes.

In the fragment shader, add this line to initialize the `diffuse` uniform variable before the main function:

```glsl
uniform vec3 diffuse;
```

In the python code, modify the `diffuse` key in the `Mesh.uniforms()` attribute:

```python
torus.uniforms['diffuse'] = [.2, .8, .8]
```

**Note:** All ratcave objects come with some default uniforms, to make setting up easier and to make naming schemas more consistent. This shouldn’t restrict you, though—new uniforms are automatically initialized when you add them dictionary-style, like `torus.uniforms['my_uniform'] = 3.0`!

If you run the code now, you should now see a cyan rotating torus. Let’s make it a little more dynamic, shall we?
import time
import math
def update_color(dt):
    torus.uniforms['diffuse'][0] = 0.5 * math.sin(time.clock()) + 1
pyglet.clock.schedule(update_color)

Now the torus will change color!

Summary

Here’s the updated code:

```python
import pyglet
import ratcave as rc
import time
import math

vert_shader = ""
#version 330
layout(location = 0) in vec3 vertexPosition;
```

(continues on next page)
uniform mat4 projection_matrix, view_matrix, model_matrix;
out vec4 vVertex;

void main()
{
  vVertex = model_matrix * vec4(vertexPosition, 1.0);
  gl_Position = projection_matrix * view_matrix * vVertex;
}

frag_shader = ""
#version 330
out vec4 final_color;
uniform vec3 diffuse;
void main()
{
  final_color = vec4(diffuse, 1.0);
}
""

shader = rc.Shader(vert=vert_shader, frag=frag_shader)

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()

# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))
torus.uniforms['diffuse'] = [.5, .0, .8]
scene = rc.Scene(meshes=[torus])

# Constantly-Running mesh rotation, for fun
def update(dt):
    torus.rotation.y += 20. * dt
pyglet.clock.schedule(update)

def update_color(dt):
    torus.uniforms['diffuse'][0] = 0.5 * math.sin(time.clock() * 10) + .5
pyglet.clock.schedule(update_color)

# Draw Function
@window.event
def on_draw():
    with shader:
        scene.draw()

# Pyglet's event loop run function
pyglet.app.run()

In the next tutorial, we'll follow this up by drawing to an FBO dynamically!
1.5.4 Tutorial 4: Using Cubemapping to Render a CAVE VR System

CAVE VR relies on position updates from head trackers to render a virtual scene from the subject’s perspective in virtual space.

- Two different Scene objects are used: a virtual Scene, which contains the virtual environment to be cubemapped which is rendered from the subject’s perspective (meaning, the camera goes where the subject is) - a “real” Scene, which contains just the model (also a Mesh) of the screen on which the VR is being projected, seen from the perspective of the video projector.

While this is difficult to show without having an actual tracking system, we’ll illustrate this effect and the code needed to run it by making an animation:

| Warning: This tutorial assumes knowledge gained from the previous tutorials. If you are just getting started, it’s recommended to start from Tutorial 1! |

Import Pyglet and ratcave, and Start the Window and OpenGL Context

At the beginning of the script:

```python
import pyglet
import ratcave as rc
from ratcave.resources import cube_shader

window = pyglet.window.Window(resizable=True)
shader = rc.Shader.from_file(*rc.resources.genShader)
```

At the end of the script:

```python
pyglet.app.run()
```

Create the Virtual Scene

Let’s say that our virtual scene contains a red sphere and a cyan cube:

```python
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
sphere = obj_reader.get_mesh("Sphere", position=(0, 0, 2), scale=0.2)
sphere.uniforms['diffuse'] = 1, 0, 0
cube = obj_reader.get_mesh("Cube", position=(0, 0, 0), scale=0.2)
cube.uniforms['diffuse'] = 1, 1, 0

# Put inside a Scene
virtual_scene = rc.Scene(meshes=[sphere, cube])
```

Note that we have one object at the origin (0, 0, 0). Since our light is also at 0,0,0 by default, this may affect how things appear. Let’s move the scene’s light:

```python
virtual_scene.light.position = 0, 3, -1
```
Create the Projected Scene

The Projected Scene is what is actually sent to the display. It will contain the screen (or rodent arena, if you’re in a rodent neuroscience lab like us!). Here, let’s just use a flat plane to be used as our screen, and use a monkey to show where the subject is looking from (note: the subject isn’t necessary for actual VR, it’s just used here for illustration of the cubemapping approach).

```python
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1), scale=0.8)
screen = obj_reader.get_mesh("Plane", position=(0, 0, 1), rotation=(1.5, 180, 0))
projected_scene = rc.Scene(meshes=[monkey, screen], bgColor=(1., 1., 1.))
projected_scene.light.position = virtual_scene.light.position
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0))
```

Setting Your Cameras

A Camera used for Cubemapping

Cubemapping involves rendering an image from six different angles: up, down, left, right, forward, and backward, and stitching each of these six images onto the faces of a cube (for more info, see http://www.nvidia.com/object/cube_map_ogl_tutorial.html). For this algorithm to work, then, two of the Camera’s properties must be customized:

- Camera.aspect(): The camera’s image must be square (meaning it’s width-to-height aspect ratio must be 1.0)
- Camera.fov_y(): The camera must be able to see 90-degrees, so that the sides all match up.

Altering the camera to be useful for cubemapping is straightforward:

```python
cube_camera = rc.Camera(projection=rc.PerspectiveProjection(fov_y=90, aspect=1.))
virtual_scene.camera = cube_camera
```

The Projector Camera

In order to do CAVE VR, the camera you use to render the screen must exactly match not only the position and rotation of your video projector relative to the screen, but also the lens characteristics as well. This requires some calibration and measuring on your part, which will differ based on your setup and hardware. Since this is just a demo, let’s just arbitrarily place the camera above the scene, looking down:

```python
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0), z_far=6)
```

The aspect of the camera should, ideally, match that of the window. Let’s do that here, using Pyglet’s on_resize event handler so that it will happen automatically, even when the screen is resized:

```python
@window.event
def on_resize(width, height):
    projected_scene.camera.aspect = width / float(height)
```

Create the OpenGL FrameBuffer and Cube Texture

So far, we’ve always rendered our Scenes straight to the monitor. However, we can also render to a texture! This lets us do all kinds of image postprocessing effects, but here we’ll just use it to update a cube texture, so the screen always has the latest VR image:
cube_texture = rc.texture.TextureCube()  # this is the actual cube texture
cube_fbo = rc.FBO(cube_texture)

All that’s left is to apply the texture the screen:

screen.textures.append(cube_texture)

**Warning:** The built-in shader that comes with ratcave requires the subject’s position to be sent to it through the `playerPos` uniform. This may be remedied in future releases, or can be changed in your own custom shaders. To do this, use: `screen.uniforms['playerPos'] = virtual_scene.camera.position`

### Move the Subject

Let’s have the Monkey move left-to-right, just to illustrate what cubemapping does:

```python
clock = 0.
def update(dt):
    global clock
    clock += dt
    monkey.position.x = math.sin(1.3 * clock)
    virtual_scene.camera.position.xyz = monkey.position.xyz
    screen.uniforms['playerPos'] = virtual_scene.camera.position.xyz
pyglet.clock.schedule(update)
```

### Draw the Scenes

All that’s left is for the scenes to be drawn. The `virtual_scene` should be drawn to the FBO, and the `projected_scene` to the window. To perform the rotations correctly and in the right order, a convenient `Scene.draw360_to_texture()` method has been supplied:

```python
@window.event
def on_draw():
    with cube_shader:
        with cube_fbo as fbo:
            virtual_scene.draw360_to_texture(fbo.texture)
            projected_scene.draw()
```

### Summary

Here’s the full code:

```python
import pyglet
import ratcave as rc
import math, time
from ratcave.resources import cube_shader

window = pyglet.window.Window(resizable=True)

# Assemble the Virtual Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
sphere = obj_reader.get_mesh("Sphere", position=(0, 0, 2), scale=0.2)

(continues on next page)
sphere.uniforms['diffuse'] = 1, 0, 0

cube = obj_reader.get_mesh("Cube", position=(0, 0, 0), scale=0.2)
cube.uniforms['diffuse'] = 1, 1, 0

# virtual_scene = rc.Scene(meshes=[sphere, cube], bgColor=(0., 0., 1.))
virtual_scene = rc.Scene(meshes=[cube, sphere], bgColor=(0., 0., 1.))
virtual_scene.light.position.xyz = 0, 3, -1

cube_camera = rc.Camera(projection=rc.PerspectiveProjection(fov_y=90, aspect=1.))
virtual_scene.camera = cube_camera

# Assemble the Projected Scene
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1), scale=0.8)
screen = obj_reader.get_mesh("Plane", position=(0, 0, 1), rotation=(1.5, 180, 0))
projected_scene = rc.Scene(meshes=[monkey, screen, sphere, cube], bgColor=(1., .5, 1.))
projected_scene.light.position = virtual_scene.light.position
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0))
projected_scene.camera.projection.z_far = 6

# Create Framebuffer and Textures

cube_texture = rc.texture.TextureCube(width=1024, height=1024)  # this is the actual
→
cube texture

cube_fbo = rc.FBO(texture=cube_texture)
screen.textures.append(cube_texture)

clock = 0.
def update(dt):
    global clock
    clock += dt
    monkey.position.x = math.sin(1.3 * clock)
    virtual_scene.camera.position.xyz = monkey.position.xyz
    screen.uniforms['playerPos'] = virtual_scene.camera.position.xyz
    pyglet.clock.schedule(update)

@window.event
def on_draw():
    with cube_shader:
        with cube_fbo as fbo:
            virtual_scene.draw360_to_texture(fbo.texture)
            projected_scene.draw()

pyglet.app.run()
dependencies (in between parent and children objects), by defining relationships between objects. Additionally the Empty Entity objects will be introduced.

**Warning:** This tutorial builds on the previous tutorials. If you’re just getting started, it’s recommended to go back and do those tutorials first!

**Initial Script**

Since the previous tutorials have already covered a lot of ratcave methods, let’s just start with the following script:

```python
import pyglet
import ratcave as rc

# Create Window
window = pyglet.window.Window(resizable=True)

def update(dt):
    pass
pyglet.clock.schedule(update)

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)
```

**Meshes and Empty Entities**

We can not define more then one rotation speed for the sun object. To introduce different rotation for each of the planets, we need to use Empty Entities. Empty Entities are objects, that occupy physical space, but doesn’t actually draw anything when `Scene.draw()` is called (just passing the values). Later, we can set rotation speed for each of the Empty Entity object:

```python
# Create Meshes
sun = obj_reader.get_mesh("Sphere", name='sun!')
merkury = obj_reader.get_mesh("Sphere", scale =.1, name='merkury')
venus = obj_reader.get_mesh("Sphere", scale =.2, name='venus')
earth = obj_reader.get_mesh("Sphere", scale =.2, name='earth')
mars = obj_reader.get_mesh("Sphere", scale =.2, name='mars')
jupyter = obj_reader.get_mesh("Sphere", scale =.4, name='jupyter')
moon = obj_reader.get_mesh("Sphere", scale =.5, name='moon')

# Create Empty Entities
empty_merkury = rc.EmptyEntity(name='sun_merkury')
empty_venus = rc.EmptyEntity(name='sun_venus')
empty_earth = rc.EmptyEntity(name='sun_earth')
empty_mars = rc.EmptyEntity(name='sun_mars')
empty_jupyter = rc.EmptyEntity(name='sun_jupyter')
```
Define Relationships and Relative Positions

Relationships Between Objects

To define layout relationship in between objects in ratcave, user has to link them together using SceneGraph properties:

- `Mesh.add_child()`
- `Mesh.add_children()`
- `Mesh.parent()`

The following code does the job:

```python
# Define Relationships
sun.add_children(empty_merkury, empty_earth, empty_venus, empty_mars, empty_jupyter)
empty_merkury.add_child(merkury)
empty_venus.add_child(venus)
empty_earth.add_child(earth)
empty_mars.add_child(mars)
empty_jupyter.add_child(jupyter)
earth.add_child(moon)
```

Relative Objects Positions

Additionally it is important to define the position of the children in relative position to the parent (position of planets in relation to the sun). This can be done in a following way:

```python
# Define Relative Positions
sun.rotation.x = 50
sun.position.xyz = 0, 0, -12
merkury.position.z += 1
venus.position.z += 2
earth.position.z += 3
mars.position.z += 4
jupyter.position.z += 5
moon.position.z += 1
```

Setting Rotations

Each of the rotations has to be set separately:

```python
def on_draw():
    with rc.default_shader:
        sun.rotation.y += 0.5
        earth.rotation.y += 0.5
        empty_merkury.rotation.y += 2
        empty_venus.rotation.y += 1.5
        empty_earth.rotation.y += 1
        empty_mars.rotation.y += 0.75
        empty_jupyter.rotation.y += 0.5
```
Scene - Update

After definition of a scene:

```python
scene = rc.Scene(meshes=sun, bgColor=(0,0,0))
```

sun and all of its children now get drawn when `scene.draw()` gets called. There is no further need of updating any of the Meshes (or its children) included in the scene. You can also decide which of the elements are going to be drawn, by calling them separately, the position of the planets will still be relative to the sun (also when sun itself is not being drawn):

```python
def on_draw():
    window.clear()
    with rc.default_shader, scene.camera, scene.light:
        sun.draw()
        earth.draw()
```

Additionally you can parent the camera and light to one of the Mesh objects. It can be done in following manner:

```python
#Define Relationships For Cameras and Objects
earth.add_child(scene.camera)
earth.add_child(scene.light)
```

If you run it, you should see this simulation of solar system:
Summary

Here is the full code for the Tutorial 5:

```python
import pyglet
from pyglet.window import key
import ratcave as rc

# Create Window
window = pyglet.window.Window(resizable=True)
keys = key.KeyStateHandler()
window.push_handlers(keys)

def update(dt):
    pass
pyglet.clock.schedule(update)

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)
```

(continues on next page)
# Create Meshes
sun = obj_reader.get_mesh("Sphere", name='sun')
merkury = obj_reader.get_mesh("Sphere", scale =.1, name='merkury')
venus = obj_reader.get_mesh("Sphere", scale =.2, name='venus')
earth = obj_reader.get_mesh("Sphere", scale =.2, name='earth')
mars = obj_reader.get_mesh("Sphere", scale =.2, name='mars')
jupyter = obj_reader.get_mesh("Sphere", scale =.4, name='jupyter')
moon = obj_reader.get_mesh("Sphere", scale =.5, name='moon')

# Create Empty Entities
empty_merkury = rc.EmptyEntity(name='sun_merkury')
empty_venus = rc.EmptyEntity(name='sun_venus')
empty_earth = rc.EmptyEntity(name='sun_earth')
empty_mars = rc.EmptyEntity(name='sun_mars')
empty_jupyter = rc.EmptyEntity(name='sun_jupyter')

# Define Relationships
sun.add_children(empty_merkury, empty_earth, empty_venus, empty_mars, empty_jupyter)
empty_merkury.add_child(merkury)
empty_venus.add_child(venus)
empty_earth.add_child(earth)
empty_mars.add_child(mars)
empty_jupyter.add_child(jupyter)
earth.add_child(moon)

# Define Relative Positions
sun.rotation.x = 50
sun.position.xyz = 0, 0, -12
merkury.position.z += 1
venus.position.z += 2
earth.position.z += 3
mars.position.z += 4
jupyter.position.z += 5
moon.position.z += 1
sun.textures.append(rc.Texture.from_image(rc.resources.img_colorgrid))

# Create Scene
scene = rc.Scene(meshes=sun, bgColor=(0,0,0))
scene.camera.projection.z_far = 20

# Define Relationships For Cameras and Objects
# earth.add_child(scene.camera)
# earth.add_child(scene.light)
planets = [sun, earth, jupyter]

def move_camera(dt):
    """function used to parent the camera to a different planet""
    if keys[key.LEFT]:
        cam_parent = planets.pop(0)
        cam_parent.add_child(scene.camera)
planets.append(cam_parent)
pyglet.clock.schedule(move_camera)

@window.event
def on_draw():
    window.clear()
    sun.rotation.y += 0.5
    earth.rotation.y += 0.5
    empty_merkury.rotation.y += 2
    empty_venus.rotation.y += 1.5
    empty_earth.rotation.y += 1
    empty_mars.rotation.y += 0.75
    empty_jupyter.rotation.y += 0.5

    with rc.default_shader:
        scene.draw()

pyglet.app.run()

1.5.6 Tutorial 6: Stereopsis (3D)

Up until now, we were creating a scene that contains 3D objects, and we were looking at the scene through our screen(s). This means the 3D scene is projected onto a 2D screen. We still observe depth due to motion parallax, lighting, and other so-called monocular cues, but we do not have a 3D view which gives us a better sense of objects’ depth.

Stereopsis is the perception of depth obtained through binocular disparity, which requires the graphics package to provide separate images for each eye. Luckily, ratcave does that!

Let’s get the monkey on our screen.

Initial Script

Here is a script which displays the monkey on your screen:

```python
import pyglet
import ratcave as rc

window = pyglet.window.Window(resizable=True)

model_file = rc.resources.obj_primitives
monkey = rc.WavefrontReader(model_file).get_mesh('Monkey')
monkey.position.xyz = 0, 0, -2.5

camera = rc.Camera()

@window.event
def on_draw():
    window.clear()

    with rc.default_shader, rc.default_states, camera:
        monkey.draw()
```

(continues on next page)
When the light is reflected from the monkey’s face we get a feeling that the monkey is a 3D object. But are we seeing 3D? No.

### StereoCameraGroup

Earlier we talked about stereopsis - having two different images for each eye to perceive depth. Therefore, the first step is to actually have two eyes. The “eye” in our virtual scene is the camera. We can create a StereoCameraGroup object which creates those two cameras automatically for us. We simply replace Camera() with StereoCameraGroup():

```python
import pyglet
import ratcave as rc

window = pyglet.window.Window(resizable=True)

model_file = rc.resources.obj_primitives
monkey = rc.WavefrontReader(model_file).get_mesh('Monkey')
monkey.position.xyz = 0, 0, -2.5

camera = rc.StereoCameraGroup()
```

At this point we have a camera group that with a little bit more work can give us depth.

### 3D Display Mode

Due to different hardware solutions, there are two modes that stereopsis can be implemented on your screen:

- Anaglyph (red and blue)
- Active mode

Anaglyph mode is the simplest one. In this mode, we apply a color filter to the image that is seen by the camera. That is, for instance, we filter red color for right camera (right eye), and filter the cyan for the left camera (left eye).

The other mode is the active mode which utilizes active shutter 3D glasses. This mode requires your graphics card to support 3D displays. We will go through implementation of anaglyph 3D mode in this tutorial and to see 3D at the end of this tutorial, all you need is an anaglyph glasses.

### Switching between cameras

After creating a StereoCameraGroup object, we have two cameras: camera.left and camera.right:
When you run the above code, you should be able to see two overlapping monkeys. However, this is still not useful with the anaglyph glasses - we need to apply a color filter. To do that, we can use OpenGL `glColorMask()` provided by `pyglet.gl` class. With this function we can activate or deactivate color channels on our scene. For instance, to deactivate red channel:

```python
from pyglet.gl import gl

gl.glColorMask(True, False, False, True)
```

Now if we apply this color filter with its corresponding camera, the `on_draw()` changes to:

```python
@window.event
def on_draw():
    window.clear()
    with rc.default_shader, rc.default_states:
        with camera.right:
            gl.glColorMask(False, True, True, True)
            monkey.draw()

        gl.glClear(gl.GL_DEPTH_BUFFER_BIT)

        with camera.left:
            gl.glColorMask(True, False, False, True)
            window.clear()
            monkey.draw()
```

Make sure to add `glClear(gl.GL_DEPTH_BUFFER_BIT)` before drawing the scene of the second camera. This ensures that the depth information of the previously drawn object is removed and the color information of both camera scenes are drawn on the screen without the interruption of depth testing.

Now you can use your anaglyph glasses and enjoy the 3D view. Here is the complete code, and its output:

```python
import pyglet
from pyglet.gl import gl
import ratcave as rc

window = pyglet.window.Window(resizable=True)

# import an object
model_file = rc.resources.obj_primitives
monkey = rc.WavefrontReader(model_file).get_mesh('Monkey')
monkey.position.xyz = 0, 0, -2.5

# create a stereopsis camera group object
camera = rc.StereoCameraGroup()
```
@window.event
def on_draw():
    window.clear()
    with rc.default_shader, rc.default_states:
        with camera.right:
            gl.glColorMask(False, True, True, True)
            monkey.draw()
            gl.glClear(gl.GL_DEPTH_BUFFER_BIT)
        with camera.left:
            gl.glColorMask(True, False, False, True)
            monkey.draw()

def update(dt):
    pass

pyglet.clock.schedule(update)
pyglet.app.run()

_images/tut6_stereomonkey.png

1.6 CAVE Virtual Reality

Building your own virtual reality setup requires several parts:

- A 3D Graphics Engine (to render the virtual environment)
- Video Displays (to show the environment to the user). Ideally, these should be large enough to allow the user to see a large amount of the virtual environment! This means you’ll either want something that is:
  - Head-fixed (the display is attached to the user’s head, so that they can always see a virtual space, no matter where they turn)
  - Projected on surfaces all around the user. If there are screens in a 360-degree arc around the user, you get a CAVE system!
- A head tracking system, to update the virtual environment when the user moves their head. This is one of the things that gives such a strong immersive sense to virtual reality.

1.6.1 Our Setup

The VR setup we made is intended for use by rats, mice, and gerbils for cognitive neuroscience experiments, which is why we call this python package ratcave! In selecting our components, we were both limited by and helped by the small sizes of our users:

- We use a simplified 3D graphics engine, to make our experiment scripts small and simple to deploy (python, with pyglet or psychopy + ratcave)
• Head-mounting a display on mice wasn’t an option because of their small size, so we use a single wide-lens video projector, which front-projects onto the walls and floor of the rodent’s arena. This gives them a 360-degree view of the virtual environment, while reducing the computational demands of our setup.

• We use an optical tracking system for measuring the rodent’s head space via data collected from camera arrays. We use _NaturalPoint’s Optitrack System: [http://www.optitrack.com/](http://www.optitrack.com/) - To control this tracking system from Python, we wrote a Python package called MotivePy, available here: [https://github.com/neuroneuro15/motivepy](https://github.com/neuroneuro15/motivepy)
  - To access the data from the tracking system on a remote client in our experiment scripts, we wrote another Python client called NatNetClient, available here: [https://github.com/neuroneuro15/natnetclient](https://github.com/neuroneuro15/natnetclient)

1.6.2 Example VR Experiment Script

Writing a VR Script is relatively straightforward, and involves three main components:

1. Connect to your tracking system
2. Render your 3D environment
3. In a loop, re-render the 3D environment, setting the camera position at the new eye position of your user.

In Pyglet, ratcave, and NatNetClient, this would something like this:

```python
import pyglet
import ratcave as rc
import natnetclient

# Connect to Tracking System
client = natnetclient.NatClient(ip='197.0.0.10', port=5023)
user = client.rigid_bodies['User']
```
# Create Scene and put in a draw loop
window = pyglet.window.Window()
reader = rc.WavefrontReader(rc.resources.obj_primitives)
scene = rc.scene(meshes=[reader.get_mesh('Sphere', position=(0, 1, -2))])

@window.event()
def on_draw():
    scene.draw()

# Update camera position, based on user's position
def update(dt):
    scene.camera.position = user.position
pyglet.clock.schedule()

# Start App
pyglet.app.run(

In Psychopy, which is written in a more imperative format, it looks like this:

```python
from psychopy import visual, event
import ratcave as rc
import natnetclient

# Connect to Tracking System
client = natnetclient.NatClient(ip='197.0.0.10', port=5023)
user = client.rigid_bodies['User']

window = visual.Window()

# Main Loop
while 'escape' not in event.getKeys():
    # Create Scene
    reader = rc.WavefrontReader(rc.resources.obj_primitives)
    scene = rc.scene(meshes=[reader.get_mesh('Sphere', position=(0, 1, -2))])

    # Update Camera position, based on user's position
    scene.camera.position = user.position

    # Draw
    scene.draw()
    window.flip()
```

1.6.3 Modular Nature of VR

At its core, VR does not stand for any one technology. Whether you are using head-mounted screens or projector, optical camera array tracking systems or treadmills, or any sort of graphics engine out there, the key is that you are changing the camera position on a loop, based on some user movement. Hopefully, this description has given you some ideas for how you can set up your own VR system!
1.7 API Documentation

1.7.1 Main Modules Contents

Mesh

This module contains the Mesh and EmptyEntity classes.

```python
class Mesh(arrays, textures=(), mean_center=True, gl_states=(), drawmode=, point_size=15, dynamic=False, visible=True, **kwargs)
Bases: ratcave.shader.HasUniformsUpdater, ratcave.physical.PhysicalGraph, ratcave.utils.mixins.NameLabelMixin
```

Returns a Mesh object, containing the position, rotation, and color info of an OpenGL Mesh.

Meshes have two coordinate system, the “local” and “world” systems, on which the transforms are performed sequentially. This allows them to be placed in the scene while maintaining a relative position to one another.

**Note:** Meshes are not usually instantiated directly, but from a 3D file, like the WavefrontReader .obj and .mtl files.

**Parameters**

- **arrays (tuple)** – a list of 2D arrays to be rendered. All arrays should have same number of rows. Arrays will be accessible in shader in same attrib location order.
- **mean_center (bool)** –
- **texture (Texture)** – a Texture instance, which is linked when the Mesh is rendered.
- **gl_states** –
- **drawmode** – specifies the OpenGL draw mode
- **point_size (int)** –
- **dynamic (bool)** – enables dynamic manipulation of vertices
- **visible (bool)** – whether the Mesh is available to be rendered. To make hidden (invisible), set to False.

**Returns** Mesh instance

```python
copy()
```
Returns a copy of the Mesh.

```python
draw()
```
Draw the Mesh if it’s visible, from the perspective of the camera and lit by the light. The function sends the uniforms

```python
dynamic
```
dynamic property of the mesh. If set to True, enables the user to modify vertices dynamically.

```python
classmethod from_incomplete_data (vertices, normals=(), texcoords=(), **kwargs)
```
Return a Mesh with (vertices, normals, texcoords) as arrays, in that order. Useful for when you want a standardized array location format across different amounts of info in each mesh.

```python
classmethod from_pickle (filename)
```
Loads and Returns a Mesh from a pickle file, given a filename.
normals
Mesh normals array.

reset_uniforms()
Resets the uniforms to the Mesh object to the ""global"" coordinate system

texcoords
UV coordinates

texture
to_pickle(filename)
Save Mesh to a pickle file, given a filename.

vertices
Mesh vertices, centered around 0,0,0.

vertices_global
Vertex position, in world coordinate space (modified by model_matrix)

vertices_local
Vertex position, in local coordinate space (modified by model_matrix)

Camera

class Camera(projection=None, orientation0=(0, 0, -1), **kwargs)
Bases: ratcave.physical.PhysicalGraph, ratcave.shader.HasUniformsUpdater,
ratcave.utils.mixins.NameLabelMixin

Returns a camera object

Parameters

• projection (obj) – the projection type for the camera. It can either be an instance of
  OrthoProjection or PerspectveProjection

• orientation0 (tuple) –

Returns Camera instance

classmethod from_pickle(filename)
Loads and Returns a Camera from a pickle file, given a filename.

projection
Returns the Camera’s Projection

projection_matrix
Returns projection matrix of the Camera

reset_uniforms()

to_pickle (filename)
Save Camera to a pickle file, given a filename.

Texture

class Texture(id=None, name='TextureMap', width=1024, height=1024, data=None, mipmap=False, **kwargs)
Bases: ratcave.shader.HasUniforms, ratcave.utils.mixins.BindTargetMixin

2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.
attach_to_fbo()
Attach the texture to a bound FBO object, for rendering to texture.

bind()

classmethod from_image(img_filename, mipmap=False, **kwargs)
Uses Pyglet's image.load function to generate a Texture from an image file. If `mipmap`, then texture will have mipmap layers calculated.

generate_mipmap()

max_texture_limit
The maximum number of textures available for this graphic card's fragment shader.

name
reset_uniforms()

slot
The texture's ActiveTexture slot.

unbind()

Scene
class Scene(meshes=(), camera=None, light=None, bgColor=(0.4, 0.4, 0.4), gl_states=(,), **kwargs)
Bases: ratcave.utils.mixins.NameLabelMixin

Returns a Scene object, that manages the creation of the scene needed to view the projection of the Objects. Class manages rendering of Meshes, Lights and Cameras.

Parameters
- meshes (Mesh) – all of Mesh instances that you want to view in the Scene
- camera (Camera) – a Camera instance, if not provided created automatically
- light (Light) – a Light instance, if not provided created automatically
- bgColor (float) – defines the color of the background

Returns
Scene instance
clear()
Clear Screen and Apply Background Color
draw(clear=True)
Draw each visible mesh in the scene from the perspective of the scene’s camera and lit by its light.
draw360_to_texture(cubetexture, **kwargs)
Draw each visible mesh in the scene from the perspective of the scene’s camera and lit by its light, and applies it to each face of cubetexture, which should be currently bound to an FBO.
draw_anaglyph(clear=True, inter_eye_distance=0.08)

Shader
class Shader(vert=", frag=", geom=", lazy=False)

GLSL Shader program object for rendering in OpenGL. To activate, call the Shader.bind() method, or pass it to a context manager (the ‘with’ statement).
Examples and inspiration for shader programs can be found at https://www.shadertoy.com/.

**Parameters**
- **vert** (-) – The vertex shader program string
- **frag** (-) – The fragment shader program string
- **geom** (-) – The geometry shader program

**Example:**
```python
code = 'your_code_here'
with shader:
    mesh.draw()
```

**bind()**
Activate this Shader, making it the currently-bound program.

Any Mesh.draw() calls after bind() will have their data processed by this Shader. To unbind, call Shader.unbind().

**Example:**
```python
shader.bind()
mesh.draw()
shader.unbind()
```

**Note:** Shader.bind() and Shader.unbind() can be also be called implicitly by using the ‘with’ statement.

**Example of with statement with Shader**:  
```python
with shader: mesh.draw()
```

**compile()**

**classmethod from_file**(vert, frag, **kwargs)  
Reads the shader programs, given the vert and frag filenames

**Parameters**
- **vert** (-) – The filename of the vertex shader program (ex: ‘vertshader.vert’)
- **frag** (-) – The filename of the fragment shader program (ex: ‘fragshader.frag’)

**Returns** The Shader using these files.

**Return type**
- shader (Shader)

**link()**
link the program, making it the active shader.

**Note:** Shader.bind() is preferred here, because link() Requires the Shader to be compiled already.
1.7.2 Main Topic Details

Meshes

mesh.py

This module contains the Mesh and EmptyEntity classes.

```python
class Mesh(arrays, textures=(), mean_center=True, gl_states=(), drawmode=, point_size=15, dynamic=False, visible=True, **kwargs)
```

Bases: `ratcave.shader.HasUniformsUpdater`, `ratcave.physical.PhysicalGraph`, `ratcave.utils.mixins.NameLabelMixin`

Returns a Mesh object, containing the position, rotation, and color info of an OpenGL Mesh.

Meshes have two coordinate system, the “local” and “world” systems, on which the transforms are performed sequentially. This allows them to be placed in the scene while maintaining a relative position to one another.

**Note:** Meshes are not usually instantiated directly, but from a 3D file, like the WavefrontReader .obj and .mtl files.

**Parameters**

- `arrays (tuple)` – a list of 2D arrays to be rendered. All arrays should have same number of rows. Arrays will be accessible in shader in same attrib location order.
- `mean_center (bool)` –
- `texture (Texture)` – a Texture instance, which is linked when the Mesh is rendered.
- `gl_states` –
- `drawmode` – specifies the OpenGL draw mode
- `point_size (int)` –
- `dynamic (bool)` – enables dynamic manipulation of vertices
- `visible (bool)` – whether the Mesh is available to be rendered. To make hidden (invisible), set to False.

**Returns** Mesh instance

`copy()`

Returns a copy of the Mesh.

`draw()`

Draw the Mesh if it’s visible, from the perspective of the camera and lit by the light. The function sends the uniforms

`dynamic`

dynamic property of the mesh. If set to True, enables the user to modify vertices dynamically.

**classmethod from_incomplete_data (vertices, normals=(), texcoords=(), **kwargs)**

Return a Mesh with (vertices, normals, texcoords) as arrays, in that order. Useful for when you want a standardized array location format across different amounts of info in each mesh.

**classmethod from_pickle (filename)**

Loads and Returns a Mesh from a pickle file, given a filename.
normals
Mesh normals array.

points

reset_uniforms()
Resets the uniforms to the Mesh object to the """"global"""" coordinate system

texcoords
UV coordinates

texture
to_pickle(filename)
Save Mesh to a pickle file, given a filename.

triangles

vertices
Mesh vertices, centered around 0,0,0.

vertices_global
Vertex position, in world coordinate space (modified by model_matrix)

vertices_local
Vertex position, in local coordinate space (modified by model_matrix)

class EmptyEntity(**kwargs)
Bases: ratcave.shader.HasUniformsUpdater, ratcave.physical.PhysicalGraph, ratcave.utils.mixins.NameLabelMixin
Returns an EmptyEntity object that occupies physical space and uniforms, but doesn't draw anything when draw() is called.

draw(*args, **kwargs)
Passes all given arguments

reset_uniforms()
Passes all arguments

wavefront.py
class WavefrontReader(file_name)
Bases: object

Reads Wavefront (.obj) files created in Blender to build ratcave.graphics Mesh objects. :param file_name: .obj file to read (assumes an accompanying .mtl file has the same base file name.) :type file_name: str :return: :rtype: WavefrontReader

generate(body_name, **kwargs)
Builds Mesh from geom name in the wavefront file. Takes all keyword arguments that Mesh takes.

material_property_map = {'Ka': 'ambient', 'Kd': 'diffuse', 'Ke': 'emission', 'Ks':

Cameras
camera.py
class Camera(projection=None, orientation0=(0, 0, -1), **kwargs)
Bases: ratcave.physical.PhysicalGraph, ratcave.shader.HasUniformsUpdater,
ratcave.utils.mixins.NameLabelMixin

Returns a camera object

Parameters

- **projection** (obj) – the projection type for the camera. It can either be an instance of OrthoProjection or PerspeectiveProjection
- **orientation0** (tuple) –

Returns Camera instance

classmethod from_pickle(filename)

Loads and Returns a Camera from a pickle file, given a filename.

projection

Returns the Camera’s Projection

projection_matrix

Returns projection matrix of the Camera

reset_uniforms()

save Camera to a pickle file, given a filename.

class CameraGroup (cameras=None, *args, **kwargs)

Bases: ratcave.physical.PhysicalGraph

Creates a group of cameras that behave dependently

look_at (x, y, z)

Converges the two cameras to look at the specific point

class OrthoProjection (origin='center', coords='relative', **kwargs)

Bases: ratcave.camera.ProjectionBase

Orthogonal Projection Object cretes projection Object that can be used in Camera

Parameters

- **origin** (str) – ‘center’ or ‘corner’
- **coords** (str) – ‘relative’ or ‘absolute’

Returns OrthoProjection instance

coords

Returns coordinates

origin

Returns origin of the Projection

class PerspectiveProjection (fov_y=60.0, aspect=1.25, x_shift=0.0, y_shift=0.0, **kwargs)

Bases: ratcave.camera.ProjectionBase

aspect

fov_y

match_aspect_to_viewport ()

Updates Camera.aspect to match the viewport’s aspect ratio.

x_shift

y_shift
class ProjectionBase(z_near=0.1, z_far=12.0, **kwargs)

Bases: object

Abstract Base Class for the Projections. Used to create projection matrix that later represents Camera Space. Vertex with position=(0,0,0), should be located in the middle of the scene. Projection matrix has defined z - distance to the camera.

Parameters

- **z_near** (float) – the nearest distance to the camera, has to be positive
- **z_far** (float) – the furthest point from the camera that is visible, has to be positive and bigger then z_near

Returns ProjectionBase instance

copy()

Returns a copy of the projection matrix

projection_matrix

Return projection_matrix

update()

Updates projection_matrix

viewport

returns the viewport

z_far

Return z_far value

z_near

Return z_near value

class ScreenEdges(left, right, bottom, top)

Bases: tuple

Create new instance of ScreenEdges(left, right, bottom, top)

bottom

Alias for field number 2

left

Alias for field number 0

right

Alias for field number 1

top

Alias for field number 3

class StereoCameraGroup(distance=0.1, projection=None, convergence=0.0, *args, **kwargs)

Bases: ratcave.camera.CameraGroup

Creates a group of cameras that behave dependently

corvergence

distance

class Viewport(x, y, width, height)

Bases: tuple

Create new instance of Viewport(x, y, width, height)
height
    Alias for field number 3

width
    Alias for field number 2

x
    Alias for field number 0

y
    Alias for field number 1

light.py

class Light(**kwargs)
    Bases: ratcave.camera.Camera, ratcave.shader.HasUniformsUpdater, ratcave.utils.mixins.NameLabelMixin
    reset_uniforms()

Textures
texture.py

class DepthTexture(name='DepthMap', *args, **kwargs)
    Bases: ratcave.texture.Texture
    the Color Cube Texture class.
    attachment_point
    internal_fmt
    pixel_fmt

class GrayscaleTexture(id=None, name='TextureMap', width=1024, height=1024, data=None, mipmap=False, **kwargs)
    Bases: ratcave.texture.Texture
    2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.
    internal_fmt
    pixel_fmt

class GrayscaleTextureCube(name='CubeMap', *args, **kwargs)
    Bases: ratcave.texture.TextureCube
    the Color Cube Texture class.
    internal_fmt
    pixel_fmt

class RenderBuffer(width, height)
    attach_to_fbo()
    attachment_point
bindfun

internal_fmt
target

class Texture (id=None, name='TextureMap', width=1024, height=1024, data=None, mipmap=False, **kwargs)
Bases: ratcave.shader.HasUniforms, ratcave.utils.mixins.BindTargetMixin

2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.

attach_to_fbo ()
    Attach the texture to a bound FBO object, for rendering to texture.

attachment_point
bind ()
bindfun
classmethod from_image (img_filename, mipmap=False, **kwargs)
    Uses Pyglet's image.load function to generate a Texture from an image file. If ‘mipmap’, then texture will have mipmap layers calculated.

generate_mipmap ()

internal_fmt
max_texture_limit
    The maximum number of textures available for this graphic card’s fragment shader.

name
pixel_fmt
reset_uniforms ()

slot
    The texture’s ActiveTexture slot.

target
target0
unbind ()

class TextureCube (name='CubeMap', *args, **kwargs)
Bases: ratcave.texture.Texture

the Color Cube Texture class.

attach_to_fbo (face=0)
    Attach the texture to a bound FBO object, for rendering to texture.

classmethod from_image (img_filename)
    Uses Pyglet’s image.load function to generate a Texture from an image file. If ‘mipmap’, then texture will have mipmap layers calculated.
	
target
target0
Relationships

scene.py

```python
class Scene(meshes=(), camera=None, light=None, bgColor=(0.4, 0.4, 0.4), gl_states=(), **kwargs)
    Bases: ratcave.utils.mixins.NameLabelMixin
    Returns a Scene object, that manages the creation of the scene needed to view the projection of the Objects. Class manages rendering of Meshes, Lights and Cameras.

    Parameters
    • meshes (Mesh) – all of Mesh instances that you want to view in the Scene
    • camera (Camera) – a Camera instance, if not provided created automatically
    • light (Light) – a Light instance, if not provided created automatically
    • bgColor (float) – defines the color of the background

    Returns
    Scene instance

clear()
    Clear Screen and Apply Background Color

draw (clear=True)
    Draw each visible mesh in the scene from the perspective of the scene’s camera and lit by its light.

draw360_to_texture (cubetexture, **kwargs)
    Draw each visible mesh in the scene from the perspective of the scene’s camera and lit by its light, and applies it to each face of cubetexture, which should be currently bound to an FBO.

draw_anaglyph (clear=True, inter_eye_distance=0.08)
```

scenegraph.py

```python
class SceneGraph (parent=None, children=None, **kwargs)
    Bases: object
    A Node of the Scenegraph. Has children, but no parent.

    add_child (child)
        Adds an object as a child in the scene graph.

    add_children (*children, **kwargs)
        Convenience function: Adds objects as children in the scene graph.

    children

    parent
        A SceneNode object that is this object’s parent in the scene graph.

    remove_children (*children)
```

physical.py

```python
class Physical (position=(0.0, 0.0, 0.0), rotation=(0.0, 0.0, 0.0), scale=1.0, orientation0=(1.0, 0.0, 0.0), **kwargs)
    Bases: ratcave.utils.observers.AutoRegisterObserver
    XYZ Position, Scale and XYZ Euler Rotation Class.
Parameters

- **position** – \((x, y, z)\) translation values.
- **rotation** – \((x, y, z)\) rotation values
- **scale** (float) – uniform scale factor. 1 = no scaling.

**look_at** \((x, y, z)\)

Rotate so orientation is toward \((x, y, z)\) coordinates.

**model_matrix**

**normal_matrix**

**on_change**

Callback for if change detected. Meant to be overridable by subclasses.

**orientation**

The object’s orientation as a vector, calculated by rotation from orientation0, the starting orientation.

**orientation0**

Starting orientation (3-element unit vector). New orientations are calculated by rotating from this vector.

**position**

**rotation**

**scale**

**view_matrix**

**class PhysicalGraph(**kwargs)**

**Bases:** ratcave.physical.Physical, ratcave.scenegraph.SceneGraph

Object with xyz position and rotation properties that are relative to its parent.

**add_child** \((child, modify=False)\)

Adds an object as a child in the scene graph. With modify=True, model_matrix_transform gets change from identity and prevents the changes of the coordinates of the child

**model_matrix_global**

**normal_matrix_global**

**notify**

Flags Observer to perform update() at proper time.

**on_change**

Callback for if change detected. Meant to be overridable by subclasses.

**orientation_global**

Orientation vector, in world coordinates.

**position_global**

**rotation_global**

**view_matrix_global**

**coordinates.py**

**class Coordinates(**args, **kwargs)**

**Bases:** ratcave.utils.observers.IterObservable
Returns a Coordinates object

coops = {'x': 0, 'y': 1, 'z': 2}

class RotationBase
    Bases: object
        classmethod from_matrix(matrix)

rotate(vector)
    Takes a vector and returns it rotated by self.

to_euler(units='rad')

to_matrix()

to_quaternion()

class RotationEuler(x, y, z, axes='rxyz', **kwargs)
    Bases: ratcave.coordinates.RotationBase, ratcave.coordinates.Coordinates

class RotationEulerDegrees(x, y, z, axes='rxyz', **kwargs)
    Bases: ratcave.coordinates.RotationEuler

        classmethod from_matrix(matrix, axes='rxyz')

        to_degrees()

        to_euler(units='rad')

        to_matrix()

        to_quaternion()

        to_radians()

class RotationEulerRadians(x, y, z, axes='rxyz', **kwargs)
    Bases: ratcave.coordinates.RotationEuler

        classmethod from_matrix(matrix, axes='rxyz')

        to_degrees()

        to_euler(units='rad')

        to_matrix()

        to_quaternion()

        to_radians()

class RotationQuaternion(w, x, y, z, **kwargs)
    Bases: ratcave.coordinates.RotationBase, ratcave.coordinates.Coordinates

        coords = {'w': 0, 'x': 1, 'y': 2, 'z': 3}

        classmethod from_matrix(matrix)

        to_euler(units='rad')

        to_matrix()

        to_quaternion()

class Scale(*args, **kwargs)
    Bases: ratcave.coordinates.Coordinates

    to_matrix()
class Translation(*args, **kwargs)
    Bases: ratcave.coordinates.Coordinates

to_matrix()

cross_product_matrix(vec)
    Returns a 3x3 cross-product matrix from a 3-element vector.

rotation_matrix_between_vectors(from_vec, to_vec)
    Returns a rotation matrix to rotate from 3d vector “from_vec” to 3d vector “to_vec”. Equation from https://math.stackexchange.com/questions/180418/calculate-rotation-matrix-to-align-vector-a-to-vector-b-in-3d

Shader

shader.py

class HasUniforms(uniforms=None, **kwargs)
    Bases: object

    Interface for drawing. Ensures that there is a uniforms attribute to the class, which can be reset upon demand.

    reset_uniforms()

class HasUniformsUpdater(**kwargs)
    Bases: ratcave.shader.HasUniforms

    HasUniforms can not inherit from Observer, we need something on the level of Mesh, that can update the uniforms

    uniforms
        The dict-like collection of uniform values. To send data to the graphics card, simply add it as a key-value pair.

        Example:

        ```python
        mesh.uniforms['diffuse'] = 1., 1., 0.  # Will sends a 3-value vector of floats to the graphics card, when drawn.
        ```

class Shader(vert='"', frag='"', geom='"', lazy=False)

    GLSL Shader program object for rendering in OpenGL. To activate, call the Shader.bind() method, or pass it to a context manager (the ‘with’ statement).

    Examples and inspiration for shader programs can found at https://www.shadertoy.com/.

    Parameters

    • vert ("\") – The vertex shader program string
    • frag ("\") – The fragment shader program string
    • geom ("\") – The geometry shader program

    Example:

    ```python
    shader = Shader.from_file(vert='vertshader.vert', frag='fragshader.frag')
    with shader:
        mesh.draw()
    ```
bind()
Activate this Shader, making it the currently-bound program.
Any Mesh.draw() calls after bind() will have their data processed by this Shader. To unbind, call Shader.unbind().

Example:
```python
shader.bind()
mesh.draw()
shader.unbind()
```

**Note:** Shader.bind() and Shader.unbind() can be also be called implicitly by using the ‘with’ statement.

Example of with statement with Shader::
```python
with shader: mesh.draw()
```

bindfun
compile()
classmethod from_file(vert, frag, **kwargs)
Reads the shader programs, given the vert and frag filenames

**Parameters**
- `vert` (-) – The filename of the vertex shader program (ex: ‘vertshader.vert’)
- `frag` (-) – The filename of the fragment shader program (ex: ‘fragshader.frag’)

**Returns** The Shader using these files.

**Return type**
- shader (Shader)

link()
link the program, making it the active shader.

**Note:** Shader.bind() is preferred here, because link() Requires the Shader to be compiled already.

class UniformCollection(**kwargs)
Bases: collections.UserDict, object
Returns a dict-like collection of arrays that can copy itself to shader programs as GLSL Uniforms.

Uniforms can be thought of as pipes to the program on the graphics card. Variables set in UniformCollection can be directly used in the graphics card.

Example:
```python
uniforms = UniformCollection()
uniforms['diffuse'] = 1., 1., 0.
uniforms['model_matrix'] = numyp.eye(4)
```

In the shader, this would be used by initializing the uniform variable, for example:
uniform vec3 diffuse;
uniform mat4 model_matrix;

Any key-value pairs are sent to a bound shader program when UniformCollection.send() is called.

More information about GLSL Uniforms can be found at https://www.khronos.org/opengl/wiki/Uniform_(GLSL)

**Note:** This class isn’t usually constructed directly. It can be found as ‘uniforms’ attributes of Meshes and Cameras.

```
send()
```

Sends all the key-value pairs to the graphics card. These uniform variables will be available in the currently-bound shader.

**Other classes**

**vertex.py**

**class ElementArrayBuffer**(data, *args, **kwargs)

Bases: ratcave.vertex.VBO

```
target
```

**class VAO**(indices=None, **kwargs)


OpenGL Vertex Array Object. Sends array data in a Vertex Buffer to the GPU. This data can be accessed in the vertex shader using the ‘layout(location = N)’ header line, where N = the index of the array given the VAO.

Example: VAO(vertices, normals, texcoords):

Fragshader: layout(location = 0) in vec3 vertexCoord; layout(location = 1) in vec2 texCoord; layout(location = 2) in vec3 normalCoord;

```
assign_vertex_attrib_location(vbo, location)
```

Load data into a vbo

```
bindfun
draw(mode=)
```

**element_array_buffer**

**class VBO**(data, *args, **kwargs)


```
bindfun
target
```
**experimental.py**

`draw_vr_anaglyph(cube_fbo, vr_scene, active_scene, eye_poses=(0.035, -0.035))`  
Experimental anaglyph drawing function for VR system with red/blue glasses, used in Sirota lab. Draws a virtual scene in red and blue, from subject’s (heda trackers) perspective in active scene.

Note: assumes shader uses playerPos like ratcave’s default shader

**Parameters**

- `cube_fbo` – texture frameBuffer object.
- `vr_scene` – virtual scene object
- `active_scene` – active scene object
- `eye_poses` – the eye positions

**Returns:**

- `genindex`
Python Module Index

r
ratcave.camera, 35
ratcave.coordinates, 45
ratcave.experimental, 50
ratcave.light, 42
ratcave.mesh, 34
ratcave.physical, 44
ratcave.scene, 44
ratcave.scenegraph, 44
ratcave.shade, 47
ratcave.texture, 42
ratcave.vertex, 49
ratcave.wavefront, 39
Index

**A**
- `add_child()` (PhysicalGraph method), 45
- `add_child()` (SceneGraph method), 44
- `add_children()` (SceneGraph method), 44
- `aspect` (PerspectiveProjection attribute), 40
- `assign_vertex_attrib_location()` (VAO method), 49
- `attach_to_fbo()` (RenderBuffer method), 42
- `attach_to_fbo()` (Texture method), 35, 43
- `attach_to_fbo()` (TextureCube method), 43
- `attachment_point` (DepthTexture attribute), 42
- `attachment_point` (RenderBuffer attribute), 42
- `attachment_point` (Texture attribute), 43

**B**
- `bind()` (Shader method), 37, 47
- `bind()` (Texture method), 36, 43
- `bindfun` (RenderBuffer attribute), 42
- `bindfun` (Shader attribute), 48
- `bindfun` (Texture attribute), 43
- `bindfun` (VAO attribute), 49
- `bindfun` (VBO attribute), 49
- `bottom` (ScreenEdges attribute), 41

**C**
- `Camera` (class in ratcave.camera), 35, 39
- `CameraGroup` (class in ratcave.camera), 40
- `children` (SceneGraph attribute), 44
- `clear()` (Scene method), 36, 44
- `compile()` (Shader method), 37, 48
- `convergence` (StereoCameraGroup attribute), 41
- `Coordinates` (class in ratcave.coordinates), 45
- `coords` (Coordinates attribute), 46
- `coords` (OrthoProjection attribute), 40
- `coords` (RotationQuaternion attribute), 46
- `copy()` (Mesh method), 34, 38
- `copy()` (ProjectionBase method), 41
- `cross_product_matrix()` (in module ratcave.coordinates), 47

**D**
- `DepthTexture` (class in ratcave.texture), 42
- `distance` (StereoCameraGroup attribute), 41
- `draw()` (EmptyEntity method), 39
- `draw()` (Mesh method), 34, 38
- `draw()` (Scene method), 36, 44
- `draw()` (VAO method), 49
- `draw360_to_texture()` (Scene method), 36, 44
- `draw_anaglyph()` (Scene method), 36, 44
- `draw_vr_anaglyph()` (in module ratcave.experimental), 50
- `dynamic` (Mesh attribute), 34, 38

**E**
- `element_array_buffer` (VAO attribute), 49
- `ElementArrayBuffer` (class in ratcave.vertex), 49
- `EmptyEntity` (class in ratcave.mesh), 39

**F**
- `fov_y` (PerspectiveProjection attribute), 40
- `from_file()` (ratcave.shader.Shader class method), 37, 48
- `from_image()` (ratcave.texture.Texture class method), 36, 43
- `from_image()` (ratcave.texture.TextureCube class method), 43
- `from_incomplete_data()` (ratcave.mesh.Mesh class method), 34, 38
- `from_matrix()` (ratcave.coordinates.RotationBase class method), 46
- `from_matrix()` (ratcave.coordinates.RotationEulerDegrees class method), 46
- `from_matrix()` (ratcave.coordinates.RotationEulerRadians class method), 46
- `from_matrix()` (ratcave.coordinates.RotationQuaternion class method), 46
- `from_pickle()` (ratcave.camera.Camera class method), 35, 40
- `from_pickle()` (ratcave.mesh.Mesh class method), 34, 38
generate_mipmap() (Texture method), 36, 43
get_mesh() (WavefrontReader method), 39
GrayscaleTexture (class in ratcave.texture), 42
GrayscaleTextureCube (class in ratcave.texture), 42
HasUniforms (class in ratcave.shader), 47
HasUniformsUpdater (class in ratcave.shader), 47
height (Viewport attribute), 41
internal_fmt (DepthTexture attribute), 42
internal_fmt (GrayscaleTexture attribute), 42
internal_fmt (GrayscaleTextureCube attribute), 42
internal_fmt (RenderBuffer attribute), 43
internal_fmt (Texture attribute), 43
left (ScreenEdges attribute), 41
Light (class in ratcave.light), 42
link() (Shader method), 37, 48
look_at() (CameraGroup method), 40
look_at() (Physical method), 45
match_aspect_to_viewport() (PerspectiveProjection method), 40
material_property_map (WavefrontReader attribute), 39
max_texture_limit (Texture attribute), 36, 43
Mesh (class in ratcave.mesh), 34, 38
model_matrix (Physical attribute), 45
model_matrix_global (PhysicalGraph attribute), 45
name (Texture attribute), 36, 43
normal_matrix (Physical attribute), 45
normal_matrix_global (PhysicalGraph attribute), 45
normals (Mesh attribute), 34, 38
notify() (PhysicalGraph method), 45
on_change() (Physical method), 45
on_change() (PhysicalGraph method), 45
orientation (Physical attribute), 45
orientation0 (Physical attribute), 45
orientation_global (PhysicalGraph attribute), 45
origin (OrthoProjection attribute), 40
OrthoProjection (class in ratcave.camera), 40
parent (SceneGraph attribute), 44
PerspectiveProjection (class in ratcave.camera), 40
Physical (class in ratcave.physical), 44
PhysicalGraph (class in ratcave.physical), 45
pixel_fmt (DepthTexture attribute), 42
pixel_fmt (GrayscaleTexture attribute), 42
pixel_fmt (GrayscaleTextureCube attribute), 42
pixel_fmt (Texture attribute), 43
points (Mesh attribute), 39
position (Physical attribute), 45
position_global (PhysicalGraph attribute), 45
projection (Camera attribute), 35, 40
projection_matrix (Camera attribute), 35, 40
projection_matrix (ProjectionBase attribute), 41
ProjectionBase (class in ratcave.camera), 40
ratcave.camera (module), 35, 39
ratcave.coordinates (module), 39
ratcave.experimental (module), 50
ratcave.light (module), 42
ratcave.mesh (module), 34, 38
ratcave.physical (module), 44
ratcave.scene (module), 36, 44
ratcave.scenegraph (module), 44
ratcave.shader (module), 36, 47
ratcave.texture (module), 35, 42
ratcave.vertex (module), 49
ratcave.wavefront (module), 39
remove_children() (SceneGraph method), 44
RenderBuffer (class in ratcave.texture), 42
reset_uniforms() (Camera method), 35, 40
reset_uniforms() (EmptyEntity method), 39
reset_uniforms() (HasUniforms method), 47
reset_uniforms() (Light method), 42
reset_uniforms() (Mesh method), 35, 39
reset_uniforms() (Texture method), 36, 43
right (ScreenEdges attribute), 41
rotate() (RotationBase method), 46
rotation (Physical attribute), 45
rotation_global (PhysicalGraph attribute), 45
rotation_matrix_between_vectors() (in module ratcave.coordinates), 47
RotationBase (class in ratcave.coordinates), 46
RotationEuler (class in ratcave.coordinates), 46
RotationEulerDegrees (class in ratcave.coordinates), 46
RotationEulerRadians (class in ratcave.coordinates), 46
RotationQuaternion (class in ratcave.coordinates), 46
Scale (class in ratcave.coordinates), 46
scale (Physical attribute), 45
Scene (class in ratcave.scene), 36, 44
SceneGraph (class in ratcave.scenegraph), 44
ScreenEdges (class in ratcave.camera), 41
send() (UniformCollection method), 49
Shader (class in ratcave.shader), 36, 47
slot (Texture attribute), 36, 43
StereoCameraGroup (class in ratcave.camera), 41

T
target (ElementArrayBuffer attribute), 49
target (RenderBuffer attribute), 43
target (Texture attribute), 43
target (TextureCube attribute), 43
target (VBO attribute), 49
target0 (Texture attribute), 43
target0 (TextureCube attribute), 43
texcoords (Mesh attribute), 35, 39
Texture (class in ratcave.texture), 35, 43
texture (Mesh attribute), 35, 39
TextureCube (class in ratcave.texture), 43
to_degrees() (RotationEulerDegrees method), 46
to_degrees() (RotationEulerRadians method), 46
to_euler() (RotationBase method), 46
to_euler() (RotationEulerDegrees method), 46
to_euler() (RotationEulerRadians method), 46
to_euler() (RotationQuaternion method), 46
to_matrix() (RotationBase method), 46
to_matrix() (RotationEulerDegrees method), 46
to_matrix() (RotationEulerRadians method), 46
to_matrix() (RotationQuaternion method), 46
to_matrix() (Scale method), 46
to_matrix() (Translation method), 47
to_pickle() (Camera method), 35, 40
to_pickle() (Mesh method), 35, 39
to_quaternion() (RotationBase method), 46
to_quaternion() (RotationEulerDegrees method), 46
to_quaternion() (RotationEulerRadians method), 46
to_quaternion() (RotationQuaternion method), 46
to_radians() (RotationEulerDegrees method), 46
to_radians() (RotationEulerRadians method), 46
top (ScreenEdges attribute), 41
Translation (class in ratcave.coordinates), 46
triangles (Mesh attribute), 39

U
unbind() (Texture method), 36, 43
UniformCollection (class in ratcave.shader), 48
uniforms (HasUniformsUpdater attribute), 47
update() (ProjectionBase method), 41

V
VAO (class in ratcave.vertex), 49
VBO (class in ratcave.vertex), 49
vertices (Mesh attribute), 35, 39
vertices_global (Mesh attribute), 35, 39
vertices_local (Mesh attribute), 35, 39
view_matrix (Physical attribute), 45
view_matrix_global (PhysicalGraph attribute), 45

W
WavefrontReader (class in ratcave.wavefront), 39
width (Viewport attribute), 42

X
x (Viewport attribute), 42
x_shift (PerspectiveProjection attribute), 40

Y
y (Viewport attribute), 42
y_shift (PerspectiveProjection attribute), 40

Z
z_far (ProjectionBase attribute), 41
z_near (ProjectionBase attribute), 41