vispy Documentation

Release 0.1

vispy contributers

August 24, 2013
Vispy is a collaborative project that has the goal to allow more sharing of code between visualization projects based on OpenGL. It does this by providing powerful interfaces to OpenGL, at different levels of abstraction and generality.

This is the auto-generated API documentation for the vispy library. See http://vispy.org for more information on the vispy project.

Contents:
OPENGL ES 2.0 API

The `vispy.gl` namespace provides the OpenGL ES 2.0 API, consisting of 302 constants and 144 functions (with 79 and 15 more in the extensions). At this moment, the functions are taken from `OpenGL.GL` (provided by the PyOpenGL package).

`vispy.gl`

- `GL_ACTIVE_ATTRIBUTES`
- `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`
- `GL_ACTIVE_TEXTURE`
- `GL_ACTIVE_UNIFORMS`
- `GL_ACTIVE_UNIFORM_MAX_LENGTH`
- `GL_ALIASED_LINE_WIDTH_RANGE`
- `GL_ALIASED_POINT_SIZE_RANGE`
- `GL_ALPHA`
- `GL_ALPHA_BITS`
- `GL_ALWAYS`
- `GL_ARRAY_BUFFER`
- `GL_ARRAY_BUFFER_BINDING`
- `GL_ATTACHED_SHADERS`
- `GL_BACK`
- `GL_BLEND`
- `GL_BLEND_COLOR`
- `GL_BLEND_DST_ALPHA`
- `GL_BLEND_DST_RGB`
- `GL_BLEND_EQUATION`
- `GL_BLEND_EQUATION_ALPHA`
- `GL_BLEND_EQUATION_RGB`
- `GL_BLEND_SRC_ALPHA`
- `GL_BLEND_SRC_RGB`
• GL_BLUE_BITS
• GL_BOOL
• GL_BOOL_VEC2
• GL_BOOL_VEC3
• GL_BOOL_VEC4
• GL_BUFFER_SIZE
• GL_BUFFER_USAGE
• GL_BYTE
• GL_CCW
• GL_CLAMP_TO_EDGE
• GL_COLOR_ATTACHMENT0
• GL_COLOR_BUFFER_BIT
• GL_COLOR_CLEAR_VALUE
• GL_COLOR_WRITEMASK
• GL_COMPILE_STATUS
• GL_COMPRESSED_TEXTURE_FORMATS
• GL_CONSTANT_ALPHA
• GL_CONSTANT_COLOR
• GL_CULL_FACE
• GL_CULL_FACE_MODE
• GL_CURRENT_PROGRAM
• GL_CURRENT_VERTEX_ATTRIB
• GL_CW
• GL_DECR
• GL_DECR_WRAP
• GL_DELETE_STATUS
• GL_DEPTH_ATTACHMENT
• GL_DEPTH_BITS
• GL_DEPTH_BUFFER_BIT
• GL_DEPTH_CLEAR_VALUE
• GL_DEPTH_COMPONENT
• GL_DEPTH_COMPONENT16
• GL_DEPTH_FUNC
• GL_DEPTH_RANGE
• GL_DEPTH_TEST
• GL_DEPTH_WRITEMASK
• GL_DITHER
• GL_DONT_CARE
• GL_DST_ALPHA
• GL_DST_COLOR
• GL_DYNAMIC_DRAW
• GL_ELEMENT_ARRAY_BUFFER
• GL_ELEMENT_ARRAY_BUFFER_BINDING
• GL_EQUAL
• GL_ES_VERSION_2_0
• GL_EXTENSIONS
• GL_FALSE
• GL_FASTEST
• GL_FIXED
• GL_FLOAT
• GL_FLOAT_MAT2
• GL_FLOAT_MAT3
• GL_FLOAT_MAT4
• GL_FLOAT_VEC2
• GL_FLOAT_VEC3
• GL_FLOAT_VEC4
• GL_FRAGMENT_SHADER
• GL_FRAMEBUFFER
• GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME
• GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE
• GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE
• GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL
• GL_FRAMEBUFFER_BINDING
• GL_FRAMEBUFFER_COMPLETE
• GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT
• GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS
• GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT
• GL_FRAMEBUFFER_UNSUPPORTED
• GL_FRONT
• GL_FRONT_AND_BACK
• GL_FRONT_FACE
• GL_FUNC_ADD
• GL_FUNC_REVERSE_SUBTRACT
• GL_FUNC_SUBTRACT
• GL_GENERATE_MIPMAP_HINT
• GL_GEQUAL
• GL_GREATER
• GL_GREEN_BITS
• GL_HIGH_FLOAT
• GL_HIGH_INT
• GL_IMPLEMENTATION_COLOR_READ_FORMAT
• GL_IMPLEMENTATION_COLOR_READ_TYPE
• GL_INCR
• GL_INCR_WRAP
• GL_INFO_LOG_LENGTH
• GL_INT
• GL_INT_VEC2
• GL_INT_VEC3
• GL_INT_VEC4
• GL_INVALID_ENUM
• GL_INVALID_FRAMEBUFFER_OPERATION
• GL_INVALID_OPERATION
• GL_INVALID_VALUE
• GL_INVERT
• GL_KEEP
• GL_LEQUAL
• GL_LESS
• GL_LINEAR
• GL_LINEAR_MIPMAP_LINEAR
• GL_LINEAR_MIPMAP_NEAREST
• GL_LINES
• GL_LINE_LOOP
• GL_LINE_STRIP
• GL_LINE_WIDTH
• GL_LINK_STATUS
• GL_LOW_FLOAT
• GL_LOW_INT
• GL_LUMINANCE
• GL_LUMINANCE_ALPHA
• GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
• GL_MAX_CUBE_MAP_TEXTURE_SIZE
• GL_MAX_FRAGMENT_UNIFORM_VECTORS
• GL_MAX_RENDERBUFFER_SIZE
• GL_MAX_TEXTURE_IMAGE_UNITS
• GL_MAX_TEXTURE_SIZE
• GL_MAX_VARYING_VECTORS
• GL_MAX_VERTEX_ATTRIBS
• GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
• GL_MAX_VERTEX_UNIFORM_VECTORS
• GL_MAX_VIEWPORT_DIMS
• GL_MEDIUM_FLOAT
• GL_MEDIUM_INT
• GL_MIRRORED_REPEAT
• GL_NEAREST
• GL_NEAREST_MIPMAP_LINEAR
• GL_NEAREST_MIPMAP_NEAREST
• GL_NEVER
• GL_NICEST
• GL_NONE
• GL_NOTEQUAL
• GL_NO_ERROR
• GL_NUM_COMPRESSED_TEXTURE_FORMATS
• GL_NUM_SHADER_BINARY_FORMATS
• GL_ONE
• GL_ONE_MINUS_CONSTANT_ALPHA
• GL_ONE_MINUS_CONSTANT_COLOR
• GL_ONE_MINUS_DST_ALPHA
• GL_ONE_MINUS_DST_COLOR
• GL_ONE_MINUS_SRC_ALPHA
• GL_ONE_MINUS_SRC_COLOR
• GL_OUT_OF_MEMORY
• GL_PACK_ALIGNMENT
• GL_POINTS
• GL_POLYGON_OFFSET_FACTOR
• GL_POLYGON_OFFSET_FILL
• GL_POLYGON_OFFSET_UNITS
• GL_RED_BITS
• GL_RENDERBUFFER
• GL_RENDERBUFFER_ALPHA_SIZE
• GL_RENDERBUFFER_BINDING
• GL_RENDERBUFFER_BLUE_SIZE
• GL_RENDERBUFFER_DEPTH_SIZE
• GL_RENDERBUFFER_GREEN_SIZE
• GL_RENDERBUFFER_HEIGHT
• GL_RENDERBUFFER_INTERNAL_FORMAT
• GL_RENDERBUFFER_RED_SIZE
• GL_RENDERBUFFER_STENCIL_SIZE
• GL_RENDERBUFFER_WIDTH
• GL_RENDERER
• GL_REPEAT
• GL_REPLACE
• GL_RGB
• GL_RGB565
• GL_RGB5_A1
• GL_RGBA
• GL_RGBA4
• GL_SAMPLER_2D
• GL_SAMPLER_CUBE
• GL_SAMPLES
• GL_SAMPLE_ALPHA_TO_COVERAGE
• GL_SAMPLE_BUFFERS
• GL_SAMPLE_COVERAGE
• GL_SAMPLE_COVERAGE_INVERT
• GL_SAMPLE_COVERAGE_VALUE
• GL_SCISSOR_BOX
• GL_SCISSOR_TEST
• GL_SHADER_BINARY_FORMATS
• GL_SHADER_COMPILER
• GL_SHADER_SOURCE_LENGTH
• GL_SHADER_TYPE
• GL_SHADING_LANGUAGE_VERSION
• GL_SHORT
• GL_SRC_ALPHA
• GL_SRC_ALPHA_SATURATE
• GL_SRC_COLOR
• GL_STATIC_DRAW
• GL_STENCIL_ATTACHMENT
• GL_STENCIL_BACK_FAIL
• GL_STENCIL_BACK_FUNC
• GL_STENCIL_BACK_PASS_DEPTH_FAIL
• GL_STENCIL_BACK_PASS_DEPTH_PASS
• GL_STENCIL_BACK_REF
• GL_STENCIL_BACK_VALUE_MASK
• GL_STENCIL_BACK_WRITEMASK
• GL_STENCIL_BITS
• GL_STENCIL_BUFFER_BIT
• GL_STENCIL_CLEAR_VALUE
• GL_STENCIL_FAIL
• GL_STENCIL_FUNC
• GL_STENCIL_INDEX8
• GL_STENCIL_PASS_DEPTH_FAIL
• GL_STENCIL_PASS_DEPTH_PASS
• GL_STENCIL_REF
• GL_STENCIL_TEST
• GL_STENCIL_VALUE_MASK
• GL_STENCIL_WRITEMASK
• GL_STREAM_DRAW
• GL_SUBPIXEL_BITS
• GL_TEXTURE
• GL_TEXTURE0
• GL_TEXTURE1
• GL_TEXTURE10
• GL_TEXTURE11
• GL_TEXTURE12
• GL_TEXTURE13
• GL_TEXTURE14
• GL_TEXTURE15
• GL_TEXTURE16
• GL_TEXTURE17
• GL_TEXTURE18
• GL_TEXTURE19
• GL_TEXTURE2
• GL_TEXTURE20
• GL_TEXTURE21
• GL_TEXTURE22
• GL_TEXTURE23
• GL_TEXTURE24
• GL_TEXTURE25
• GL_TEXTURE26
• GL_TEXTURE27
• GL_TEXTURE28
• GL_TEXTURE29
• GL_TEXTURE3
• GL_TEXTURE30
• GL_TEXTURE31
• GL_TEXTURE4
• GL_TEXTURE5
• GL_TEXTURE6
• GL_TEXTURE7
• GL_TEXTURE8
• GL_TEXTURE9
• GL_TEXTURE_2D
• GL_TEXTURE_BINDING_2D
• GL_TEXTURE_BINDING_CUBE_MAP
• GL_TEXTURE_CUBE_MAP
• GL_TEXTURE_CUBE_MAP_NEGATIVE_X
• GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
• GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
• GL_TEXTURE_CUBE_MAP_POSITIVE_X
• GL_TEXTURE_CUBE_MAP_POSITIVE_Y
• GL_TEXTURE_CUBE_MAP_POSITIVE_Z
• GL_TEXTURE_MAG_FILTER
• GL_TEXTURE_MIN_FILTER
• GL_TEXTURE_WRAP_S
• GL_TEXTURE_WRAP_T
• GL_TRIANGLES
• GL_TRIANGLE_FAN
• GL_TRIANGLE_STRIP
• GL_TRUE
• GL_UNPACK_ALIGNMENT
• GL_UNSIGNED_BYTE
• GL_UNSIGNED_INT
• GL_UNSIGNED_SHORT
• GL_UNSIGNED_SHORT_4_4_4_4
• GL_UNSIGNED_SHORT_5_5_5_1
• GL_UNSIGNED_SHORT_5_6_5
• GL_VALIDATE_STATUS
• GL_VENDOR
• GL_VERSION
• GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING
• GL_VERTEX_ATTRIB_ARRAY_ENABLED
• GL_VERTEX_ATTRIB_ARRAY_NORMALIZED
• GL_VERTEX_ATTRIB_ARRAY_POINTER
• GL_VERTEX_ATTRIB_ARRAY_SIZE
• GL_VERTEX_ATTRIB_ARRAY_STRIDE
• GL_VERTEX_ATTRIB_ARRAY_TYPE
• GL_VERTEX_SHADER
• GL_VIEWPORT
• GL_ZERO
• glActiveTexture()
• glAttachShader()
• glBindAttribLocation()
• glBindBuffer()
• glBindFramebuffer()
• glBindRenderbuffer()
• glBindTexture()
• glBlendColor()
• glBlendEquation()
• glEnableVertexAttribArray()
• glFinish()
• glFlush()
• glFlushcounter()
• glGetActiveAttrib()
• glGetActiveUniform()
• glGetActiveUniformBlockName()
• glGetActiveUniformBlockIndex()
• glGetActiveUniformsiv()
• glGetAttachedShaders()
• glGetAttribLocation()
• glGetBufferParameteriv()
• glGetBufferSubData()
• glGetBuffersiv()
• glGetCacheHint()
• glGetCapabilities()
• glGetCameraParameterui64v()
• glGetColorSpaceiv()
• glGetColorSpaceNameiv()
• glGetColorSpaceStringiv()
• glGetColorSpaceiv()
• glGetColorSpaceNameiv()
• glGetColorSpaceStringiv()
• glGetColorTable()
• glGetCompressedTexImage2D()
• glGetCompressedTexImage3D()
• glGetCompressedTexImage4D()
• glGetCompressedTexImage1D()
• glGetCompressedTexImage2D()
- glFlush()
- glFramebufferRenderbuffer()
- glFramebufferTexture2D()
- glFrontFace()
- glGenBuffers()
- glGenFramebuffers()
- glGenRenderbuffers()
- glGenTextures()
- glGenerateMipmap()
- glGetActiveAttrib()
- glGetActiveUniform()
- glGetAttachedShaders()
- glGetAttribLocation()
- glGetBooleanv()
- glGetBufferParameteriv()
- glGetError()
- glGetFloatv()
- glGetFramebufferAttachmentParameteriv()
- glGetIntegerv()
- glGetProgramInfoLog()
- glGetProgramiv()
- glGetRenderbufferParameteriv()
- glGetShaderInfoLog()
- glGetShaderPrecisionFormat()
- glGetShaderSource()
- glGetShaderiv()
- glGetString()
- glGetTexParameterfv()
- glGetTexParameteriv()
- glGetUniformLocation()
- glGetUniformfv()
- glGetUniformiv()
- glGetVertexAttribPointerv()
- glGetVertexAttribfv()
- glGetVertexAttribiv()
- glHint()
• glIsBuffer()
• glIsEnabled()
• glIsFramebuffer()
• glIsProgram()
• glIsRenderbuffer()
• glIsShader()
• glIsTexture()
• glLineWidth()
• glLinkProgram()
• glPixelStorei()
• glPolygonOffset()
• glReadPixels()
• glReleaseShaderCompiler()
• glRenderbufferStorage()
• glSampleCoverage()
• glScissor()
• glShaderBinary()
• glShaderSource()
• glStencilFunc()
• glStencilFuncSeparate()
• glStencilMask()
• glStencilMaskSeparate()
• glStencilOp()
• glStencilOpSeparate()
• glTexImage2D()
• glTexParameter()
• glTexParameterf()
• glTexParameterfv()
• glTexParameteri()
• glTexParameteriv()
• glTexSubImage2D()
• glUniform1f()
• glUniform1fv()
• glUniform1i()
• glUniform1iv()
• glUniform2f()
• glUniform2fv()
• glUniform2i()
• glUniform2iv()
• glUniform3f()
• glUniform3fv()
• glUniform3i()
• glUniform3iv()
• glUniform4f()
• glUniform4fv()
• glUniform4i()
• glUniform4iv()
• glUniformMatrix2fv()
• glUniformMatrix3fv()
• glUniformMatrix4fv()
• glUseProgram()
• glValidateProgram()
• glVertexAttrib1f()
• glVertexAttrib1fv()
• glVertexAttrib2f()
• glVertexAttrib2fv()
• glVertexAttrib3f()
• glVertexAttrib3fv()
• glVertexAttrib4f()
• glVertexAttrib4fv()
• glVertexAttribPointer()
• glViewport()
• glhelper()

**vispy.gl.ext**

• GL_ALPHA8
• GL_BUFFER_ACCESS
• GL_BUFFER_MAPPED
• GL_BUFFER_MAP_POINTER
• GL_DEPTH24_STENCIL8
• GL_DEPTH_COMPONENT16
• GL_DEPTH_COMPONENT24
• GL_DEPTH_COMPONENT32
• GL_DEPTH_STENCIL
• GL_EGL_image
• GL_EGL_image_external
• GL_ETC1_RGB8
• GL_FRAGMENT_SHADER_DERIVATIVE_HINT
• GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET
• GL_FRAMEBUFFER_UNDEFINED
• GL_HALF_FLOAT
• GL_INT_10_10_10_2
• GL_LUMINANCE4_ALPHA4
• GL_LUMINANCE8
• GL_LUMINANCE8_ALPHA8
• GL_MAX_3D_TEXTURE_SIZE
• GL_NUM_PROGRAM_BINARY_FORMATS
• GL_PALETTE4_R5_G6_B5
• GL_PALETTE4_RGB5_A1
• GL_PALETTE4_RGB8
• GL_PALETTE4_RGBA4
• GL_PALETTE4_RGBA8
• GL_PALETTE8_R5_G6_B5
• GL_PALETTE8_RGB5_A1
• GL_PALETTE8_RGB8
• GL_PALETTE8_RGBA4
• GL_PALETTE8_RGBA8
• GL_PROGRAM_BINARY_FORMATS
• GL_PROGRAM_BINARY_LENGTH
• GL_REQUIRED_TEXTURE_IMAGE_UNITS
• GL_RGB565
• GL_RGB5_A1
• GL_RGB8
• GL_RGBA4
• GL_RGBA8
• GL_SAMPLER_3D
• GL_SAMPLER_EXTERNAL
• GL_STENCIL_INDEX1
• GL_STENCIL_INDEX4
• GL_TEXTURE_3D
• GL_TEXTURE_BINDING_3D
• GL_TEXTURE_BINDING_EXTERNAL
• GL_TEXTURE_EXTERNAL
• GL_TEXTURE_WRAP_R
• GL_UNSIGNED_INT_10_10_10_2
• GL_UNSIGNED_INT_24_8
• GL_VERTEX_ARRAY_BINDING
• GL_WRITE_ONLY
• GL_compressed_ETC1_RGB8_texture
• GL_compressed_paletted_texture
• GL_depth24
• GL_depth32
• GL_depth_texture
• GL_element_index_uint
• GL_fbo_render_mipmap
• GL_fragment_precision_high
• GL_get_program_binary
• GL_mapbuffer
• GL_packed_depth_stencil
• GL_required_internalformat
• GL_rgb8_rgba8
• GL_standard_derivatives
• GL_stencil1
• GL_stencil4
• GL_surfaceless_context
• GL_texture_3D
• GL_texture_float
• GL_texture_float_linear
• GL_texture_half_float
• GL_texture_half_float_linear
• GL_texture_npot
• GL_vertex_array_object
• GL_vertex_half_float
• GL_vertex_type_10_10_10_2
• glBindVertexArray()
• glCompressedTexImage3D()
• glCompressedTexSubImage3D()
• glCopyTexSubImage3D()
• glDeleteVertexArrays()
• glFramebufferTexture3D()
• glGenVertexArrays()
• glGetBufferPointerv()
• glGetProgramBinary()
• glIsVertexArray()
• glMapBuffer()
• glProgramBinary()
• glTexImage3D()
• glTexSubImage3D()
• glUnmapBuffer()
THE OBJECT ORIENTED OPENGL API (OOGL)

(name of the module is subject to change) Object oriented interface to OpenGL.

This module implements classes for most things that are “objects” in OpenGL, such as textures, FBO’s, VBO’s and shaders. Further, some convenience classes are implemented (like the collection class?).

This set of classes provides a friendly (Pythonic) interface to OpenGL, and is designed to provide OpenGL’s full functionality.

Central to each visualization is the ShaderProgram. To enable it, it should be used as a context manager. Other objects, such as Texture2D and VertexBuffer should be set as uniforms and attributes of the ShaderProgram object.

Example:

```python
# Init
program = ShaderProgram(...)  # ShaderProgram...
program.attributes['a_position'] = VertexBuffer(my_positions_array)

... 

# Paint event handler
with program:
    program.uniforms['u_color'] = 0.0, 1.0, 0.0
    program.draw_arrays(gl.GL_TRIANGLES)
```

The oogl classes:

- ShaderProgram
- FragmentShader and VertexShader
- VertexBuffer and ElementBuffer
- Texture2D, Texture3D, TextureCubeMap
- FrameBuffer
- RenderBuffer

Note: With vispy.oogl we strive to offer a Python interface that provides the full functionality of OpenGL. However, this layer is a work in progress and there are yet a few known limitations. Most notably:

- TextureCubeMap is not yet implemented
- FBO’s can only to 2D textures (not 3D textures or cube maps)
• Sharing of Shaders and RenderBuffers (between multiple ShaderProgram and FrameBuffers, respectively) is not well supported.

• We’re having some problems with point sprites due to incompatibilities between OpenGL ES 2.0 and normal OpenGL.

• There is no support for texture mipmaping yet

• No support for compressed textures.

• Besides the above, there might be the occasional bug, please report!

---

class vispy.oogl.GLObject
Base class for classes that wrap an OpenGL object. All GObject’s can be used as a context manager to enable them, although some are better used by setting them as a uniform or attribute of a ShaderProgram.

All GObject’s apply deferred (a.k.a. lazy) loading, which means that the objects can be created and data can be set even if no OpenGL context is available yet.

There are a few exceptions, most notably when enabling an object by using it as a context manager or via ShaderProgram.enable_object(), and the delete method. In these cases, the called should ensure that the proper OpenGL context is current.

def delete()
    Delete the object from OpenGL memory. Note that the right context should be active when this method is called.

handle
    The handle (i.e. id or name) of the underlying OpenGL object.

class vispy.oogl.ShaderProgram(*shaders)
Representation of a shader program. It combines (links) one or more vertex and fragment shaders to compose a complete program.

Objects of this class are also used to set the uniforms and attributes that are used by the shaders. To do so, simply add attributes to the uniforms and attributes members. The names of the added attributes should match with those used in the shaders.

attach_shader(shader)
    Attach the given vertex or fragment shader to this shader program. Multiple shaders can be attached (also e.g. multiple FragmentShaders).

attributes
    A dictionary for the attribute inputs to this shader program. For example:
    program.attributes[’a_position’] = my_positions_array.

    Attributes can be a tuple of 1 to 4 elements (global attributes), a numpy array of per vertex attributes, or a VertexBuffer object (recommended over the numpy array).

    Note that one can use prog.attributes.update(my_stuctured_array) or prog.attributes.update(my_stuctured_vbo) to map field names to attribute names automatically.

detach_shader(shader)
    Detach the given shader from this shader program.

draw_arrays(mode, first=None, count=None)
    Draw the attribute arrays in the specified mode. Only call when the program is enabled.

first [int] The starting vertex index in the vertex array. Default 0.

count [int] The number of vertices to draw. Default all.

draw_elements (mode, indices)
Draw the attribute arrays using a specified set of vertices, in the specified mode. Only call when the program is enabled.


indices [numpy_array or ElementBuffer] The indices to the vertices in the vertex arrays to draw. For performance, ElementBuffer objects are recommended over numpy arrays. If an ElementBuffer is provided, this method takes care of enabling it.

enable_object (object)
Enable an object, e.g. a texture. The program will make sure that the object is disabled again. Can only be called while being used in a context.

shaders
List of shaders associated with this shading program.

uniforms
A dictionary for the uniform inputs to this shader program. For example:

program.uniforms['u_color'] = 0.0, 1.0, 0.0.

Uniforms can be a tuple/array of 1 to 4 elements to specify a vector, 4, 9 or 16 elements to specify a matrix, or a Texture object to specify a sampler.

---

class vispy.oogl.VertexShader (source=None)
Representation of a vertex shader object. Inherits BaseShader.

class vispy.oogl.FragmentShader (source=None)
Representation of a fragment shader object. Inherits BaseShader.

class vispy.oogl.shader.BaseShader (type, source=None)
Abstract shader class.

add_source (source)
Templating, for later.

set_source (source)
Set the source of the shader.

---

class vispy.oogl.VertexBuffer (data=None, buffer=None)
Representation of vertex buffer object of type GL_ARRAY_BUFFER, which can be used to store vertex data. Inherits from Buffer.

To use a VertexBuffer, set it as a member of a_shader_program.attributes.

set_data (data, offset=None)
Set vertex attribute data, or a part of it. If the data matches the current size of the buffer, the data is updated faster.

The data can be (and is recommended to be) a numpy array with dtype fields, one for each vertex attribute. In that way, the buffer represents an “array of structures” topology which is good for performance.
VertexBufferView instances can be created that are associated with the same underlying Buffer. This can be done by slicing or by indexing using the appropriate field name. VertexBufferView instances can themselves be sliced/indexed as well.

**data**  [numpy array] The data to set.

**offset**  [int] The offset, to update part of the buffer. Optional.

**shape**  The shape of the data that this VertexBuffer represents. This is always a 2-element tuple (size, vector_size).

**type**  The name of the dtype of the data. If the data is a numpy array with fields, this yields a list of field specifiers.

class `vispy.oogl.ElementBuffer(data=None)`  
Representation of vertex buffer object of type GL_ELEMENT_ARRAY_BUFFER, which can be used to store indices to vertex data. Inherits from Buffer.

To use an ElementBuffer, enable it before drawing. When enabled, the indices pointer in glDrawElements becomes a byte offset.

**count**  The number of indices that this ElementBuffer represents.

**set_data** *(data, offset=None)*  
Set vertex element data, or a part of it. If the data matches the current size of the buffer, the data is updated faster.

**data**  [numpy array] The data to set.

**offset**  [int] The offset, to update part of the buffer. Optional.

**type**  The name of the dtype of the data. If the data is a numpy array with fields, this yields a list of field specifiers.

class `vispy.oogl.vbo.Buffer(target, data=None)`  
The buffer is used to store vertex data. It is recommended to use one of the subclasses: VertexBuffer or ElementBuffer.

**nbytes**  The number of bytes that the buffer uses.

**set_data** *(data, offset=None)*  
Set the data for this buffer. If the size of the given data matches the current buffer size, the data is updated in a fast manner.

Users typically do not use this class directly. The buffer class does not care about type of shape; it’s just a collection of bytes. The VertexBuffer and ElementBuffer wrap the Buffer class to give meaning to the data.

class `vispy.oogl.Texture2D(*args, **kwargs)`  
Representation of a 2D texture. Inherits Texture.

class `vispy.oogl.Texture3D(*args, **kwargs)`  
Representation of a 3D texture. Note that for this the GL_texture_3D extension needs to be available. Inherits Texture.

class `vispy.oogl.TextureCubeMap(*args, **kwargs)`  
Representation of a cube map, to store texture data for the 6 sided of a cube. Used for instance to create environment mappings.
This class is not yet implemented.

```python
class vispy.oogl.Texture(target, data=None, format=None, clim=None)
Representation of an OpenGL texture.

set_data(data, level=0, format=None, clim=None)
Set the data for this texture. This method can be called at any time (even if there is no context yet).
It is relatively cheap to call this function multiple times, only the last data is set right before drawing. If
the shape of the given data matches the shape of the current texture, the data is updated in a fast manner.

data [numpy array] The texture data to set.
level [int] The mipmap level. Default 0.
format [OpenGL enum] The format representation of the data. If not given or None, it is decuced from
the given data. Can be GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE_ALPHA,
GL_ALPHA.
clim [(min, max)] Contrast limits for the data. If specified, min will end up being 0.0 (black) and max will
end up as 1.0 (white). If not given or None, clim is determined automatically. For floats they become
(0.0, 1.0). For integers the are mapped to the full range of the type.

set_filter(mag_filter, min_filter)
Set interpolation filters. Either parameter can be None to not (re)set it.
mag_filter [GL_ENUM or string] The magnification filter (when texels are larger than screen pixels). Can be GL_NEAREST, GL_LINEAR.
min_filter [GL_ENUM or string] The minification filter (when texels are smaller than screen pixels). For this filter, mipmapping can be applied to perform an-
tialiasing (if mipmaps are available for this texture). Can be GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR,
GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR.

set_storage(shape, level=0, format=None)
Allocate storage for this texture. This is useful if the texture is used as a render target for an FBO.
A call that only uses the shape argument does not result in an action if the call would not change the shape.
shape [tuple] The shape of the “virtual” data. By specifying e.g. (20,20,3) for a Texture2D, one implicitly
sets the format to GL_RGB. Note that shape[0] is height.
level [int] The mipmap level. Default 0.
format [OpenGL enum] The format representation of the data. If not given or None, it is decuced from
the given data. Can be GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE_ALPHA,
GL_ALPHA.

set_subdata(offset, data, level=0, format=None, clim=None)
Set a region of data for this texture. This method can be called at any time (even if there is no context yet).
In contrast to set_data(), each call to this method results in an OpenGL api call.
offset [tuple] The offset for each dimension, to update part of the texture.
data [numpy array] The texture data to set. The data (with offset) cannot exceed the boundaries of the
current texture.
level [int] The mipmap level. Default 0.
format [OpenGL enum] The format representation of the data. If not given or None, it is decued from
the given data. Can be GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE_ALPHA,
GL_ALPHA.
```
**clim** [(min, max)] Contrast limits for the data. If specified, min will end up being 0.0 (black) and max will end up as 1.0 (white). If not given or None, clim is determined automatically. For floats they become (0.0, 1.0). For integers the are mapped to the full range of the type.

**set_wrapping**(wrapx, wrapy, wrapz=None)
Set texture coordinate wrapping.

**wrapx** [GL_ENUM or string] The wrapping mode in the x-direction. Can be GL_REPEAT, GL_CLAMP_TO_EDGE, GL_MIRRORED_REPEAT.

**wrapy** [GL_ENUM or string] Dito for y.

**wrapz** [GL_ENUM or string] Dito for z. Only makes sense for 3D textures, and requires the texture_3d extension. Optional.

---

**class** **vispy.oogl.FrameBuffer**(color=None, depth=None, stencil=None)
Representation of a frame buffer object (a.k.a. FBO). FrameBuffers allow off-screen rendering instead of to the screen. This is for instance used for special effects and post-processing.

**attach_color**(object, level=0)
Attach a RenderBuffer of Texture instance to collect color output for this FrameBuffer. Pass None for object to detach the attachment. If a texture is given, level specifies the mipmap level (default 0).

**attach_depth**(object, level=0)
Attach a RenderBuffer of Texture instance to collect depth output for this FrameBuffer. Pass None for object to detach the attachment. If a texture is given, level specifies the mipmap level (default 0).

**attach_stencil**(object)
Attach a RenderBuffer instance to collect stencil output for this FrameBuffer. Pass None for object to detach the attachment.

**color_attachment**
Get the color attachment.

**depth_attachment**
Get the depth attachment.

**set_size**(width, height)
Convenience function to set the space allocated for all attachments in use.

**stencil_attachment**
Get the stencil attachment.

**class** **vispy.oogl.RenderBuffer**(shape=None, format=None)
Representation of a render buffer, to be attached to a FrameBuffer object.

**set_storage**(shape, format=None)
Allocate storage for this render buffer.

This function can be repeatedly called without much cost if the shape is not changed.

In general, it’s easier to just call FrameBuffer.set_size() to allocate space for all attachments.

**shape** [tuple] The shape of the “virtual” data. Note that shape[0] is height.

**format** [OpenGL enum] The format representation of the data. If not given or None, it is determined automatically depending on the shape and the kind of atatchment. Can be GL_RGB565, GL_RGBA4, GL_RGB5_A1, GL_RGB8, GL_RGBA8, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT24, GL_DEPTH_COMPONENT32, GL_STENCIL_INDEX8, GL_STENCIL_INDEX1, GL_STENCIL_INDEX4.
The visuals layer implements several visuals, object that represent a certain kind of visualisation. Examples are lines, markers, images, volumes, meshes, barplots, axes, grids, etc.

This layer is planned; it does not exist yet.
THE EVENT MODULE

The event module implements the classes that make up the event system. The Event class and its subclasses are used to represent "stuff that happens". The EventEmitter class provides an interface to connect to events and to emit events. The EmitterGroup groups EventEmitter objects.

For more information see [http://github.com/vispy/vispy/wiki/API_Events](http://github.com/vispy/vispy/wiki/API_Events)

class `vispy.event.Event` *(type, native=None, **kwds)*

Class describing events that occur and can be reacted to with callbacks. Each event instance contains information about a single event that has occurred such as a key press, mouse motion, timer activation, etc.

**Subclasses**: `KeyEvent`, `MouseEvent`, `TouchEvent`, `StylusEvent`

The creation of events and passing of events to the appropriate callback functions in the responsibility of `EventEmitter` instances.

Note that each event object has an attribute for each of the input arguments listed below.

- **type** [str] String indicating the event type (e.g. mouse_press, key_release)
- **native** [object (optional)] The native GUI event object
- ****kwds [keyword arguments] All extra keyword arguments become attributes of the event object.

**blocked**

This boolean property indicates whether the event will be delivered to event callbacks. If it is set to True, then no further callbacks will receive the event. When possible, it is recommended to use Event.handled rather than Event.blocked.

**handled**

This boolean property indicates whether the event has already been acted on by an event handler. Since many handlers may have access to the same events, it is recommended that each check whether the event has already been handled as well as set handled=True if it decides to act on the event.

- **source**
  
  The object that the event applies to (i.e. the source of the event).

- **sources**
  
  List of objects that the event applies to (i.e. are or have been a source of the event). Can contain multiple objects in case the event traverses a hierarchy of objects.

class `vispy.app.canvas.MouseEvent` *(type, pos=None, button=None, modifiers=None, delta=None, **kwds)*

Class describing mouse events.

Note that each event object has an attribute for each of the input
arguments listed below.

**type** [str] String indicating the event type (e.g. mouse_press, key_release)

**native** [object (optional)] The native GUI event object

**pos** [(int, int)] The position of the mouse (in screen coordinates).

**button** [int] The button that this event applies to (can be None). Left=1, right=2, middle=3.

**modifiers** [tuple of Key instances] Tuple that specifies which modifier keys were pressed down at the time of
the event (shift, control, alt, meta).

**delta** [(float, float)] The amount of scrolling in horizontal and vertical direction. One “tick” corresponds to a
delta of 1.0.

**kwds** [keyword arguments] All extra keyword arguments become attributes of the event object.

class vispy.app.canvas.KeyEvent (type, key=None, text='', modifiers=None, **kwds)

Class describing mouse events.

Note that each event object has an attribute for each of the input arguments listed below.

**type** [str] String indicating the event type (e.g. mouse_press, key_release)

**native** [object (optional)] The native GUI event object

**key** [vispy.keys.Key instance] The Key object for this event. Can be compared to string names.

**text** [str] The text representation of the key (can be an empty string).

**modifiers** [tuple of Key instances] Tuple that specifies which modifier keys were pressed down at the time of
the event (shift, control, alt, meta).

**kwds** [keyword arguments] All extra keyword arguments become attributes of the event object.

class vispy.app.canvas.ResizeEvent (type, size=None, **kwds)

Class describing canvas resize events.

Note that each event object has an attribute for each of the input arguments listed below.

**type** [str] String indicating the event type (e.g. mouse_press, key_release)

**native** [object (optional)] The native GUI event object

**size** [(int, int)] The new size of the Canvas.

**kwds** [extra keyword arguments] All extra keyword arguments become attributes of the event object.

class vispy.app.canvas.PaintEvent (type, region=None, **kwds)

Class describing canvas paint events. This type of event is sent to Canvas.events.paint when a repaint is required.

Note that each event object has an attribute for each of the input arguments listed below.

**type** [str] String indicating the event type (e.g. mouse_press, key_release)

**native** [object (optional)] The native GUI event object

**region** [(int, int, int, int) or None] The region of the canvas which needs to be repainted (x, y, w, h). If None,
the entire canvas must be repainted.

**kwds** [extra keyword arguments] All extra keyword arguments become attributes of the event object.
class vispy.event.EventEmitter (source=None, type=None, event_class=<class 'vispy.event.Event'>)

Encapsulates a list of event callbacks.

Each instance of EventEmitter represents the source of a stream of similar events, such as mouse click events or timer activation events. For example, the following diagram shows the propagation of a mouse click event to the list of callbacks that are registered to listen for that event:

User clicks | Canvas creates | Canvas invokes its |
Canvas | MouseEvent: | 'mouse_press' EventEmitter: |
| | | callbacks in sequence:
| --|event = MouseEvent(...) --|Canvas.events.mouse_press(event) --|callback1(event)
| | | --|callback2(event)
| | | --|callback3(event)

Callback functions may be added or removed from an EventEmitter using connect() or disconnect().

Calling an instance of EventEmitter will cause each of its callbacks to be invoked in sequence. All callbacks are invoked with a single argument which will be an instance of Event.

EventEmitters are generally created by an EmitterGroup instance.

source [object] The object that the generated events apply to. All emitted Events will have their .source property set to this value.

type [str or None] String indicating the event type (e.g. mouse_press, key_release)

event_class [subclass of Event] The class of events that this emitter will generate.

ignore_callback_errors [bool] If True, exceptions raised while invoking callbacks will be caught by the emitter, allowing it to continue invoking other callbacks.

print_callback_errors [bool] If True, the emitter prints a message and stack trace whenever a callback raises an exception. (assumes ignore_callback_errors=True)

__call__(**kwds)
Invoke all callbacks for this emitter.

Emit a new event object, created with the given keyword arguments, which must match with the input arguments of the corresponding event class. Note that the 'type' argument is filled in by the emitter.

Alternatively, the emitter can also be called with an Event instance as the only argument. In this case, the specified Event will be used rather than generating a new one. This allows customized Event instances to be emitted and also allows EventEmitters to be chained by connecting one directly to another.

Note that the same Event instance is sent to all callbacks. This allows some level of communication between the callbacks (notably, via Event.handled) but also requires that callbacks be careful not to inadvertently modify the Event.

block()
Block this emitter. Any attempts to emit an event while blocked will be silently ignored.

Calls to block are cumulative; the emitter must be unblocked the same number of times as it is blocked.

blocker()
Return an EventBlocker to be used in ‘with’ statements:

with emitter.blocker(): ..do stuff; no events will be emitted..

connect (callback)
Connect this emitter to a new callback.
callback may be either a callable object or a tuple (object, attr_name) where object.attr_name will point to a callable object.

If the callback is already connected, then the request is ignored.

The new callback will be added to the beginning of the callback list; thus the callback that is connected _last_ will be the _first_ to receive events from the emitter.

**disconnect** (callback=None)

Disconnect a callback from this emitter.

If no callback is specified, then _all_ callbacks are removed. If the callback was not already connected, then the call does nothing.

**source**

The object that events generated by this emitter apply to.

**unblock**()

Unblock this emitter. See event.EventEmitter.block().

---

### vispy.event.**EmitterGroup** (source=None, auto_connect=True, **emitters)

EmitterGroup instances manage a set of related EventEmitter. Its primary purpose is to provide organization for objects that make use of multiple emitters and to reduce the boilerplate code needed to initialize those emitters with default connections.

EmitterGroup instances are usually stored as an ‘events’ attribute on objects that use multiple emitters. For example:

```python
EmitterGroup  EventEmitter
    |                  |
Canvas.events.mouse_press
Canvas.events.resized
Canvas.events.key_press
```

EmitterGroup is also a subclass of EventEmitter, allowing it to emit its own events. Any callback that connects directly to the EmitterGroup will receive _all_ of the events generated by the group’s emitters.

**source** [object] The object that the generated events apply to.

**auto_connect** [bool] If auto_connect is True (default), then one connection will be made for each emitter that looks like emitter.connect((source, 'on_'+event_name)). This provides a simple mechanism for automatically connecting a large group of emitters to default callbacks.

**emitters** [keyword arguments] See the add method.

**add** (auto_connect=None, **kwds)

Add one or more EventEmitter instances to this emitter group. Each keyword argument may be specified as either an EventEmitter instance or an Event subclass, in which case an EventEmitter will be generated automatically. Thus:

```python
# This statement:
group.add(mouse_press=MouseEvent,
          mouse_release=MouseEvent)
```

```python
# ..is equivalent to this statement:
group.add(mouse_press=EventEmitter(group.source, 'mouse_press', MouseEvent),
          mouse_release=EventEmitter(group.source, 'mouse_press', MouseEvent))
```

**block_all**()

Block all emitters in this group.
connect (callback)
Connect the callback to the event group. The callback will receive events from _all_ of the emitters in the
group.
See EventEmitter.connect() for arguments.

disconnect (callback=None)
Disconnect the callback from this group. See connect() and EventEmitter.connect() for more
information.

emitters
List of current emitters in this group.

unblock_all ()
Unblock all emitters in this group.
The app module defines three classes: Application, Canvas, and Timer. On loading, vispy creates a default Application instance which can be used via functions in the module’s namespace.

```python
vispy.app.use(backend_name=None)
```

Select a backend by name. If the backend name is omitted, will chose a suitable backend automatically. It is an error to try to select a particular backend if one is already selected. Available backends: ‘PySide’, ‘PyQt4’, ‘Glut’, ‘Pyglet’, ‘qt’. The latter will use PySide or PyQt4, whichever works.

If a backend name is provided, and that backend could not be loaded, an error is raised.

If no backend name is provided, this function will first check if the GUI toolkit corresponding to each backend is already imported, and try that backend first. If this is unsuccessful, it will try the ‘default_backend’ provided in the vispy config. If still not successful, it will try each backend in a predetermined order.

```python
vispy.app.run()
```

Enter the native GUI event loop.

```python
vispy.app.quit(self)
```

Quit the native GUI event loop.

```python
vispy.app.process_events(self)
```

Process all pending GUI events. If the mainloop is not running, this should be done regularly to keep the visualization interactive and to keep the event system going.

```python
class vispy.app.Application
```

Representation of the vispy application. This wraps a native GUI application instance. Vispy has a default instance of this class at vispy.app.default_app.

**There are multiple stages for an Application object:**

- Backend-less - the state when it is just initialized
- Backend selected - use() has been successfully called. Note that the Canvas calls use() without arguments right before creating its backend widget.
- Native application is created - the Canvas probes the Application.native property to ensure that there is a native application right before a native widget is created.

```python
backend_module
```

The module object that defines the backend.

```python
backend_name
```

The name of the GUI backend that this app wraps.
The native GUI application instance.

Process all pending GUI events. If the mainloop is not running, this should be done regularly to keep the visualization interactive and to keep the event system going.

Quit the native GUI event loop.

Enter the native GUI event loop.

Select a backend by name. If the backend name is omitted, will chose a suitable backend automatically. It is an error to try to select a particular backend if one is already selected. Available backends: ‘PySide’, ‘PyQt4’, ‘Glut’, ‘Pyglet’, ‘qt’. The latter will use PySide or PyQt4, whichever works.

If a backend name is provided, and that backend could not be loaded, an error is raised.

If no backend name is provided, this function will first check if the GUI toolkit corresponding to each backend is already imported, and try that backend first. If this is unsuccessful, it will try the ‘default_backend’ provided in the vispy config. If still not succesful, it will try each backend in a predetermined order.

class vispy.app.Canvas(*args, **kwargs)

Representation of a GUI element that can be rendered to by an OpenGL context. The args and kwargs are used to instantiate the native widget.

Further, there are two special keyword arguments:

- app: an vispy Application instance (vispy.app is used by default)
- create_native: a bool that indicates whether to create the widget immediately (default True)

Receives the following events: initialize, resize, paint, mouse_press, mouse_release, mouse_move, mouse_wheel, key_press, key_release, stylus, touch, close

app
The vispy Application instance on which this Canvas is based.

close()
Close the canvas.

create_native()
Create the native widget if not already done so. If the widget is already created, this function does nothing.

geometry
Get or set the location and size of the Canvas in window coordinates (x, y, width, height). When setting, width and height may be omitted. Similarly, specifying None for x and y will prevent the widget from being moved.

native
The native widget object on which this Canvas is based.

show(visible=True)
Show (or hide) the canvas.

swap_buffers()
Swap GL buffers such that the offscreen buffer becomes visible.

title
The title of the canvas. If the canvas represents a window, the title is shown in its title bar.
**update()**
Inform the backend that the Canvas needs to be repainted.

---

**class vispy.app.Timer (interval=0.0, connect=None, iterations=-1, start=False, app=None)**
Timer used to schedule events in the future or on a repeating schedule.

- **app**
  The vispy Application instance on which this Timer is based.

- **connect (callback)**
  Alias for self.events.timeout.connect()

- **disconnect (callback=None)**
  Alias for self.events.timeout.disconnect()

- **native**
  The native timer on which this Timer is based.

- **start (interval=None, iterations=None)**
  Start the timer.
  A timeout event will be generated every `interval` seconds. If `interval` is None, then self.interval will be used.
  If `iterations` is specified, the timer will stop after emitting that number of events. If unspecified, then the previous value of self.iterations will be used. If the value is negative, then the timer will continue running until stop() is called.

- **stop ()**
  Stop the timer.
• examples/app-event
• examples/app-glut
• examples/app-pyglet
• examples/app-qt
• examples/boids
• examples/cloud-sdf
• examples/cloud
• examples/fake-galaxy
• examples/fireworks
• examples/game_of_life
• examples/hello_fbo
• examples/hello_quad1
• examples/hello_quad2
• examples/hello_sprite
• examples/points
• examples/simple
• examples/spinning-cube
• examples/texturing
• examples/transforms
CHAPTER
SEVEN

INDICES AND TABLES

- genindex
- modindex
- search
v
vispy.app
vispy.event
vispy.oogl