ModernGL Documentation

Release 5.6.2

Szabolcs Dombi

Sep 08, 2020

Contents

ModernGL is a high performance rendering module for Python.

CHAPTER 1

Install

1.1 From PyPI (pip)

ModernGL is available on PyPI for Windows, OS X and Linux as pre-built wheels. No complication is needed unless you are setting up a development environment.

```
$ pip install moderngl
```

Verify that the package is working:

```
$ python -m moderngl
moderngl 5.6.0
--------------
vendor: NVIDIA Corporation
renderer: GeForce RTX 2080 SUPER/PCIe/SSE2
version: 3.3.0 NVIDIA 441.87
python: 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019, 00:42:30) [MSC v.1916 64 bit␣
↪(AMD64)]
platform: win32
code: 330
```

Note: If you can only run in headless mode this might not work out of the box. You might need to set up `xvfb` and possibly supply more arguments during context creation. More info can be found in later sections.

1.2 Development environment

Ideally you want to fork the repository first.

```
# .. or clone for your fork
git clone https://github.com/moderngl/moderngl.git
cd moderngl
```

Building on various platforms:

- On Windows you need visual c++ build tools installed: https://visualstudio.microsoft.com/visual-cpp-build-tools/
- On OS X you need X Code installed + command line tools (`xcode-select --install`)
- Building on linux should pretty much work out of the box

CHAPTER 2

The Guide

2.1 A short introduction

What you will need?

To get something rendered, you will need a *VertexArray*.

VertexArrays can be created from a *Program* object and several *Buffer* objects.

To create a *Program* object, you will need some `Shader` objects.

Once you have your *Program* object, you can fill a *Buffer* with your data, then pass them to *VertexArray*, then call *VertexArray.render()*.

All of the objects above can only be created from a *Context* object.

Here is our checklist:

1. Install ModernGL.
2. Create a Context.
3. Create a Program object.
4. Create a VertexArray object.

Proceed to the *next step*.

2.2 Install ModernGL

```
$ pip install --upgrade ModernGL
```

This tutorial will also use `numpy` to generate data and `Pillow` to save the final image.

```
$ pip install --upgrade numpy Pillow
```

Proceed to the *next step*.

2.3 Context Creation

Note: From moderngl 5.6 context creation is handled by the glcontext package. This makes expanding context support easier for users lowering the bar for contributions. It also means context creation is no longer limited by a moderngl releases.

Note: This page might not list all supported backends as the glcontext project keeps evolving. If using anything outside of the default contexts provided per OS, please check the listed backends in the glcontext project.

2.3.1 Introduction

A context is an object giving moderngl access to opengl instructions (greatly simplified). How a context is created depends on your operating system and what kind of platform you want to target.

In the vast majority of cases you'll be using the default context backend supported by your operating system. This backend will be automatically selected unless a specific `backend` parameter is used.

Default backend per OS

- **Windows**: wgl / opengl32.dll
- **Linux**: x11/glx/libGL
- **OS X**: CGL

These default backends support two modes:

- Detecting an exiting active context possibly created by a window library such as glfw, sdl2, pyglet etc.
- Creating a headless context (No visible window)

Attaching to an existing active context created by a window library:

```
import moderngl
# .. do window initialization here
ctx = moderngl.create_context()
# If successful we can now render to the window
print("Default framebuffer is:", ctx.screen)
```

Creating a headless context:

```
import moderngl
# Create the context
ctx = moderngl.create_context(standalone=True)
# Create a framebuffer we can render to
fbo = ctx.simple_framebuffer((100, 100), 4)
fbo.use()
```

2.3.2 Require a minimum OpenGL version

ModernGL only support 3.3+ contexts. By default version 3.3 is passed in as the minimum required version of the context returned by the backend.

To require a specific version:

```
moderngl.create_context(require=430)
```

This will require OpenGL 4.3. If a lower context version is returned the context creation will fail.

This attribute can be accessed in *Context.version_code* and will be updated to contain the actual version code of the context (If higher than required).

2.3.3 Specifying context backend

A `backend` can be passed in for more advanced usage.

For example: Making a headless EGL context on linux:

```
ctx = moderngl.create_context(standalone=True, backend='egl')
```

Note: Each backend supports additional keyword arguments for more advanced configuration. This can for example be the exact name of the library to load. More information in the glcontext docs.

2.3.4 Context sharing

Warning: Object sharing is an experimental feature

Some context support the `share` parameters enabling object sharing between contexts. This is not needed if you are attaching to existing context with share mode enabled. For example if you create two windows with glfw enabling object sharing.

ModernGL objects (such as *moderngl.Buffer*, *moderngl.Texture*, ..) has a `ctx` property containing the context they were created in. Still **ModernGL do not check what context is currently active when accessing these objects.** This means the object can be used in both contexts when sharing is enabled.

This should in theory work fine with object sharing enabled:

```
data1 = numpy.array([1, 2, 3, 4], dtype='u1')
data2 = numpy.array([4, 3, 2, 1], dtype='u1')

ctx1 = moderngl.create_context(standalone=True)
ctx2 = moderngl.create_context(standalone=True, share=True)

with ctx1 as ctx:
    b1 = ctx.buffer(data1)

with ctx2 as ctx:
    b2 = ctx.buffer(data2)

print(b1.glo)  # Displays: 1
```

(continues on next page)

(continued from previous page)

```
print(b2.glo)  # Displays: 2

with ctx1:
    print(b1.read())
    print(b2.read())

with ctx2:
    print(b1.read())
    print(b2.read())
```

Still, there are some limitations to object sharing. Especially objects that reference other objects (framebuffer, vertex array object, etc.)

More information for a deeper dive:

- https://www.khronos.org/opengl/wiki/OpenGL_Object#Object_Sharing
- https://www.khronos.org/opengl/wiki/Memory_Model

2.4 Buffer Format

2.4.1 Description

A buffer format is a short string describing the layout of data in a vertex buffer object (VBO).

A VBO often contains a homogeneous array of C-like structures. The buffer format describes what each element of the array looks like. For example, a buffer containing an array of high-precision 2D vertex positions might have the format `"2f8"` - each element of the array consists of two floats, each float being 8 bytes wide, ie. a double.

Buffer formats are used in the *`Context.vertex_array()`* constructor, as the 2nd component of the *content* arg. See the *Example of simple usage* below.

2.4.2 Syntax

A buffer format looks like:

```
[count]type[size] [[count]type[size]...] [/usage]
```

Where:

- `count` is an optional integer. If omitted, it defaults to `1`.
- `type` is a single character indicating the data type:
  - `f` float
  - `i` int
  - `u` unsigned int
  - `x` padding
- `size` is an optional number of bytes used to store the type. If omitted, it defaults to 4 for numeric types, or to 1 for padding bytes.

  A format may contain multiple, space-separated `[count]type[size]` triples (See the *Example of single interleaved array*), followed by:

- `/usage` is optional. It should be preceded by a space, and then consists of a slash followed by a single character, indicating how successive values in the buffer should be passed to the shader:
  - `/v` per vertex. Successive values from the buffer are passed to each vertex. This is the default behavior if usage is omitted.
  - `/i` per instance. Successive values from the buffer are passed to each instance.
  - `/r` per render. the first buffer value is passed to every vertex of every instance. ie. behaves like a uniform.

  When passing multiple VBOs to a VAO, the first one must be of usage `/v`, as shown in the *Example of multiple arrays with differing /usage*.

Valid combinations of type and size are:

	size			
type	1	2	4	8
f	Unsigned byte (normalized)	Half float	Float	Double
i	Byte	Short	Int	-
u	Unsigned byte	Unsigned short	Unsigned int	-
x	1 byte	2 bytes	4 bytes	8 bytes

The entry `f1` has two unusual properties:

1. Its type is `f` (for float), but it defines a buffer containing unsigned bytes. For this size of floats only, the values are *normalized*, ie. unsigned bytes from 0 to 255 in the buffer are converted to float values from 0.0 to 1.0 by the time they reach the vertex shader. This is intended for passing in colors as unsigned bytes.
2. Three unsigned bytes, with a format of `3f1`, may be assigned to a `vec3` attribute, as one would expect. But, from ModernGL v6.0, they can alternatively be passed to a `vec4` attribute. This is intended for passing a buffer of 3-byte RGB values into an attribute which also contains an alpha channel.

There are no size 8 variants for types `i` and `u`.

This buffer format syntax is specific to ModernGL. As seen in the usage examples below, the formats sometimes look similar to the format strings passed to `struct.pack`, but that is a different syntax (documented here.)

Buffer formats can represent a wide range of vertex attribute formats. For rare cases of specialized attribute formats that are not expressible using buffer formats, there is a *`VertexArray.bind()`* method, to manually configure the underlying OpenGL binding calls. This is not generally recommended.

2.4.3 Examples

Example buffer formats

`"2f"` has a count of `2` and a type of `f` (float). Hence it describes two floats, passed to a vertex shader's `vec2` attribute. The size of the floats is unspecified, so defaults to `4` bytes. The usage of the buffer is unspecified, so defaults to `/v` (vertex), meaning each successive pair of floats in the array are passed to successive vertices during the render call.

`"3i2/i"` means three `i` (integers). The size of each integer is `2` bytes, ie. they are shorts, passed to an `ivec3` attribute. The trailing `/i` means that consecutive values in the buffer are passed to successive *instances* during an instanced render call. So the same value is passed to every vertex within a particular instance.

Buffers contining interleaved values are represented by multiple space separated count-type-size triples. Hence:

`"2f 3u x /v"` means:

- `2f`: two floats, passed to a `vec2` attribute, followed by
- `3u`: three unsigned bytes, passed to a `uvec3`, then

- `x`: a single byte of padding, for alignment.

The `/v` indicates successive elements in the buffer are passed to successive vertices during the render. This is the default, so the `/v` could be omitted.

Example of simple usage

Consider a VBO containing 2D vertex positions, forming a single triangle:

```
# a 2D triangle (ie. three (x, y) vertices)
verts = [
     0.0, 0.9,
    -0.5, 0.0,
     0.5, 0.0,
]

# pack all six values into a binary array of C-like floats
verts_buffer = struct.pack("6f", *verts)

# put the array into a VBO
vbo = ctx.buffer(verts_buffer)

# use the VBO in a VAO
vao = ctx.vertex_array(
    shader_program,
    [
        (vbo, "2f", "in_vert"), # <---- the "2f" is the buffer format
    ]
    index_buffer_object
)
```

The line `(vbo, "2f", "in_vert")`, known as the VAO content, indicates that `vbo` contains an array of values, each of which consists of two floats. These values are passed to an `in_vert` attribute, declared in the vertex shader as:

```
in vec2 in_vert;
```

The `"2f"` format omits a `size` component, so the floats default to 4-bytes each. The format also omits the trailing `/usage` component, which defaults to `/v`, so successive (x, y) rows from the buffer are passed to successive vertices during the render call.

Example of single interleaved array

A buffer array might contain elements consisting of multiple interleaved values.

For example, consider a buffer array, each element of which contains a 2D vertex position as floats, an RGB color as unsigned ints, and a single byte of padding for alignment:

position		color			padding
x	y	r	g	b	-
float	float	unsigned byte	unsigned byte	unsigned byte	byte

Such a buffer, however you choose to contruct it, would then be passed into a VAO using:

```
vao = ctx.vertex_array(
    shader_program,
    [
        (vbo, "2f 3f1 x", "in_vert", "in_color")
    ]
    index_buffer_object
)
```

The format starts with `2f`, for the two position floats, which will be passed to the shader's `in_vert` attribute, declared as:

```
in vec2 in_vert;
```

Next, after a space, is `3f1`, for the three color unsigned bytes, which get normalized to floats by `f1`. These floats will be passed to the shader's `in_color` attribute:

```
in vec3 in_color;
```

Finally, the format ends with `x`, a single byte of padding, which needs no shader attribute name.

Example of multiple arrays with differing `/usage`

To illustrate the trailing `/usage` portion, consider rendering a dozen cubes with instanced rendering. We will use:

- `vbo_verts_normals` contains vertices (3 floats) and normals (3 floats) for the vertices within a single cube.
- `vbo_offset_orientation` contains offsets (3 floats) and orientations (9 float matrices) that are used to position and orient each cube.
- `vbo_colors` contains colors (3 floats). In this example, there is only one color in the buffer, that will be used for every vertex of every cube.

Our shader will take all the above values as attributes.

We bind the above VBOs in a single VAO, to prepare for an instanced rendering call:

```
vao = ctx.vertex_array(
    shader_program,
    [
        (vbo_verts_normals,      "3f 3f /v", "in_vert", "in_norm"),
        (vbo_offset_orientation, "3f 9f /i", "in_offset", "in_orientation"),
        (vbo_colors,             "3f /r",    "in_color"),
    ]
    index_buffer_object
)
```

So, the vertices and normals, using `/v`, are passed to each vertex within an instance. This fulfills the rule tha the first VBO in a VAO must have usage `/v`. These are passed to vertex attributes as:

```
in vec3 in_vert;
in vec3 in_norm;
```

The offsets and orientations pass the same value to each vertex within an instance, but then pass the next value in the buffer to the vertices of the next instance. Passed as:

```
in vec3 in_offset;
in mat3 in_orientation;
```

The single color is passed to every vertex of every instance. If we had stored the color with `/v` or `/i`, then we would have had to store duplicate identical color values in vbo_colors - one per instance or one per vertex. To render all our cubes in a single color, this is needless duplication. Using `/r`, only one color is require the buffer, and it is passed to every vertex of every instance for the whole render call:

```
in vec3 in_color;
```

An alternative approach would be to pass in the color as a uniform, since it is constant. But doing it as an attribute is more flexible. It allows us to reuse the same shader program, bound to a different buffer, to pass in color data which varies per instance, or per vertex.

2.5 Program

ModernGL is different from standard plotting libraries. You can define your own shader program to render stuff. This could complicate things, but also provides freedom on how you render your data.

Here is a sample program that passes the input vertex coordinates as is to screen coordinates.

Screen coordinates are in the [-1, 1], [-1, 1] range for x and y axes. The (-1, 1) point is the lower left corner of the screen.

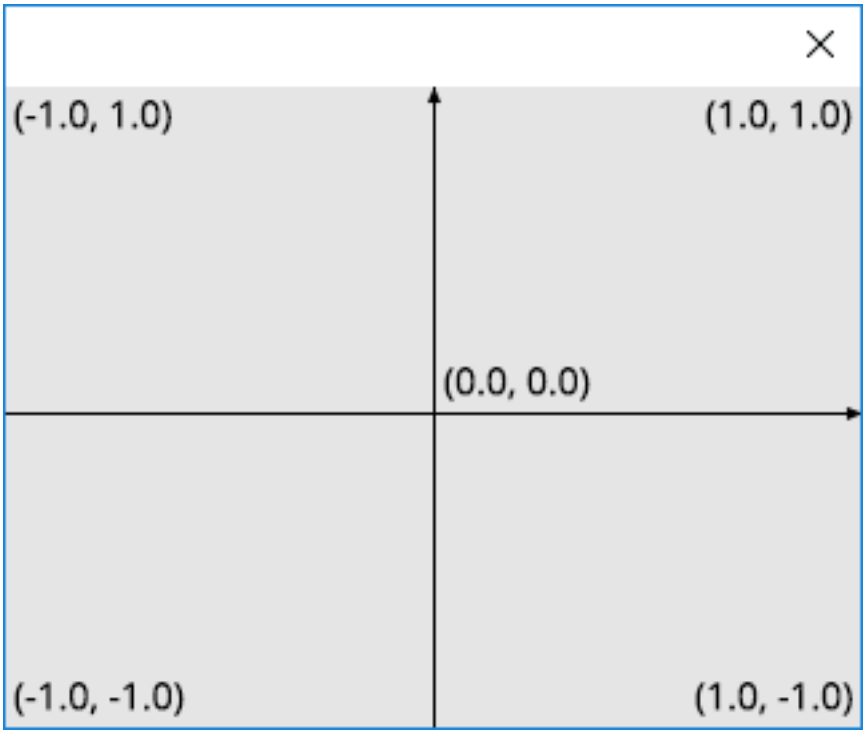


Fig. 1: The screen coordinates

The program will also process a color information.

Entire source

```python
import moderngl

ctx = moderngl.create_standalone_context()

prog = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

        void main() {
            v_color = in_color;
            gl_Position = vec4(in_vert, 0.0, 1.0);
        }
    ''',
    fragment_shader='''
        #version 330

        in vec3 v_color;

        out vec3 f_color;

        void main() {
            f_color = v_color;
        }
    ''',
)
```

Vertex Shader

```
in vec2 in_vert;
in vec3 in_color;

out vec3 v_color;

void main() {
    v_color = in_color;
    gl_Position = vec4(in_vert, 0.0, 1.0);
}
```

Fragment Shader

```
in vec3 v_color;

out vec3 f_color;

void main() {
    f_color = v_color;
}
```

Proceed to the *next step*.

2.6 VertexArray

```
import moderngl
import numpy as np

ctx = moderngl.create_standalone_context()

prog = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

        void main() {
            v_color = in_color;
            gl_Position = vec4(in_vert, 0.0, 1.0);
        }
    ''',
    fragment_shader='''
        #version 330

        in vec3 v_color;

        out vec3 f_color;

        void main() {
            f_color = v_color;
        }
    ''',
)

x = np.linspace(-1.0, 1.0, 50)
y = np.random.rand(50) - 0.5
r = np.ones(50)
g = np.zeros(50)
b = np.zeros(50)

vertices = np.dstack([x, y, r, g, b])

vbo = ctx.buffer(vertices.astype('f4').tobytes())
vao = ctx.simple_vertex_array(prog, vbo, 'in_vert', 'in_color')
```

Proceed to the *next step*.

2.7 Rendering

```
import moderngl
import numpy as np

from PIL import Image

```

(continues on next page)

(continued from previous page)

```
ctx = moderngl.create_standalone_context()

prog = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

        void main() {
            v_color = in_color;
            gl_Position = vec4(in_vert, 0.0, 1.0);
        }
    ''',
    fragment_shader='''
        #version 330

        in vec3 v_color;

        out vec3 f_color;

        void main() {
            f_color = v_color;
        }
    ''',
)

x = np.linspace(-1.0, 1.0, 50)
y = np.random.rand(50) - 0.5
r = np.ones(50)
g = np.zeros(50)
b = np.zeros(50)

vertices = np.dstack([x, y, r, g, b])

vbo = ctx.buffer(vertices.astype('f4').tobytes())
vao = ctx.simple_vertex_array(prog, vbo, 'in_vert', 'in_color')

fbo = ctx.simple_framebuffer((512, 512))
fbo.use()
fbo.clear(0.0, 0.0, 0.0, 1.0)
vao.render(moderngl.LINE_STRIP)

Image.frombytes('RGB', fbo.size, fbo.read(), 'raw', 'RGB', 0, -1).show()
```

2.8 Headless on Ubuntu 18 Server

2.8.1 Dependencies

Headless rendering can be achieved with EGL or X11. We'll cover both cases.

Starting with fresh ubuntu 18 server install we need to install required packages:

```
sudo apt-install python3-pip mesa-utils libegl1-mesa xvfb
```

This should install mesa an diagnostic tools if needed later.

- `mesa-utils` installs libgl1-mesa and tools like `glxinfo`
- `libegl1-mesa` is optional if using EGL instead of X11

2.8.2 Creating a context

The libraries we are going to interact with has the following locations:

```
/usr/lib/x86_64-linux-gnu/libGL.so.1
/usr/lib/x86_64-linux-gnu/libX11.so.6
/usr/lib/x86_64-linux-gnu/libEGL.so.1
```

Double check what library versions you actually have installed and make modifications to what versions we refer to below. moderngl will attempt to load `libGL.so`, `libX11.so` and `libEGL.so` by default. Optionally you can create symlinks or use python to locate the desired lib files. For simplicity we will be using the exact library names.

Before we can create a context we to run a virtual display:

```
export DISPLAY=:99.0
Xvfb :99 -screen 0 640x480x24 &
```

Now we can create a context with x11 or egl:

```
# X11
import moderngl
ctx = moderngl.create_context(
    standalone=True,
    libgl='libGL.so.1',
    libx11='libX11.so.6',
)

# EGL
import moderngl
ctx = moderngl.create_context(
    standalone=True,
    backend='egl',
    libgl='libGL.so.1',
    libegl='libEGL.so.1',
)
```

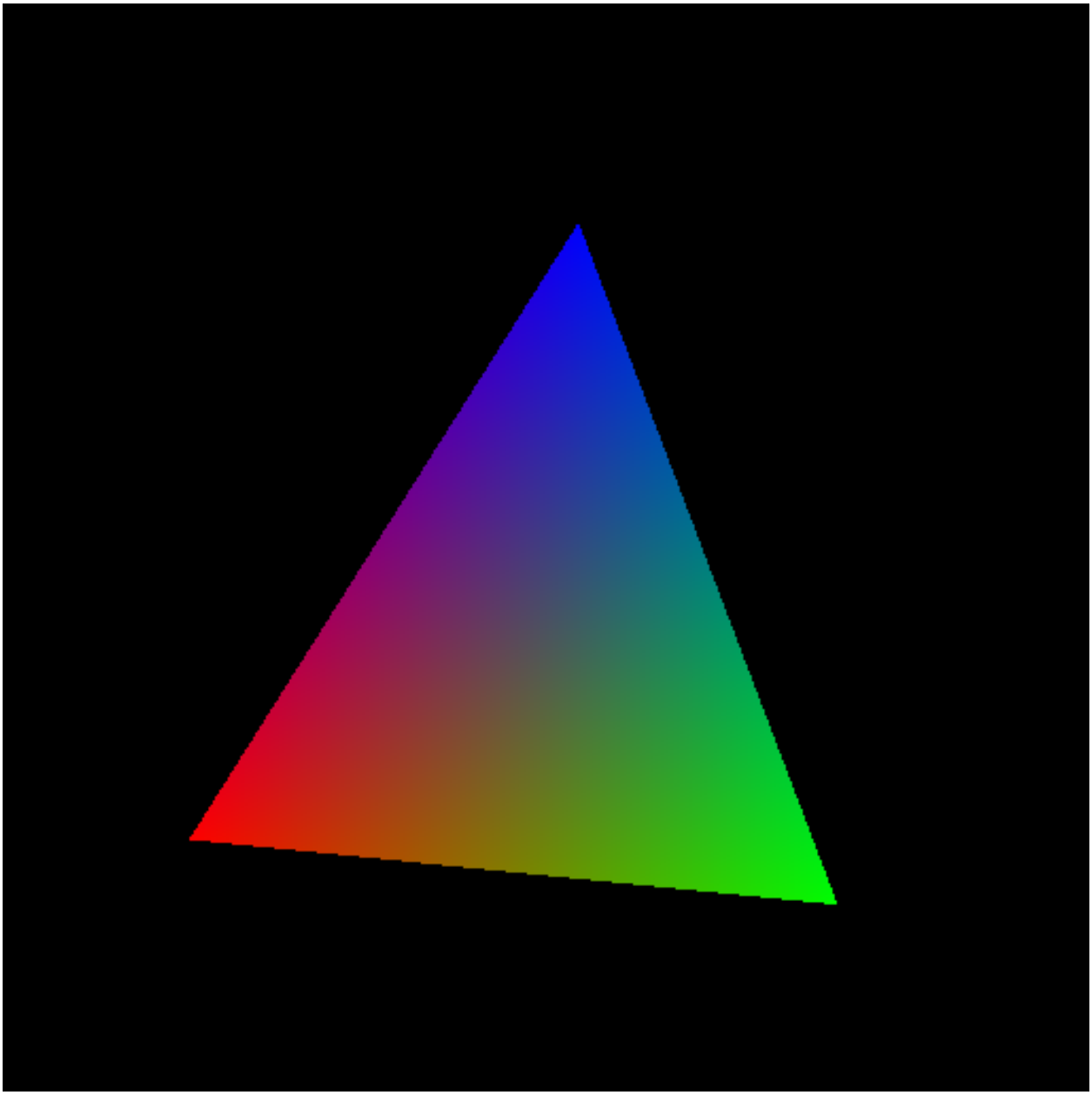
2.8.3 Running an example

Checking that everything works can be done with a basic triangle example.

Install dependencies:

```
pip3 install moderngl numpy pyrr pillow
```

The following example renders a triangle and writes it to a png file so we can verify the contents.



```
import moderngl
import numpy as np
from PIL import Image
from pyrr import Matrix44

# -------------------
# CREATE CONTEXT HERE
# -------------------

prog = ctx.program(vertex_shader="""
    #version 330
    uniform mat4 model;
    in vec2 in_vert;
    in vec3 in_color;
```

(continues on next page)

(continued from previous page)

```
    out vec3 color;
    void main() {
        gl_Position = model * vec4(in_vert, 0.0, 1.0);
        color = in_color;
    }
    """,
    fragment_shader="""
    #version 330
    in vec3 color;
    out vec4 fragColor;
    void main() {
        fragColor = vec4(color, 1.0);
    }
""")

vertices = np.array([
    -0.6, -0.6,
    1.0, 0.0, 0.0,
    0.6, -0.6,
    0.0, 1.0, 0.0,
    0.0, 0.6,
    0.0, 0.0, 1.0,
], dtype='f4')

vbo = ctx.buffer(vertices)
vao = ctx.simple_vertex_array(prog, vbo, 'in_vert', 'in_color')
fbo = ctx.framebuffer(color_attachments=[ctx.texture((512, 512), 4)])

fbo.use()
ctx.clear()
prog['model'].write(Matrix44.from_eulers((0.0, 0.1, 0.0), dtype='f4'))
vao.render(moderngl.TRIANGLES)

data = fbo.read(components=3)
image = Image.frombytes('RGB', fbo.size, data)
image = image.transpose(Image.FLIP_TOP_BOTTOM)
image.save('output.png')
```

CHAPTER 3

Reference

3.1 Context

class moderngl.**Context**
Class exposing OpenGL features. ModernGL objects can be created from this class.

3.1.1 Create

moderngl.**create_context**(*require=None*) → Context
Create a ModernGL context by loading OpenGL functions from an existing OpenGL context. An OpenGL context must exists.

Example:

```
# Accept the current context version
ctx = moderngl.create_context()

# Require at least OpenGL 4.3
ctx = moderngl.create_context(require=430)

# Create a headless context requiring OpenGL 4.3
ctx = moderngl.create_context(require=430, standalone=True)
```

Keyword Arguments

- **require** (*int*) – OpenGL version code (default: 330)
- **standalone** (*bool*) – Headless flag
- ****settings** – Other backend specific settings

Returns *Context* object

moderngl.**create_standalone_context**(*require=None*) → Context
Create a standalone ModernGL context. The preferred way to make a context ‘‘

Example:

```
# Create a context with highest possible supported version
ctx = moderngl.create_context()

# Require at least OpenGL 4.3
ctx = moderngl.create_context(require=430)
```

Keyword Arguments **require** (*int*) – OpenGL version code.

Returns *Context* object

3.1.2 ModernGL Objects

Context.**program**(*vertex_shader*, *fragment_shader=None*, *geometry_shader=None*, *tess_control_shader=None*, *tess_evaluation_shader=None*, *varyings=()*) → Program
Create a *Program* object.

Only linked programs will be returned.

A single shader in the *shaders* parameter is also accepted. The varyings are only used when a transform program is created.

Parameters

- **shaders** (*list*) – A list of Shader objects.
- **varyings** (*list*) – A list of varying names.

Returns *Program* object

Context.**simple_vertex_array**(*program*, *buffer*, **attributes*, *index_buffer=None*, *index_element_size=4*) → VertexArray
Create a *VertexArray* object.

Warning: This method is deprecated and may be removed in the future. Use *Context.vertex_array()* instead. It also supports the argument format this method describes.

Parameters

- **program** (Program) – The program used when rendering.
- **buffer** (Buffer) – The buffer.
- **attributes** (*list*) – A list of attribute names.

Keyword Arguments

- **index_element_size** (*int*) – byte size of each index element, 1, 2 or 4.
- **index_buffer** (Buffer) – An index buffer.

Returns *VertexArray* object

Context.**vertex_array**(**args*, ***kwargs*) → VertexArray
Create a *VertexArray* object.

This method also supports arguments for *Context.simple_vertex_array()*.

Parameters

- **program** (*Program*) – The program used when rendering.
- **content** (*list*) – A list of (buffer, format, attributes). See *Buffer Format*.
- **index_buffer** (*Buffer*) – An index buffer.

Keyword Arguments

- **index_element_size** (*int*) – byte size of each index element, 1, 2 or 4.
- **skip_errors** (*bool*) – Ignore skip_errors varyings.

Returns *VertexArray* object

Context.**buffer**(*data=None*, *reserve=0*, *dynamic=False*) → Buffer
Create a *Buffer* object.

Parameters **data** (*bytes*) – Content of the new buffer.

Keyword Arguments

- **reserve** (*int*) – The number of bytes to reserve.
- **dynamic** (*bool*) – Treat buffer as dynamic.

Returns *Buffer* object

Context.**texture**(*size*, *components*, *data=None*, *samples=0*, *alignment=1*, *dtype='f1'*) → Texture
Create a *Texture* object.

Parameters

- **size** (*tuple*) – The width and height of the texture.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **data** (*bytes*) – Content of the texture.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.

Returns *Texture* object

Context.**depth_texture**(*size*, *data=None*, *samples=0*, *alignment=4*) → Texture
Create a *Texture* object.

Parameters

- **size** (*tuple*) – The width and height of the texture.
- **data** (*bytes*) – Content of the texture.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.

Returns *Texture* object

Context.**texture3d**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → Texture3D
Create a *Texture3D* object.

Parameters

- **size** (*tuple*) – The width, height and depth of the texture.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **data** (*bytes*) – Content of the texture.

Keyword Arguments

- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.

Returns *Texture3D* object

Context.**texture_array**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → TextureArray
Create a *TextureArray* object.

Parameters

- **size** (*tuple*) – The `(width, height, layers)` of the texture.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **data** (*bytes*) – Content of the texture. The size must be `(width, height * layers)` so each layer is stacked vertically.

Keyword Arguments

- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.

Returns *Texture3D* object

Context.**texture_cube**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → TextureCube
Create a *TextureCube* object.

Parameters

- **size** (*tuple*) – The width, height of the texture. Each side of the cube will have this size.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **data** (*bytes*) – Content of the texture. The data should be have the following ordering: positive_x, negative_x, positive_y, negative_y, positive_z + negative_z

Keyword Arguments

- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.

Returns *TextureCube* object

Context.**simple_framebuffer**(*size*, *components=4*, *samples=0*, *dtype='f1'*) → Framebuffer
Creates a *Framebuffer* with a single color attachment and depth buffer using *moderngl.Renderbuffer* attachments.

Parameters

- **size** (*tuple*) – The width and height of the renderbuffer.

- **components** (*int*) – The number of components 1, 2, 3 or 4.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

Returns *Framebuffer* object

Context.**framebuffer**(*color_attachments=()*, *depth_attachment=None*) → Framebuffer
A *Framebuffer* is a collection of buffers that can be used as the destination for rendering. The buffers for Framebuffer objects reference images from either Textures or Renderbuffers.

Parameters

- **color_attachments** (*list*) – A list of *Texture* or *Renderbuffer* objects.
- **depth_attachment** (*Renderbuffer or Texture*) – The depth attachment.

Returns *Framebuffer* object

Context.**renderbuffer**(*size*, *components=4*, *samples=0*, *dtype='f1'*) → Renderbuffer
Renderbuffer objects are OpenGL objects that contain images. They are created and used specifically with *Framebuffer* objects.

Parameters

- **size** (*tuple*) – The width and height of the renderbuffer.
- **components** (*int*) – The number of components 1, 2, 3 or 4.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

Returns *Renderbuffer* object

Context.**depth_renderbuffer**(*size*, *samples=0*) → Renderbuffer
Renderbuffer objects are OpenGL objects that contain images. They are created and used specifically with *Framebuffer* objects.

Parameters **size** (*tuple*) – The width and height of the renderbuffer.

Keyword Arguments **samples** (*int*) – The number of samples. Value 0 means no multisample format.

Returns *Renderbuffer* object

Context.**scope**(*framebuffer=None*, *enable_only=None*, *textures=()*, *uniform_buffers=()*, *storage_buffers=()*, *samplers=()*, *enable=None*) → Scope
Create a *Scope* object.

Parameters

- **framebuffer** (*Framebuffer*) – The framebuffer to use when entering.
- **enable_only** (*int*) – The enable_only flags to set when entering.

Keyword Arguments

- **textures** (*list*) – List of (texture, binding) tuples.
- **uniform_buffers** (*list*) – List of (buffer, binding) tuples.
- **storage_buffers** (*list*) – List of (buffer, binding) tuples.

- **samplers** (*list*) – List of sampler bindings
- **enable** (*int*) – Flags to enable for this vao such as depth testing and blending

Context.**query**(*samples=False*, *any_samples=False*, *time=False*, *primitives=False*) → Query
Create a *Query* object.

Keyword Arguments

- **samples** (*bool*) – Query GL_SAMPLES_PASSED or not.
- **any_samples** (*bool*) – Query GL_ANY_SAMPLES_PASSED or not.
- **time** (*bool*) – Query GL_TIME_ELAPSED or not.
- **primitives** (*bool*) – Query GL_PRIMITIVES_GENERATED or not.

Context.**compute_shader**(*source*) → ComputeShader
A *ComputeShader* is a Shader Stage that is used entirely for computing arbitrary information. While it can do rendering, it is generally used for tasks not directly related to drawing.

Parameters **source** (*str*) – The source of the compute shader.

Returns *ComputeShader* object

Context.**sampler**(*repeat_x=True*, *repeat_y=True*, *repeat_z=True*, *filter=None*, *anisotropy=1.0*, *compare_func='?'*, *border_color=None*, *min_lod=-1000.0*, *max_lod=1000.0*, *texture=None*) → Sampler
Create a *Sampler* object.

Keyword Arguments

- **repeat_x** (*bool*) – Repeat texture on x
- **repeat_y** (*bool*) – Repeat texture on y
- **repeat_z** (*bool*) – Repeat texture on z
- **filter** (*tuple*) – The min and max filter
- **anisotropy** (*float*) – Number of samples for anisotropic filtering. Any value greater than 1.0 counts as a use of anisotropic filtering
- **compare_func** – Compare function for depth textures
- **border_color** (*tuple*) – The (r, g, b, a) color for the texture border. When this value is set the repeat_ values are overridden setting the texture wrap to return the border color when outside [0, 1] range.
- **min_lod** (*float*) – Minimum level-of-detail parameter (Default -1000.0). This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level)
- **max_lod** (*float*) – Minimum level-of-detail parameter (Default 1000.0). This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level)
- **texture** (Texture) – The texture for this sampler

Context.**clear_samplers**(*start=0*, *end=-1*)
Unbinds samplers from texture units. Sampler bindings do clear automatically between every frame, but lingering samplers can still be a source of weird bugs during the frame rendering. This methods provides a fairly brute force and efficient way to ensure texture units are clear.

Keyword Arguments

- **start** (*int*) – The texture unit index to start the clearing samplers
- **stop** (*int*) – The texture unit index to stop clearing samplers

Example:

```
# Clear texture unit 0, 1, 2, 3, 4
ctx.clear_samplers(start=0, end=5)

# Clear texture unit 4, 5, 6, 7
ctx.clear_samplers(start=4, end=8)
```

Context.**release**()
Release the ModernGL context.

If the context is not standalone the standard backends in `glcontext` will not do anything because the context was not created by moderngl.

Standalone contexts can normally be released.

3.1.3 Methods

Context.**clear**(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*, *viewport=None*, *color=None*)
Clear the bound framebuffer.

If a *viewport* passed in, a scissor test will be used to clear the given viewport. This viewport take prescense over the framebuffers *scissor*. Clearing can still be done with scissor if no viewport is passed in.

This method also respects the *color_mask* and *depth_mask*. It can for example be used to only clear the depth or color buffer or specific components in the color buffer.

If the *viewport* is a 2-tuple it will clear the `(0, 0, width, height)` where `(width, height)` is the 2-tuple.

If the *viewport* is a 4-tuple it will clear the given viewport.

Parameters

- **red** (*float*) – color component.
- **green** (*float*) – color component.
- **blue** (*float*) – color component.
- **alpha** (*float*) – alpha component.
- **depth** (*float*) – depth value.

Keyword Arguments **viewport** (*tuple*) – The viewport.

Context.**enable_only**(*flags*)
Clears all existing flags applying new ones.

Note that the enum values defined in moderngl are not the same as the ones in opengl. These are defined as bit flags so we can logical *or* them together.

Available flags:

- `moderngl.NOTHING`
- `moderngl.BLEND`
- `moderngl.DEPTH_TEST`
- `moderngl.CULL_FACE`
- `moderngl.RASTERIZER_DISCARD`
- `moderngl.PROGRAM_POINT_SIZE`

Examples:

```
# Disable all flags
ctx.enable_only(moderngl.NOTHING)

# Ensure only depth testing and face culling is enabled
ctx.enable_only(moderngl.DEPTH_TEST | moderngl.CULL_FACE)
```

Parameters **flags** (*EnableFlag*) – The flags to enable

Context.**enable**(*flags*)

Enable flags.

Note that the enum values defined in moderngl are not the same as the ones in opengl. These are defined as bit flags so we can logical *or* them together.

For valid flags, please see *enable_only()*.

Examples:

```
# Enable a single flag
ctx.enable(moderngl.DEPTH_TEST)

# Enable multiple flags
ctx.enable(moderngl.DEPTH_TEST | moderngl.CULL_FACE | moderngl.BLEND)
```

Parameters **flag** (*int*) – The flags to enable.

Context.**disable**(*flags*)

Disable flags.

For valid flags, please see *enable_only()*.

Examples:

```
# Only disable depth testing
ctx.disable(moderngl.DEPTH_TEST)

# Disable depth testing and face culling
ctx.disable(moderngl.DEPTH_TEST | moderngl.CULL_FACE)
```

Parameters **flag** (*int*) – The flags to disable.

Context.**finish**()

Wait for all drawing commands to finish.

Context.**copy_buffer**(*dst*, *src*, *size=-1*, *read_offset=0*, *write_offset=0*)

Copy buffer content.

Parameters

- **dst** (Buffer) – The destination buffer.
- **src** (Buffer) – The source buffer.
- **size** (*int*) – The number of bytes to copy.

Keyword Arguments

- **read_offset** (*int*) – The read offset.

- **write_offset** (*int*) – The write offset.

Context.**copy_framebuffer**(*dst*, *src*)
Copy framebuffer content.

Use this method to:

- blit framebuffers.
- copy framebuffer content into a texture.
- downsample framebuffers. (it will allow to read the framebuffer's content)
- downsample a framebuffer directly to a texture.

Parameters

- **dst** (*Framebuffer or Texture*) – Destination framebuffer or texture.
- **src** (*Framebuffer*) – Source framebuffer.

Context.**detect_framebuffer**(*glo=None*) → Framebuffer
Detect framebuffer.

Parameters **glo** (*int*) – Framebuffer object.

Returns *Framebuffer* object

Context.**__enter__**()
Enters the context.

This should ideally be used with the `with` statement:

```
with other_context as ctx:
    # Do something in this context
```

When exiting the context the previously bound context is activated again.

Warning: Context switching can be risky unless you know what you are doing. ModernGL objects are not aware of what context is currently active. Use with care.

Context.**__exit__**(*exc_type*, *exc_val*, *exc_tb*)
Exit the context.

See *Context.__enter__()*

3.1.4 Attributes

Context.**line_width**
Set the default line width.

Type float

Context.**point_size**
Set/get the default point size.

Type float

Context.**depth_func**
Set the default depth func. The depth function is set using a string.

Example:

```
ctx.depth_func = '<='  # GL_LEQUAL
ctx.depth_func = '<'   # GL_LESS
ctx.depth_func = '>='  # GL_GEQUAL
ctx.depth_func = '>'   # GL_GREATER
ctx.depth_func = '=='  # GL_EQUAL
ctx.depth_func = '!='  # GL_NOTEQUAL
ctx.depth_func = '0'   # GL_NEVER
ctx.depth_func = '1'   # GL_ALWAYS
```

Type int

Context.**blend_func**
Set the blend func (write only) Blend func can be set for rgb and alpha separately if needed.

Supported blend functions are:

```
moderngl.ZERO
moderngl.ONE
moderngl.SRC_COLOR
moderngl.ONE_MINUS_SRC_COLOR
moderngl.DST_COLOR
moderngl.ONE_MINUS_DST_COLOR
moderngl.SRC_ALPHA
moderngl.ONE_MINUS_SRC_ALPHA
moderngl.DST_ALPHA
moderngl.ONE_MINUS_DST_ALPHA
```

Example:

```
# For both rgb and alpha
ctx.blend_func = moderngl.SRC_ALPHA, moderngl.ONE_MINUS_SRC_ALPHA

# Separate for rgb and alpha
ctx.blend_func = (
    moderngl.SRC_ALPHA, moderngl.ONE_MINUS_SRC_ALPHA,
    moderngl.ONE, moderngl.ONE
)
```

Type tuple

Context.**blend_equation**
Set the blend equation (write only).

Blend equations specify how source and destination colors are combined in blending operations. By default FUNC_ADD is used.

Blend equation can be set for rgb and alpha separately if needed.

Supported functions are:

```
moderngl.FUNC_ADD               # source + destination
moderngl.FUNC_SUBTRACT          # source - destination
moderngl.FUNC_REVERSE_SUBTRACT  # destination - source
```

(continues on next page)

(continued from previous page)

```
moderngl.MIN                    # Minimum of source and destination
moderngl.MAX                    # Maximum of source and destination
```

Example:

```
# For both rgb and alpha channel
ctx.blend_func = moderngl.FUNC_ADD

# Separate for rgb and alpha channel
ctx.blend_func = moderngl.FUNC_ADD, moderngl.MAX
```

Type tuple

`Context.`**`viewport`**

Get or set the viewport of the active framebuffer.

Example:

```
>>> ctx.viewport
(0, 0, 1280, 720)
>>> ctx.viewport = (0, 0, 640, 360)
>>> ctx.viewport
(0, 0, 640, 360)
```

If no framebuffer is bound `(0, 0, 0, 0)` will be returned.

Type tuple

`Context.`**`scissor`**

Get or set the scissor box for the active framebuffer

When scissor testing is enabled fragments outside the defined scissor box will be discarded. This applies to rendered geometry or *`Context.clear()`*.

Setting is value enables scissor testing in the framebuffer. Setting the scissor to `None` disables scissor testing and reverts the scissor box to match the framebuffer size.

Example:

```
# Enable scissor testing
>>> ctx.scissor = 100, 100, 200, 100
# Disable scissor testing
>>> ctx.scissor = None
```

If no framebuffer is bound `(0, 0, 0, 0)` will be returned.

Type tuple

`Context.`**`version_code`**

The OpenGL version code. Reports `410` for OpenGL 4.1

Type int

`Context.`**`screen`**

A Framebuffer instance representing the screen usually set when creating a context with `create_context()` attaching to an existing context. This is the special system framebuffer represented by framebuffer `id=0`.

When creating a standalone context this property is not set.

Type *Framebuffer*

Context.**fbo**
The active framebuffer. Set every time *Framebuffer.use()* is called.

Type *Framebuffer*

Context.**front_face**
The front_face. Acceptable values are `'ccw'` (default) or `'cw'`.

Face culling must be enabled for this to have any effect: `ctx.enable(moderngl.CULL_FACE)`.

Example:

```
# Triangles winded counter-clockwise considered front facing
ctx.front_face = 'ccw'
# Triangles winded clockwise considered front facing
ctx.front_face = 'cw'
```

Type str

Context.**cull_face**
The face side to cull. Acceptable values are `'back'` (default) `'front'` or `'front_and_back'`.

This is similar to *Context.front_face()*

Face culling must be enabled for this to have any effect: `ctx.enable(moderngl.CULL_FACE)`.

Example:

```
#
ctx.cull_face = 'front'
#
ctx.cull_face = 'back'
#
ctx.cull_face = 'front_and_back'
```

Type str

Context.**wireframe**
Wireframe settings for debugging.

Type bool

Context.**max_samples**
The maximum supported number of samples for multisampling

Type int

Context.**max_integer_samples**
The max integer samples.

Type int

Context.**max_texture_units**
The max texture units.

Type int

Context.**default_texture_unit**
The default texture unit.

Type int

`Context.max_anisotropy`
The maximum value supported for anisotropic filtering.

Type float

`Context.multisample`
Enable/disable multisample mode (`GL_MULTISAMPLE`). This property is write only.

Example:

```
# Enable
ctx.multisample = True
# Disable
ctx.multisample = False
```

Type bool

`Context.patch_vertices`
The number of vertices that will be used to make up a single patch primitive.

Type int

`Context.provoking_vertex`
Specifies the vertex to be used as the source of data for flat shaded varyings.

Flatshading a vertex shader varying output (ie. `flat out vec3 pos`) means to assign all vetices of the primitive the same value for that output. The vertex from which these values is derived is known as the provoking vertex.

It can be configured to be the first or the last vertex.

This property is write only.

Example:

```
# Use first vertex
ctx.provoking_vertex = moderngl.FIRST_VERTEX_CONVENTION

# Use last vertex
ctx.provoking_vertex = moderngl.LAST_VERTEX_CONVENTION
```

Type int

`Context.error`
The result of `glGetError()` but human readable. This values is provided for debug purposes only and is likely to reduce performace when used in a draw loop.

Type str

`Context.info`
Information about the context

Example:

```
{
    'GL_VENDOR': 'NVIDIA Corporation',
    'GL_RENDERER': 'NVIDIA GeForce GT 650M OpenGL Engine',
    'GL_VERSION': '4.1 NVIDIA-10.32.0 355.11.10.10.40.102',
    'GL_POINT_SIZE_RANGE': (1.0, 2047.0),
    'GL_SMOOTH_LINE_WIDTH_RANGE': (0.5, 1.0),
```

(continues on next page)

(continued from previous page)

```
    'GL_ALIASED_LINE_WIDTH_RANGE': (1.0, 1.0),
    'GL_POINT_FADE_THRESHOLD_SIZE': 1.0,
    'GL_POINT_SIZE_GRANULARITY': 0.125,
    'GL_SMOOTH_LINE_WIDTH_GRANULARITY': 0.125,
    'GL_MIN_PROGRAM_TEXEL_OFFSET': -8.0,
    'GL_MAX_PROGRAM_TEXEL_OFFSET': 7.0,
    'GL_MINOR_VERSION': 1,
    'GL_MAJOR_VERSION': 4,
    'GL_SAMPLE_BUFFERS': 0,
    'GL_SUBPIXEL_BITS': 8,
    'GL_CONTEXT_PROFILE_MASK': 1,
    'GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT': 256,
    'GL_DOUBLEBUFFER': False,
    'GL_STEREO': False,
    'GL_MAX_VIEWPORT_DIMS': (16384, 16384),
    'GL_MAX_3D_TEXTURE_SIZE': 2048,
    'GL_MAX_ARRAY_TEXTURE_LAYERS': 2048,
    'GL_MAX_CLIP_DISTANCES': 8,
    'GL_MAX_COLOR_ATTACHMENTS': 8,
    'GL_MAX_COLOR_TEXTURE_SAMPLES': 8,
    'GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS': 233472,
    'GL_MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS': 231424,
    'GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS': 80,
    'GL_MAX_COMBINED_UNIFORM_BLOCKS': 70,
    'GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS': 233472,
    'GL_MAX_CUBE_MAP_TEXTURE_SIZE': 16384,
    'GL_MAX_DEPTH_TEXTURE_SAMPLES': 8,
    'GL_MAX_DRAW_BUFFERS': 8,
    'GL_MAX_DUAL_SOURCE_DRAW_BUFFERS': 1,
    'GL_MAX_ELEMENTS_INDICES': 150000,
    'GL_MAX_ELEMENTS_VERTICES': 1048575,
    'GL_MAX_FRAGMENT_INPUT_COMPONENTS': 128,
    'GL_MAX_FRAGMENT_UNIFORM_COMPONENTS': 4096,
    'GL_MAX_FRAGMENT_UNIFORM_VECTORS': 1024,
    'GL_MAX_FRAGMENT_UNIFORM_BLOCKS': 14,
    'GL_MAX_GEOMETRY_INPUT_COMPONENTS': 128,
    'GL_MAX_GEOMETRY_OUTPUT_COMPONENTS': 128,
    'GL_MAX_GEOMETRY_TEXTURE_IMAGE_UNITS': 16,
    'GL_MAX_GEOMETRY_UNIFORM_BLOCKS': 14,
    'GL_MAX_GEOMETRY_UNIFORM_COMPONENTS': 2048,
    'GL_MAX_INTEGER_SAMPLES': 1,
    'GL_MAX_SAMPLES': 8,
    'GL_MAX_RECTANGLE_TEXTURE_SIZE': 16384,
    'GL_MAX_RENDERBUFFER_SIZE': 16384,
    'GL_MAX_SAMPLE_MASK_WORDS': 1,
    'GL_MAX_SERVER_WAIT_TIMEOUT': -1,
    'GL_MAX_TEXTURE_BUFFER_SIZE': 134217728,
    'GL_MAX_TEXTURE_IMAGE_UNITS': 16,
    'GL_MAX_TEXTURE_LOD_BIAS': 15,
    'GL_MAX_TEXTURE_SIZE': 16384,
    'GL_MAX_UNIFORM_BUFFER_BINDINGS': 70,
    'GL_MAX_UNIFORM_BLOCK_SIZE': 65536,
    'GL_MAX_VARYING_COMPONENTS': 0,
    'GL_MAX_VARYING_VECTORS': 31,
    'GL_MAX_VARYING_FLOATS': 0,
    'GL_MAX_VERTEX_ATTRIBS': 16,
    'GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS': 16,
```

(continues on next page)

(continued from previous page)

```
    'GL_MAX_VERTEX_UNIFORM_COMPONENTS': 4096,
    'GL_MAX_VERTEX_UNIFORM_VECTORS': 1024,
    'GL_MAX_VERTEX_OUTPUT_COMPONENTS': 128,
    'GL_MAX_VERTEX_UNIFORM_BLOCKS': 14,
    'GL_MAX_VERTEX_ATTRIB_RELATIVE_OFFSET': 0,
    'GL_MAX_VERTEX_ATTRIB_BINDINGS': 0,
    'GL_VIEWPORT_BOUNDS_RANGE': (-32768, 32768),
    'GL_VIEWPORT_SUBPIXEL_BITS': 0,
    'GL_MAX_VIEWPORTS': 16
}
```

Type dict

`Context.`**`mglo`**
Internal representation for debug purposes only.

`Context.`**`extra`**
Any - Attribute for storing user defined objects

3.1.5 Context Flags

Context flags are used to enable or disable states in the context. These are not the same enum values as in opengl, but are rather bit flags so we can `or` them together setting multiple states in a simple way.

These values are available in the `Context` object and in the `moderngl` module when you don't have access to the context.

```
import moderngl

# From moderngl
ctx.enable_only(moderngl.DEPTH_TEST | moderngl.CULL_FACE)

# From context
ctx.enable_only(ctx.DEPTH_TEST | ctx.CULL_FACE)
```

`Context.`**`NOTHING = 0`**
Represents no states. Can be used with *`Context.enable_only()`* to disable all states.

`Context.`**`BLEND = 1`**
Enable/disable blending

`Context.`**`DEPTH_TEST = 2`**
Enable/disable depth testing

`Context.`**`CULL_FACE = 4`**
Enable/disable face culling

`Context.`**`RASTERIZER_DISCARD = 8`**
Enable/disable rasterization

`Context.`**`PROGRAM_POINT_SIZE = 16`**
When enabled we can write to `gl_PointSize` in the vertex shader to specify the point size. When disabled *`Context.point_size`* is used.

3.1.6 Blend Functions

Blend functions are used with *Context.blend_func* to control blending operations.

```
# Default value
ctx.blend_func = ctx.SRC_ALPHA, ctx.ONE_MINUS_SRC_ALPHA
```

Context.**ZERO = 0**

Context.**ONE = 1**

Context.**SRC_COLOR = 768**

Context.**ONE_MINUS_SRC_COLOR = 769**

Context.**SRC_ALPHA = 770**

Context.**ONE_MINUS_SRC_ALPHA = 771**

Context.**DST_ALPHA = 772**

Context.**ONE_MINUS_DST_ALPHA = 773**

Context.**DST_COLOR = 774**

Context.**ONE_MINUS_DST_COLOR = 775**

3.1.7 Blend Function Shortcuts

Context.**DEFAULT_BLENDING = (770, 771)**
: Shotcut for the default blending `SRC_ALPHA, ONE_MINUS_SRC_ALPHA`

Context.**ADDITIVE_BLENDING = (1, 1)**
: Shotcut for additive blending `ONE, ONE`

Context.**PREMULTIPLIED_ALPHA = (770, 1)**
: Shotcut for blend mode when using premultiplied alpha `SRC_ALPHA, ONE`

3.1.8 Blend Equations

Used with *Context.blend_equation*.

Context.**FUNC_ADD = 32774**
: source + destination

Context.**FUNC_SUBTRACT = 32778**
: source - destination

Context.**FUNC_REVERSE_SUBTRACT = 32779**
: destination - source

Context.**MIN = 32775**
: Minimum of source and destination

Context.**MAX = 32776**
: Maximum of source and destination

3.1.9 Other Enums

`Context.FIRST_VERTEX_CONVENTION = 36429`
: Specifies the first vertex should be used as the source of data for flat shaded varyings. Used with *Context.provoking_vertex*.

`Context.LAST_VERTEX_CONVENTION = 36430`
: Specifies the last vertex should be used as the source of data for flat shaded varyings. Used with *Context.provoking_vertex*.

3.1.10 Examples

ModernGL Context

```
import moderngl
# create a window
ctx = moderngl.create_context()
print(ctx.version_code)
```

Standalone ModernGL Context

```
import moderngl
ctx = moderngl.create_standalone_context()
print(ctx.version_code)
```

ContextManager

context_manager.py

```
import moderngl


class ContextManager:
    ctx = None

    @staticmethod
    def get_default_context(allow_fallback_standalone_context=True) -> moderngl.
↪Context:
        '''
            Default context
        '''

        if ContextManager.ctx is None:
            try:
                ContextManager.ctx = moderngl.create_context()
            except:
                if allow_fallback_standalone_context:
                    ContextManager.ctx = moderngl.create_standalone_context()
                else:
                    raise

        return ContextManager.ctx
```

example.py

```
from context_manager import ContextManager

ctx = ContextManager.get_default_context()
print(ctx.version_code)
```

3.2 Buffer

class moderngl.**Buffer**

Buffer objects are OpenGL objects that store an array of unformatted memory allocated by the OpenGL context, (data allocated on the GPU). These can be used to store vertex data, pixel data retrieved from images or the framebuffer, and a variety of other things.

A Buffer object cannot be instantiated directly, it requires a context. Use *Context.buffer()* to create one.

Copy buffer content using *Context.copy_buffer()*.

3.2.1 Create

Context.**buffer**(*data=None*, *reserve=0*, *dynamic=False*) → Buffer

Create a *Buffer* object.

Parameters **data** (*bytes*) – Content of the new buffer.

Keyword Arguments

- **reserve** (*int*) – The number of bytes to reserve.
- **dynamic** (*bool*) – Treat buffer as dynamic.

Returns *Buffer* object

3.2.2 Methods

Buffer.**assign**(*index*)

Helper method for assigning a buffer.

Returns (self, index) tuple

Buffer.**bind**(**attribs*, *layout=None*)

Helper method for binding a buffer.

Returns (self, layout, *attribs) tuple

Buffer.**write**(*data*, *offset=0*)

Write the content.

Parameters **data** (*bytes*) – The data.

Keyword Arguments **offset** (*int*) – The offset.

Buffer.**write_chunks**(*data*, *start*, *step*, *count*)

Split data to count equal parts.

Write the chunks using offsets calculated from start, step and stop.

Parameters

- **data** (*bytes*) – The data.
- **start** (*int*) – First offset.
- **step** (*int*) – Offset increment.
- **count** (*int*) – The number of offsets.

Buffer.**read**(*size=-1*, *offset=0*) → bytes
Read the content.

Parameters **size** (*int*) – The size. Value `-1` means all.

Keyword Arguments **offset** (*int*) – The offset.

Returns bytes

Buffer.**read_into**(*buffer*, *size=-1*, *offset=0*, *write_offset=0*)
Read the content into a buffer.

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the content.
- **size** (*int*) – The size. Value `-1` means all.

Keyword Arguments

- **offset** (*int*) – The read offset.
- **write_offset** (*int*) – The write offset.

Buffer.**read_chunks**(*chunk_size*, *start*, *step*, *count*) → bytes
Read the content.

Read and concatenate the chunks of size chunk_size using offsets calculated from start, step and stop.

Parameters

- **chunk_size** (*int*) – The chunk size.
- **start** (*int*) – First offset.
- **step** (*int*) – Offset increment.
- **count** (*int*) – The number of offsets.

Returns bytes

Buffer.**read_chunks_into**(*buffer*, *chunk_size*, *start*, *step*, *count*, *write_offset=0*)
Read the content.

Read and concatenate the chunks of size chunk_size using offsets calculated from start, step and stop.

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the content.
- **chunk_size** (*int*) – The chunk size.
- **start** (*int*) – First offset.
- **step** (*int*) – Offset increment.
- **count** (*int*) – The number of offsets.

Keyword Arguments **write_offset** (*int*) – The write offset.

Buffer.**clear**(*size=-1*, *offset=0*, *chunk=None*)
Clear the content.

Parameters **size** (*int*) – The size. Value `-1` means all.

Keyword Arguments

- **offset** (*int*) – The offset.
- **chunk** (*bytes*) – The chunk to use repeatedly.

`Buffer.`**bind_to_uniform_block**(*binding=0*, *offset=0*, *size=-1*)
Bind the buffer to a uniform block.

Parameters **binding** (*int*) – The uniform block binding.

Keyword Arguments

- **offset** (*int*) – The offset.
- **size** (*int*) – The size. Value `-1` means all.

`Buffer.`**bind_to_storage_buffer**(*binding=0*, *offset=0*, *size=-1*)
Bind the buffer to a shader storage buffer.

Parameters **binding** (*int*) – The shader storage binding.

Keyword Arguments

- **offset** (*int*) – The offset.
- **size** (*int*) – The size. Value `-1` means all.

`Buffer.`**orphan**(*size=-1*)
Orphan the buffer with the option to specify a new size.

It is also called buffer re-specification.

Reallocate the buffer object before you start modifying it.

Since allocating storage is likely faster than the implicit synchronization, you gain significant performance advantages over synchronization.

The old storage will still be used by the OpenGL commands that have been sent previously. It is likely that the GL driver will not be doing any allocation at all, but will just be pulling an old free block off the unused buffer queue and use it, so it is likely to be very efficient.

Keyword Arguments **size** (*int*) – The new byte size if the buffer. If not supplied the buffer size will be unchanged.

Example

```
# For simplicity the VertexArray creation is omitted

>>> vbo = ctx.buffer(reserve=1024)

# Fill the buffer

>>> vbo.write(some_temporary_data)

# Issue a render call that uses the vbo

>>> vao.render(...)

# Orphan the buffer

```

(continues on next page)

(continued from previous page)

```
>>> vbo.orphan()

# Issue another render call without waiting for the previous one

>>> vbo.write(some_temporary_data)
>>> vao.render(...)

# We can also resize the buffer. In this case we double the size

>> vbo.orphan(vbo.size * 2)
```

Buffer.**release**()
Release the ModernGL object.

3.2.3 Attributes

Buffer.**size**
The size of the buffer.

Type int

Buffer.**dynamic**
Is the buffer created with the dynamic flag?

Type bool

Buffer.**glo**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

Buffer.**mglo**
Internal representation for debug purposes only.

Buffer.**extra**
Any - Attribute for storing user defined objects

Buffer.**ctx**
The context this object belongs to

3.3 VertexArray

class moderngl.**VertexArray**
A VertexArray object is an OpenGL object that stores all of the state needed to supply vertex data. It stores the format of the vertex data as well as the Buffer objects providing the vertex data arrays.

In ModernGL, the VertexArray object also stores a reference for a *Program* object, and some Subroutine information.

A VertexArray object cannot be instantiated directly, it requires a context. Use *Context.vertex_array()* or *Context.simple_vertex_array()* to create one.

Note: Compared to OpenGL, *VertexArray* objects have some additional responsibilities:

- Binding a *Program* when *VertexArray.render()* or *VertexArray.transform()* is called.
- Subroutines can be assigned. Please see the example below.

3.3.1 Create

Context.**simple_vertex_array**(*program*, *buffer*, **attributes*, *index_buffer=None*, *index_element_size=4*) → VertexArray

Create a *VertexArray* object.

> **Warning:** This method is deprecated and may be removed in the future. Use *Context.vertex_array()* instead. It also supports the argument format this method describes.

Parameters

- **program** (Program) – The program used when rendering.
- **buffer** (Buffer) – The buffer.
- **attributes** (*list*) – A list of attribute names.

Keyword Arguments

- **index_element_size** (*int*) – byte size of each index element, 1, 2 or 4.
- **index_buffer** (Buffer) – An index buffer.

Returns *VertexArray* object

Context.**vertex_array**(**args*, ***kwargs*) → VertexArray

Create a *VertexArray* object.

This method also supports arguments for *Context.simple_vertex_array()*.

Parameters

- **program** (Program) – The program used when rendering.
- **content** (*list*) – A list of (buffer, format, attributes). See *Buffer Format*.
- **index_buffer** (Buffer) – An index buffer.

Keyword Arguments

- **index_element_size** (*int*) – byte size of each index element, 1, 2 or 4.
- **skip_errors** (*bool*) – Ignore skip_errors varyings.

Returns *VertexArray* object

3.3.2 Methods

VertexArray.**render**(*mode=None*, *vertices=-1*, *first=0*, *instances=-1*)

The render primitive (mode) must be the same as the input primitive of the GeometryShader.

Parameters

- **mode** (*int*) – By default TRIANGLES will be used.
- **vertices** (*int*) – The number of vertices to transform.

Keyword Arguments

- **first** (*int*) – The index of the first vertex to start with.

- **instances** (*int*) – The number of instances.

VertexArray.**render_indirect**(*buffer*, *mode=None*, *count=-1*, *first=0*)
The render primitive (mode) must be the same as the input primitive of the GeometryShader.

The draw commands are 5 integers: (count, instanceCount, firstIndex, baseVertex, baseInstance).

Parameters

- **buffer** (Buffer) – Indirect drawing commands.
- **mode** (*int*) – By default TRIANGLES will be used.
- **count** (*int*) – The number of draws.

Keyword Arguments **first** (*int*) – The index of the first indirect draw command.

VertexArray.**transform**(*buffer*, *mode=None*, *vertices=-1*, *first=0*, *instances=-1*, *buffer_offset=0*)
Transform vertices. Stores the output in a single buffer. The transform primitive (mode) must be the same as the input primitive of the GeometryShader.

Parameters

- **buffer** (Buffer) – The buffer to store the output.
- **mode** (*int*) – By default POINTS will be used.
- **vertices** (*int*) – The number of vertices to transform.

Keyword Arguments

- **first** (*int*) – The index of the first vertex to start with.
- **instances** (*int*) – The number of instances.
- **buffer_offset** (*int*) – Byte offset for the output buffer

VertexArray.**bind**(*attribute*, *cls*, *buffer*, *fmt*, *offset=0*, *stride=0*, *divisor=0*, *normalize=False*)
Bind individual attributes to buffers.

Parameters

- **location** (*int*) – The attribute location.
- **cls** (*str*) – The attribute class. Valid values are f, i or d.
- **buffer** (Buffer) – The buffer.
- **format** (*str*) – The buffer format.

Keyword Arguments

- **offset** (*int*) – The offset.
- **stride** (*int*) – The stride.
- **divisor** (*int*) – The divisor.
- **normalize** (*bool*) – The normalize parameter, if applicable.

VertexArray.**release**()
Release the ModernGL object.

3.3.3 Attributes

`VertexArray.`**`program`**
The program assigned to the VertexArray. The program used when rendering or transforming primitives.

Type *Program*

`VertexArray.`**`index_buffer`**
The index buffer if the index_buffer is set, otherwise `None`.

Type *Buffer*

`VertexArray.`**`index_element_size`**
The byte size of each element in the index buffer

Type int

`VertexArray.`**`scope`**
The *moderngl.Scope*.

`VertexArray.`**`vertices`**
The number of vertices detected. This is the minimum of the number of vertices possible per Buffer. The size of the index_buffer determines the number of vertices. Per instance vertex attributes does not affect this number.

Type int

`VertexArray.`**`instances`**
Get or set the number of instances to render

Type int

`VertexArray.`**`subroutines`**
The subroutines assigned to the VertexArray. The subroutines used when rendering or transforming primitives.

Type tuple

`VertexArray.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`VertexArray.`**`mglo`**
Internal representation for debug purposes only.

`VertexArray.`**`extra`**
Any - Attribute for storing user defined objects

`VertexArray.`**`ctx`**
The context this object belongs to

3.4 Program

`class` `moderngl.`**`Program`**
A Program object represents fully processed executable code in the OpenGL Shading Language, for one or more Shader stages.

In ModernGL, a Program object can be assigned to *VertexArray* objects. The VertexArray object is capable of binding the Program object once the *VertexArray.render()* or *VertexArray.transform()* is called.

Program objects has no method called `use()`, VertexArrays encapsulate this mechanism.

A Program object cannot be instantiated directly, it requires a context. Use *Context.program()* to create one.

Uniform buffers can be bound using *Buffer.bind_to_uniform_block()* or can be set individually. For more complex binding yielding higher performance consider using *moderngl.Scope*.

3.4.1 Create

Context.**program**(*vertex_shader*, *fragment_shader=None*, *geometry_shader=None*, *tess_control_shader=None*, *tess_evaluation_shader=None*, *varyings=()*) → Program

Create a *Program* object.

Only linked programs will be returned.

A single shader in the *shaders* parameter is also accepted. The varyings are only used when a transform program is created.

Parameters

- **shaders** (*list*) – A list of Shader objects.
- **varyings** (*list*) – A list of varying names.

Returns *Program* object

3.4.2 Methods

Program.**get**(*key*, *default*) → Union[Uniform, UniformBlock, Subroutine, Attribute, Varying]

Returns a Uniform, UniformBlock, Subroutine, Attribute or Varying.

Parameters **default** – This is the value to be returned in case key does not exist.

Returns *Uniform*, *UniformBlock*, *Subroutine*, *Attribute* or *Varying*

Program.**__getitem__**(*key*) → Union[Uniform, UniformBlock, Subroutine, Attribute, Varying]

Get a member such as uniforms, uniform blocks, subroutines, attributes and varyings by name.

```
# Get a uniform
uniform = program['color']

# Uniform values can be set on the returned object
# or the `__setitem__` shortcut can be used.
program['color'].value = 1.0, 1.0, 1.0, 1.0

# Still when writing byte data we need to use the `write()` method
program['color'].write(buffer)
```

Program.**__setitem__**(*key*, *value*)

Set a value of uniform or uniform block

```
# Set a vec4 uniform
uniform['color'] = 1.0, 1.0, 1.0, 1.0

# Optionally we can store references to a member and set the value directly
uniform = program['color']
uniform.value = 1.0, 0.0, 0.0, 0.0
```

(continues on next page)

(continued from previous page)

```
uniform = program['cameraMatrix']
uniform.write(camera_matrix)
```

Program.**__iter__**() → Generator[str, NoneType, NoneType]

Yields the internal members names as strings. This includes all members such as uniforms, attributes etc.

Example:

```
# Print member information
for name in program:
    member = program[name]
    print(name, type(member), member)
```

Output:

```
vert <class 'moderngl.program_members.attribute.Attribute'> <Attribute: 0>
vert_color <class 'moderngl.program_members.attribute.Attribute'> <Attribute: 1>
gl_InstanceID <class 'moderngl.program_members.attribute.Attribute'> <Attribute: -
↪1>
rotation <class 'moderngl.program_members.uniform.Uniform'> <Uniform: 0>
scale <class 'moderngl.program_members.uniform.Uniform'> <Uniform: 1>
```

We can filter on member type if needed:

```
for name in prog:
    member = prog[name]
    if isinstance(member, moderngl.Uniform):
        print("Uniform", name, member)
```

or a less verbose version using dict comprehensions:

```
uniforms = {name: self.prog[name] for name in self.prog
            if isinstance(self.prog[name], moderngl.Uniform)}
print(uniforms)
```

Output:

```
{'rotation': <Uniform: 0>, 'scale': <Uniform: 1>}
```

Program.**__eq__**(*other*) → bool

Compares two programs opengl names (mglo).

Returns If the programs have the same opengl name

Return type bool

Example:

```
# True if the internal opengl name is the same
program_1 == program_2
```

Program.**release**()

Release the ModernGL object.

3.4.3 Attributes

`Program.`**`geometry_input`**
The geometry input primitive. The GeometryShader's input primitive if the GeometryShader exists. The geometry input primitive will be used for validation.

Type int

`Program.`**`geometry_output`**
The geometry output primitive. The GeometryShader's output primitive if the GeometryShader exists.

Type int

`Program.`**`geometry_vertices`**
The maximum number of vertices that the geometry shader will output.

Type int

`Program.`**`subroutines`**
The subroutine uniforms.

Type tuple

`Program.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`Program.`**`mglo`**
Internal representation for debug purposes only.

`Program.`**`extra`**
Any - Attribute for storing user defined objects

`Program.`**`ctx`**
The context this object belongs to

3.4.4 Examples

A simple program designed for rendering

```
my_render_program = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 vert;

        void main() {
            gl_Position = vec4(vert, 0.0, 1.0);
        }
    ''',
    fragment_shader='''
        #version 330

        out vec4 color;

        void main() {
            color = vec4(0.3, 0.5, 1.0, 1.0);
        }
```

(continues on next page)

(continued from previous page)

```
    ''',
)
```

A simple program designed for transforming

```
my_transform_program = ctx.program(
    vertex_shader='''
        #version 330

        in vec4 vert;
        out float vert_length;

        void main() {
            vert_length = length(vert);
        }
    ''',
    varyings=['vert_length']
)
```

3.4.5 Program Members

Uniform

class moderngl.**Uniform**

A uniform is a global GLSL variable declared with the "uniform" storage qualifier. These act as parameters that the user of a shader program can pass to that program.

In ModernGL, Uniforms can be accessed using *Program.__getitem__()* or *Program.__iter__()*

Methods

Uniform.**read**() → bytes

Read the value of the uniform.

Uniform.**write**(*data*)

Write the value of the uniform.

Attributes

Uniform.**location**

The location of the uniform. The location holds the value returned by the glGetUniformLocation. To set the value of the uniform use the *value* instead.

Type int

Uniform.**dimension**

The dimension of the uniform.

GLSL type	dimension
sampler2D	1
sampler2DCube	1
sampler2DShadow	1
bool	1
bvec2	2
bvec3	3
bvec4	4
int	1
ivec2	2
ivec3	3
ivec4	4
uint	1
uvec2	2
uvec3	3
uvec4	4
float	1
vec2	2
vec3	3
vec4	4
double	1
dvec2	2
dvec3	3
dvec4	4
mat2	4
mat2x3	6
mat2x4	8
mat3x2	6
mat3	9
mat3x4	12
mat4x2	8
mat4x3	12
mat4	16
dmat2	4
dmat2x3	6
dmat2x4	8
dmat3x2	6
dmat3	9
dmat3x4	12
dmat4x2	8
dmat4x3	12
dmat4	16

Type int

`Uniform.`**`array_length`**
The length of the array of the uniform. The array_length is *1* for non array uniforms.

Type int

`Uniform.`**`name`**
The name of the uniform. The name does not contain leading *[0]*. The name may contain *[ ]* when the uniform is part of a struct.

Type str

`Uniform.`**`value`**
The value of the uniform. Reading the value of the uniform may force the GPU to sync.

The value must be a tuple for non array uniforms. The value must be a list of tuples for array uniforms.

`Uniform.`**`extra`**
Any - Attribute for storing user defined objects

`Uniform.`**`mglo`**
Internal representation for debug purposes only.

UniformBlock

`class` `moderngl.`**`UniformBlock`**

`UniformBlock.`**`binding`**
The binding of the uniform block.

Type int

`UniformBlock.`**`value`**
The value of the uniform block.

Type int

`UniformBlock.`**`name`**
The name of the uniform block.

Type str

`UniformBlock.`**`index`**
The index of the uniform block.

Type int

`UniformBlock.`**`size`**
The size of the uniform block.

Type int

`UniformBlock.`**`extra`**
Any - Attribute for storing user defined objects

`UniformBlock.`**`mglo`**
Internal representation for debug purposes only.

Subroutine

`class` `moderngl.`**`Subroutine`**
This class represents a program subroutine.

`Subroutine.`**`index`**
The index of the subroutine.

Type int

`Subroutine.`**`name`**
The name of the subroutine.

Type str

`Subroutine.`**`extra`**
Any - Attribute for storing user defined objects

Attribute

`class` `moderngl.`**`Attribute`**
This class represents a program attribute.

`Attribute.`**`location`**
The location of the attribute. The result of the glGetAttribLocation.

Type int

`Attribute.`**`array_length`**
If the attribute is an array the array_length is the length of the array otherwise *1*.

Type int

`Attribute.`**`dimension`**
The attribute dimension.

GLSL type	dimension
int	1
ivec2	2
ivec3	3
ivec4	4
uint	1
uvec2	2
uvec3	3
uvec4	4
float	1
vec2	2
vec3	3
vec4	4
double	1
dvec2	2
dvec3	3
dvec4	4
mat2	4
mat2x3	6
mat2x4	8
mat3x2	6
mat3	9
mat3x4	12
mat4x2	8
mat4x3	12
mat4	16
dmat2	4
dmat2x3	6
dmat2x4	8
dmat3x2	6
dmat3	9
dmat3x4	12

Continued on next page

Table 2 – continued from previous page

GLSL type	dimension
dmat4x2	8
dmat4x3	12
dmat4	16

Type int

`Attribute.`**`shape`**
The shape is a single character, representing the scalar type of the attribute.

shape	GLSL types
`'i'`	int
	ivec2 ivec3 ivec4
`'I'`	uint
	uvec2 uvec3 uvec4
`'f'`	float
	vec2 vec3 vec4
	mat2 mat3 mat4
	mat2x3 mat2x4 mat3x4 mat4x2 mat4x2 mat4x3
`'d'`	double
	dvec2 dvec3 dvec4
	dmat2 dmat3 dmat4
	dmat2x3 dmat2x4 dmat3x4 dmat4x2 dmat4x2 dmat4x3

Type str

`Attribute.`**`name`**
The attribute name. The name will be filtered to have no array syntax on it's end. Attribute name without `'[0]'` ending if any.

Type str

`Attribute.`**`extra`**
Any - Attribute for storing user defined objects

Varying

`class` `moderngl.`**`Varying`**
This class represents a program varying.

`Varying.`**`name`**
The name of the varying.

Type str

`Varying.`**`number`**
The number of the varying.

Type int

`Varying.`**`extra`**
Any - Attribute for storing user defined objects

3.5 Sampler

class moderngl.**Sampler**

A Sampler Object is an OpenGL Object that stores the sampling parameters for a Texture access inside of a shader. When a sampler object is bound to a texture image unit, the internal sampling parameters for a texture bound to the same image unit are all ignored. Instead, the sampling parameters are taken from this sampler object.

Unlike textures, a samplers state can also be changed freely be at any time without the sampler object being bound/in use.

Samplers are bound to a texture unit and not a texture itself. Be careful with leaving samplers bound to texture units as it can cause texture incompleteness issues (the texture bind is ignored).

Sampler bindings do clear automatically between every frame so a texture unit need at least one bind/use per frame.

3.5.1 Create

Context.**sampler**(*repeat_x=True*, *repeat_y=True*, *repeat_z=True*, *filter=None*, *anisotropy=1.0*, *compare_func='?'*, *border_color=None*, *min_lod=-1000.0*, *max_lod=1000.0*, *texture=None*) → Sampler

Create a *Sampler* object.

Keyword Arguments

- **repeat_x** (*bool*) – Repeat texture on x
- **repeat_y** (*bool*) – Repeat texture on y
- **repeat_z** (*bool*) – Repeat texture on z
- **filter** (*tuple*) – The min and max filter
- **anisotropy** (*float*) – Number of samples for anisotropic filtering. Any value greater than 1.0 counts as a use of anisotropic filtering
- **compare_func** – Compare function for depth textures
- **border_color** (*tuple*) – The (r, g, b, a) color for the texture border. When this value is set the `repeat_` values are overridden setting the texture wrap to return the border color when outside `[0, 1]` range.
- **min_lod** (*float*) – Minimum level-of-detail parameter (Default `-1000.0`). This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level)
- **max_lod** (*float*) – Minimum level-of-detail parameter (Default `1000.0`). This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level)
- **texture** (`Texture`) – The texture for this sampler

3.5.2 Methods

Sampler.**use**(*location=0*)

Bind the sampler to a texture unit

Parameters **location** (*int*) – The texture unit

Sampler.**clear**(*location=0*)

Clear the sampler binding on a texture unit

Parameters **location** (*int*) – The texture unit

Sampler.**assign**(*index*)
Helper method for assigning samplers to scopes.

Example:

```
s1 = ctx.sampler(...)
s2 = ctx.sampler(...)
ctx.scope(samplers=(s1.assign(0), s1.assign(1)), ...)
```

Returns (self, index) tuple

Sampler.**release**()
Release/destroy the ModernGL object.

3.5.3 Attributes

Sampler.**texture**

Sampler.**repeat_x**
The x repeat flag for the sampler (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
sampler.repeat_x = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
sampler.repeat_x = False
```

Type bool

Sampler.**repeat_y**
The y repeat flag for the sampler (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
sampler.repeat_y = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
sampler.repeat_y = False
```

Type bool

Sampler.**repeat_z**
The z repeat flag for the sampler (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
sampler.repeat_z = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
sampler.repeat_z = False
```

Type bool

Sampler.**filter**
The minification and magnification filter for the sampler. (Default `(moderngl.LINEAR. moderngl.LINEAR)`)

Example:

```
sampler.filter == (moderngl.NEAREST, moderngl.NEAREST)
sampler.filter == (moderngl.LINEAR_MIPMAP_LINEAR, moderngl.LINEAR)
sampler.filter == (moderngl.NEAREST_MIPMAP_LINEAR, moderngl.NEAREST)
sampler.filter == (moderngl.LINEAR_MIPMAP_NEAREST, moderngl.NEAREST)
```

Type tuple

Sampler.**compare_func**
The compare function for a depth textures (Default `'?'`)

By default samplers don't have depth comparison mode enabled. This means that depth texture values can be read as a `sampler2D` using `texture()` in a GLSL shader by default.

When setting this property to a valid compare mode, `GL_TEXTURE_COMPARE_MODE` is set to `GL_COMPARE_REF_TO_TEXTURE` so that texture lookup functions in GLSL will return a depth comparison result instead of the actual depth value.

Accepted compare functions:

```
.compare_func = ''    # Disale depth comparison completely
sampler.compare_func = '<='  # GL_LEQUAL
sampler.compare_func = '<'   # GL_LESS
sampler.compare_func = '>='  # GL_GEQUAL
sampler.compare_func = '>'   # GL_GREATER
sampler.compare_func = '=='  # GL_EQUAL
sampler.compare_func = '!='  # GL_NOTEQUAL
sampler.compare_func = '0'   # GL_NEVER
sampler.compare_func = '1'   # GL_ALWAYS
```

Type tuple

Sampler.**anisotropy**
Number of samples for anisotropic filtering (Default `1.0`). The value will be clamped in range `1.0` and `ctx.max_anisotropy`.

Any value greater than 1.0 counts as a use of anisotropic filtering:

```
# Disable anisotropic filtering
sampler.anisotropy = 1.0

# Enable anisotropic filtering suggesting 16 samples as a maximum
sampler.anisotropy = 16.0
```

Type float

Sampler.**border_color**
When setting this value the `repeat_` values are overridden setting the texture wrap to return the border color when outside [0, 1] range.

Example:

```
# Red border color
sampler.border_color = (1.0, 0.0, 0.0, 0.0)
```

`Sampler.`**`min_lod`**
Minimum level-of-detail parameter (Default `-1000.0`). This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level)

Type float

`Sampler.`**`max_lod`**
Minimum level-of-detail parameter (Default `1000.0`). This floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level)

Type float

`Sampler.`**`extra`**
Any - Attribute for storing user defined objects

`Sampler.`**`mglo`**
Internal representation for debug purposes only.

`Sampler.`**`ctx`**
The context this object belongs to

3.6 Texture

`class` `moderngl.`**`Texture`**
A Texture is an OpenGL object that contains one or more images that all have the same image format. A texture can be used in two ways. It can be the source of a texture access from a Shader, or it can be used as a render target.

A Texture object cannot be instantiated directly, it requires a context. Use *`Context.texture()`* or *`Context.depth_texture()`* to create one.

3.6.1 Create

`Context.`**`texture`**(*size*, *components*, *data=None*, *samples=0*, *alignment=1*, *dtype='f1'*) → Texture
Create a *`Texture`* object.

Parameters

- **`size`** (*`tuple`*) – The width and height of the texture.
- **`components`** (*`int`*) – The number of components 1, 2, 3 or 4.
- **`data`** (*`bytes`*) – Content of the texture.

Keyword Arguments

- **`samples`** (*`int`*) – The number of samples. Value 0 means no multisample format.
- **`alignment`** (*`int`*) – The byte alignment 1, 2, 4 or 8.
- **`dtype`** (*`str`*) – Data type.

Returns *`Texture`* object

`Context.`**`depth_texture`**(*size*, *data=None*, *samples=0*, *alignment=4*) → Texture
Create a *`Texture`* object.

Parameters

- **size** (*tuple*) – The width and height of the texture.
- **data** (*bytes*) – Content of the texture.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.

Returns *Texture* object

3.6.2 Methods

`Texture.`**`read`**(*level=0*, *alignment=1*) → bytes

Read the pixel data as bytes into system memory.

Keyword Arguments

- **level** (*int*) – The mipmap level.
- **alignment** (*int*) – The byte alignment of the pixels.

Returns bytes

`Texture.`**`read_into`**(*buffer*, *level=0*, *alignment=1*, *write_offset=0*)

Read the content of the texture into a bytearray or *Buffer*. The advantage of reading into a *Buffer* is that pixel data does not need to travel all the way to system memory:

```
# Reading pixel data into a bytearray
data = bytearray(4)
texture = ctx.texture((2, 2), 1)
texture.read_into(data)

# Reading pixel data into a buffer
data = ctx.buffer(reserve=4)
texture = ctx.texture((2, 2), 1)
texture.read_into(data)
```

Parameters **buffer** (*Union[bytearray, Buffer]*) – The buffer that will receive the pixels.

Keyword Arguments

- **level** (*int*) – The mipmap level.
- **alignment** (*int*) – The byte alignment of the pixels.
- **write_offset** (*int*) – The write offset.

`Texture.`**`write`**(*data*, *viewport=None*, *level=0*, *alignment=1*)

Update the content of the texture from byte data or a moderngl *Buffer*:

```
# Write data from a moderngl Buffer
data = ctx.buffer(reserve=4)
texture = ctx.texture((2, 2), 1)
texture.write(data)

# Write data from bytes
```

(continues on next page)

(continued from previous page)

```
data = b'ÿÿÿÿ'
texture = ctx.texture((2, 2), 1)
texture.write(data)
```

Parameters

- **data** (*Union[bytes, Buffer]*) – The pixel data.
- **viewport** (*tuple*) – The viewport.

Keyword Arguments

- **level** (*int*) – The mipmap level.
- **alignment** (*int*) – The byte alignment of the pixels.

Texture.**build_mipmaps**(*base=0*, *max_level=1000*)

Generate mipmaps.

This also changes the texture filter to `LINEAR_MIPMAP_LINEAR, LINEAR` (Will be removed in `6.x`)

Keyword Arguments

- **base** (*int*) – The base level
- **max_level** (*int*) – The maximum levels to generate

Texture.**bind_to_image**(*unit: int*, *read: bool = True*, *write: bool = True*, *level: int = 0*, *format: int = 0*)

Bind a texture to an image unit (OpenGL 4.2 required)

This is used to bind textures to image units for shaders. The idea with image load/store is that the user can bind one of the images in a Texture to a number of image binding points (which are separate from texture image units). Shaders can read information from these images and write information to them, in ways that they cannot with textures.

It's important to specify the right access type for the image. This can be set with the `read` and `write` arguments. Allowed combinations are:

- **Read-only**: `read=True` and `write=False`
- **Write-only**: `read=False` and `write=True`
- **Read-write**: `read=True` and `write=True`

`format` specifies the format that is to be used when performing formatted stores into the image from shaders. `format` must be compatible with the texture's internal format. **By default the format of the texture is passed in. The format parameter is only needed when overriding this behavior.**

More information:

- https://www.khronos.org/opengl/wiki/Image_Load_Store
- https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBindImageTexture.xhtml

Parameters

- **unit** (*int*) – Specifies the index of the image unit to which to bind the texture
- **texture** (*moderngl.Texture*) – The texture to bind

Keyword Arguments

- **read** (*bool*) – Allows the shader to read the image (default: `True`)

- **write** (*bool*) – Allows the shader to write to the image (default: `True`)
- **level** (*int*) – Level of the texture to bind (default: `0`).
- **format** (*int*) – (optional) The OpenGL enum value representing the format (defaults to the texture's format)

`Texture.use(location=0)`

Bind the texture to a texture unit.

The location is the texture unit we want to bind the texture. This should correspond with the value of the `sampler2D` uniform in the shader because samplers read from the texture unit we assign to them:

```
# Define what texture unit our two sampler2D uniforms should represent
program['texture_a'] = 0
program['texture_b'] = 1
# Bind textures to the texture units
first_texture.use(location=0)
second_texture.use(location=1)
```

Parameters **location** (*int*) – The texture location/unit.

`Texture.release()`

Release the ModernGL object.

3.6.3 Attributes

`Texture.repeat_x`

The x repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_x = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_x = False
```

Type bool

`Texture.repeat_y`

The y repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_y = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_y = False
```

Type bool

`Texture.filter`

The minification and magnification filter for the texture. (Default `(moderngl.LINEAR. moderngl.LINEAR)`)

Example:

```
texture.filter == (moderngl.NEAREST, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_LINEAR, moderngl.LINEAR)
texture.filter == (moderngl.NEAREST_MIPMAP_LINEAR, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_NEAREST, moderngl.NEAREST)
```

Type tuple

Texture.**swizzle**

The swizzle mask of the texture (Default `'RGBA'`).

The swizzle mask change/reorder the `vec4` value returned by the `texture()` function in a GLSL shaders. This is represented by a 4 character string were each character can be:

```
'R' GL_RED
'G' GL_GREEN
'B' GL_BLUE
'A' GL_ALPHA
'0' GL_ZERO
'1' GL_ONE
```

Example:

```
# Alpha channel will always return 1.0
texture.swizzle = 'RGB1'

# Only return the red component. The rest is masked to 0.0
texture.swizzle = 'R000'

# Reverse the components
texture.swizzle = 'ABGR'
```

Type str

Texture.**compare_func**

The compare function of the depth texture (Default `'<='`)

By default depth textures have `GL_TEXTURE_COMPARE_MODE` set to `GL_COMPARE_REF_TO_TEXTURE`, meaning any texture lookup will return a depth comparison value.

If you need to read the actual depth value in shaders, setting `compare_func` to a blank string will set `GL_TEXTURE_COMPARE_MODE` to `GL_NONE` making you able to read the depth texture as a `sampler2D`:

```
uniform sampler2D depth;
out vec4 fragColor;
in vec2 uv;

void main() {
    float raw_depth_nonlinear = texture(depth, uv);
    fragColor = vec4(raw_depth_nonlinear);
}
```

Accepted compare functions:

```
texture.compare_func = ''    # Disale depth comparison completely
texture.compare_func = '<='  # GL_LEQUAL
```

(continues on next page)

(continued from previous page)

```
texture.compare_func = '<'    # GL_LESS
texture.compare_func = '>='   # GL_GEQUAL
texture.compare_func = '>'    # GL_GREATER
texture.compare_func = '=='   # GL_EQUAL
texture.compare_func = '!='   # GL_NOTEQUAL
texture.compare_func = '0'    # GL_NEVER
texture.compare_func = '1'    # GL_ALWAYS
```

Type tuple

`Texture.`**`anisotropy`**
Number of samples for anisotropic filtering (Default `1.0`). The value will be clamped in range `1.0` and `ctx.max_anisotropy`.

Any value greater than 1.0 counts as a use of anisotropic filtering:

```
# Disable anisotropic filtering
texture.anisotropy = 1.0

# Enable anisotropic filtering suggesting 16 samples as a maximum
texture.anisotropy = 16.0
```

Type float

`Texture.`**`width`**
The width of the texture.

Type int

`Texture.`**`height`**
The height of the texture.

Type int

`Texture.`**`size`**
The size of the texture.

Type tuple

`Texture.`**`dtype`**
Data type.

Type str

`Texture.`**`components`**
The number of components of the texture.

Type int

`Texture.`**`samples`**
The number of samples set for the texture used in multisampling.

Type int

`Texture.`**`depth`**
Is the texture a depth texture?

Type bool

`Texture.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`Texture.`**`mglo`**
Internal representation for debug purposes only.

`Texture.`**`extra`**
Any - Attribute for storing user defined objects

`Texture.`**`ctx`**
The context this object belongs to

3.7 TextureArray

`class` `moderngl.`**`TextureArray`**
An Array Texture is a Texture where each mipmap level contains an array of images of the same size. Array textures may have Mipmaps, but each mipmap in the texture has the same number of levels.

A TextureArray object cannot be instantiated directly, it requires a context. Use *`Context.texture_array()`* to create one.

3.7.1 Create

`Context.`**`texture_array`**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → TextureArray
Create a *`TextureArray`* object.

Parameters

- **`size`** (*`tuple`*) – The `(width, height, layers)` of the texture.
- **`components`** (*`int`*) – The number of components 1, 2, 3 or 4.
- **`data`** (*`bytes`*) – Content of the texture. The size must be `(width, height * layers)` so each layer is stacked vertically.

Keyword Arguments

- **`alignment`** (*`int`*) – The byte alignment 1, 2, 4 or 8.
- **`dtype`** (*`str`*) – Data type.

Returns *`Texture3D`* object

3.7.2 Methods

`TextureArray.`**`read`**(*alignment=1*) → bytes
Read the pixel data as bytes into system memory.

Keyword Arguments **`alignment`** (*`int`*) – The byte alignment of the pixels.

Returns bytes

`TextureArray.`**`read_into`**(*buffer*, *alignment=1*, *write_offset=0*)
Read the content of the texture array into a bytearray or *`Buffer`*. The advantage of reading into a *`Buffer`* is that pixel data does not need to travel all the way to system memory:

```
# Reading pixel data into a bytearray
data = bytearray(8)
texture = ctx.texture((2, 2, 2), 1)
texture.read_into(data)

# Reading pixel data into a buffer
data = ctx.buffer(reserve=8)
texture = ctx.texture((2, 2, 2), 1)
texture.read_into(data)
```

Parameters **buffer** (*Union[bytearray, Buffer]*) – The buffer that will receive the pixels.

Keyword Arguments

- **alignment** (*int*) – The byte alignment of the pixels.
- **write_offset** (*int*) – The write offset.

TextureArray.**write**(*data*, *viewport=None*, *alignment=1*)

Update the content of the texture array from byte data or a moderngl *Buffer*.

The `viewport` can be used for finer control of where the data should be written in the array. The valid versions are:

```
# Writing multiple layers from the begining of the texture
texture.write(data, viewport=(width, hight, num_layers))

# Writing sub-sections of the array
texture.write(data, viewport=(x, y, layer, width, height, num_layers))
```

Like with other texture types we can also use bytes or *Buffer* as a source:

```
# Using a moderngl buffer
data = ctx.buffer(reserve=8)
texture = ctx.texture_array((2, 2, 2), 1)
texture.write(data)

# Using byte data from system memory
data = b"ÿÿÿÿÿÿÿÿ"
texture = ctx.texture_array((2, 2, 2), 1)
texture.write(data)
```

Parameters

- **data** (*bytes*) – The pixel data.
- **viewport** (*tuple*) – The viewport.

Keyword Arguments **alignment** (*int*) – The byte alignment of the pixels.

TextureArray.**build_mipmaps**(*base=0*, *max_level=1000*)

Generate mipmaps.

This also changes the texture filter to `LINEAR_MIPMAP_LINEAR, LINEAR` (Will be removed in `6.x`)

Keyword Arguments

- **base** (*int*) – The base level

- **max_level** (*int*) – The maximum levels to generate

TextureArray.**use**(*location=0*)

Bind the texture to a texture unit.

The location is the texture unit we want to bind the texture. This should correspond with the value of the `sampler2DArray` uniform in the shader because samplers read from the texture unit we assign to them:

```
# Define what texture unit our two sampler2DArray uniforms should represent
program['texture_a'] = 0
program['texture_b'] = 1
# Bind textures to the texture units
first_texture.use(location=0)
second_texture.use(location=1)
```

Parameters **location** (*int*) – The texture location/unit.

TextureArray.**release**()

Release the ModernGL object.

3.7.3 Attributes

TextureArray.**repeat_x**

The x repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_x = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_x = False
```

Type bool

TextureArray.**repeat_y**

The y repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_y = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_y = False
```

Type bool

TextureArray.**filter**

The minification and magnification filter for the texture. (Default `(moderngl.LINEAR. moderngl.LINEAR)`)

Example:

```
texture.filter == (moderngl.NEAREST, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_LINEAR, moderngl.LINEAR)
texture.filter == (moderngl.NEAREST_MIPMAP_LINEAR, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_NEAREST, moderngl.NEAREST)
```

Type tuple

TextureArray.**swizzle**

The swizzle mask of the texture (Default `'RGBA'`).

The swizzle mask change/reorder the `vec4` value returned by the `texture()` function in a GLSL shaders. This is represented by a 4 character string were each character can be:

```
'R' GL_RED
'G' GL_GREEN
'B' GL_BLUE
'A' GL_ALPHA
'0' GL_ZERO
'1' GL_ONE
```

Example:

```
# Alpha channel will always return 1.0
texture.swizzle = 'RGB1'

# Only return the red component. The rest is masked to 0.0
texture.swizzle = 'R000'

# Reverse the components
texture.swizzle = 'ABGR'
```

Type str

TextureArray.**anisotropy**

Number of samples for anisotropic filtering (Default `1.0`). The value will be clamped in range `1.0` and `ctx.max_anisotropy`.

Any value greater than 1.0 counts as a use of anisotropic filtering:

```
# Disable anisotropic filtering
texture.anisotropy = 1.0

# Enable anisotropic filtering suggesting 16 samples as a maximum
texture.anisotropy = 16.0
```

Type float

TextureArray.**width**

The width of the texture array.

Type int

TextureArray.**height**

The height of the texture array.

Type int

`TextureArray.`**`layers`**
The number of layers of the texture array.

Type int

`TextureArray.`**`size`**
The size of the texture array.

Type tuple

`TextureArray.`**`dtype`**
Data type.

Type str

`TextureArray.`**`components`**
The number of components of the texture array.

Type int

`TextureArray.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`TextureArray.`**`mglo`**
Internal representation for debug purposes only.

`TextureArray.`**`extra`**
Any - Attribute for storing user defined objects

`TextureArray.`**`ctx`**
The context this object belongs to

3.8 Texture3D

`class` `moderngl.`**`Texture3D`**
A Texture is an OpenGL object that contains one or more images that all have the same image format. A texture can be used in two ways. It can be the source of a texture access from a Shader, or it can be used as a render target.

A Texture3D object cannot be instantiated directly, it requires a context. Use *`Context.texture3d()`* to create one.

3.8.1 Create

`Context.`**`texture3d`**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → Texture3D
Create a *`Texture3D`* object.

Parameters

- **`size`** (*`tuple`*) – The width, height and depth of the texture.
- **`components`** (*`int`*) – The number of components 1, 2, 3 or 4.
- **`data`** (*`bytes`*) – Content of the texture.

Keyword Arguments

- **`alignment`** (*`int`*) – The byte alignment 1, 2, 4 or 8.

- **dtype** (*str*) – Data type.

Returns *Texture3D* object

3.8.2 Methods

Texture3D.**read**(*alignment=1*) → bytes
Read the pixel data as bytes into system memory.

Keyword Arguments **alignment** (*int*) – The byte alignment of the pixels.

Returns bytes

Texture3D.**read_into**(*buffer*, *alignment=1*, *write_offset=0*)
Read the content of the texture into a bytearray or *Buffer*. The advantage of reading into a *Buffer* is that pixel data does not need to travel all the way to system memory:

```
# Reading pixel data into a bytearray
data = bytearray(8)
texture = ctx.texture3d((2, 2, 2), 1)
texture.read_into(data)

# Reading pixel data into a buffer
data = ctx.buffer(reserve=8)
texture = ctx.texture3d((2, 2), 1)
texture.read_into(data)
```

Parameters **buffer** (*Union[bytearray, Buffer]*) – The buffer that will receive the pixels.

Keyword Arguments

- **alignment** (*int*) – The byte alignment of the pixels.
- **write_offset** (*int*) – The write offset.

Texture3D.**write**(*data*, *viewport=None*, *alignment=1*)
Update the content of the texture from byte data or a moderngl *Buffer*:

```
# Write data from a moderngl Buffer
data = ctx.buffer(reserve=8)
texture = ctx.texture3d((2, 2, 2), 1)
texture.write(data)

# Write data from bytes
data = b'ÿÿÿÿÿÿÿÿ'
texture = ctx.texture3d((2, 2), 1)
texture.write(data)
```

Parameters

- **data** (*bytes*) – The pixel data.
- **viewport** (*tuple*) – The viewport.

Keyword Arguments **alignment** (*int*) – The byte alignment of the pixels.

Texture3D.**build_mipmaps**(*base=0*, *max_level=1000*)
Generate mipmaps.

This also changes the texture filter to `LINEAR_MIPMAP_LINEAR, LINEAR` (Will be removed in `6.x`)

Keyword Arguments

- **base** (*int*) – The base level
- **max_level** (*int*) – The maximum levels to generate

Texture3D.**use**(*location=0*)
Bind the texture to a texture unit.

The location is the texture unit we want to bind the texture. This should correspond with the value of the `sampler3D` uniform in the shader because samplers read from the texture unit we assign to them:

```
# Define what texture unit our two sampler3D uniforms should represent
program['texture_a'] = 0
program['texture_b'] = 1
# Bind textures to the texture units
first_texture.use(location=0)
second_texture.use(location=1)
```

Parameters **location** (*int*) – The texture location/unit.

Texture3D.**release**()
Release the ModernGL object.

3.8.3 Attributes

Texture3D.**repeat_x**
The x repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_x = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_x = False
```

Type bool

Texture3D.**repeat_y**
The y repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_y = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_y = False
```

Type bool

Texture3D.**repeat_z**
The z repeat flag for the texture (Default `True`)

Example:

```
# Enable texture repeat (GL_REPEAT)
texture.repeat_z = True

# Disable texture repeat (GL_CLAMP_TO_EDGE)
texture.repeat_z = False
```

Type bool

Texture3D.**filter**
The filter of the texture.

Type tuple

Texture3D.**swizzle**
The swizzle mask of the texture (Default `'RGBA'`).

The swizzle mask change/reorder the `vec4` value returned by the `texture()` function in a GLSL shaders. This is represented by a 4 character string were each character can be:

```
'R' GL_RED
'G' GL_GREEN
'B' GL_BLUE
'A' GL_ALPHA
'0' GL_ZERO
'1' GL_ONE
```

Example:

```
# Alpha channel will always return 1.0
texture.swizzle = 'RGB1'

# Only return the red component. The rest is masked to 0.0
texture.swizzle = 'R000'

# Reverse the components
texture.swizzle = 'ABGR'
```

Type str

Texture3D.**width**
The width of the texture.

Type int

Texture3D.**height**
The height of the texture.

Type int

Texture3D.**depth**
The depth of the texture.

Type int

`Texture3D.`**`size`**
The size of the texture.

Type tuple

`Texture3D.`**`dtype`**
Data type.

Type str

`Texture3D.`**`components`**
The number of components of the texture.

Type int

`Texture3D.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`Texture3D.`**`mglo`**
Internal representation for debug purposes only.

`Texture3D.`**`extra`**
Any - Attribute for storing user defined objects

`Texture3D.`**`ctx`**
The context this object belongs to

3.9 TextureCube

class `moderngl.`**`TextureCube`**
A Texture is an OpenGL object that contains one or more images that all have the same image format. A texture can be used in two ways. It can be the source of a texture access from a Shader, or it can be used as a render target.

Note: ModernGL enables `GL_TEXTURE_CUBE_MAP_SEAMLESS` globally to ensure filtering will be done across the cube faces.

A Texture3D object cannot be instantiated directly, it requires a context. Use *`Context.texture_cube()`* to create one.

3.9.1 Create

`Context.`**`texture_cube`**(*size*, *components*, *data=None*, *alignment=1*, *dtype='f1'*) → TextureCube
Create a *`TextureCube`* object.

Parameters

- **`size`** (*`tuple`*) – The width, height of the texture. Each side of the cube will have this size.
- **`components`** (*`int`*) – The number of components 1, 2, 3 or 4.
- **`data`** (*`bytes`*) – Content of the texture. The data should be have the following ordering: positive_x, negative_x, positive_y, negative_y, positive_z + negative_z

Keyword Arguments

- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.

Returns *TextureCube* object

3.9.2 Methods

TextureCube.**read**(*face*, *alignment=1*) → bytes

Read a face from the cubemap as bytes into system memory.

Parameters **face** (*int*) – The face to read.

Keyword Arguments **alignment** (*int*) – The byte alignment of the pixels.

TextureCube.**read_into**(*buffer*, *face*, *alignment=1*, *write_offset=0*)

Read a face from the cubemap texture.

Read a face of the cubemap into a bytearray or *Buffer*. The advantage of reading into a *Buffer* is that pixel data does not need to travel all the way to system memory:

```
# Reading pixel data into a bytearray
data = bytearray(4)
texture = ctx.texture_cube((2, 2), 1)
texture.read_into(data, 0)

# Reading pixel data into a buffer
data = ctx.buffer(reserve=4)
texture = ctx.texture_cube((2, 2), 1)
texture.read_into(data, 0)
```

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the pixels.
- **face** (*int*) – The face to read.

Keyword Arguments

- **alignment** (*int*) – The byte alignment of the pixels.
- **write_offset** (*int*) – The write offset.

TextureCube.**write**(*face*, *data*, *viewport=None*, *alignment=1*)

Update the content of the texture.

Update the content of a face in the cubemap from byte data or a moderngl *Buffer*:

```
# Write data from a moderngl Buffer
data = ctx.buffer(reserve=4)
texture = ctx.texture_cube((2, 2), 1)
texture.write(0, data)

# Write data from bytes
data = b'ÿÿÿÿ'
texture = ctx.texture_cube((2, 2), 1)
texture.write(0, data)
```

Parameters

- **face** (*int*) – The face to update.
- **data** (*bytes*) – The pixel data.
- **viewport** (*tuple*) – The viewport.

Keyword Arguments **alignment** (*int*) – The byte alignment of the pixels.

TextureCube.**use**(*location=0*)
Bind the texture to a texture unit.

The location is the texture unit we want to bind the texture. This should correspond with the value of the samplerCube uniform in the shader because samplers read from the texture unit we assign to them:

```
# Define what texture unit our two samplerCube uniforms should represent
program['texture_a'] = 0
program['texture_b'] = 1
# Bind textures to the texture units
first_texture.use(location=0)
second_texture.use(location=1)
```

Parameters **location** (*int*) – The texture location/unit.

TextureCube.**release**()
Release the ModernGL object.

3.9.3 Attributes

TextureCube.**size**
The size of the texture.

Type tuple

TextureCube.**dtype**
Data type.

Type str

TextureCube.**components**
The number of components of the texture.

Type int

TextureCube.**filter**
The minification and magnification filter for the texture. (Default (moderngl.LINEAR. moderngl.LINEAR))

Example:

```
texture.filter == (moderngl.NEAREST, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_LINEAR, moderngl.LINEAR)
texture.filter == (moderngl.NEAREST_MIPMAP_LINEAR, moderngl.NEAREST)
texture.filter == (moderngl.LINEAR_MIPMAP_NEAREST, moderngl.NEAREST)
```

Type tuple

TextureCube.**swizzle**
The swizzle mask of the texture (Default 'RGBA').

The swizzle mask change/reorder the `vec4` value returned by the `texture()` function in a GLSL shaders. This is represented by a 4 character string were each character can be:

```
'R' GL_RED
'G' GL_GREEN
'B' GL_BLUE
'A' GL_ALPHA
'0' GL_ZERO
'1' GL_ONE
```

Example:

```
# Alpha channel will always return 1.0
texture.swizzle = 'RGB1'

# Only return the red component. The rest is masked to 0.0
texture.swizzle = 'R000'

# Reverse the components
texture.swizzle = 'ABGR'
```

Type str

`TextureCube.`**`anisotropy`**

Number of samples for anisotropic filtering (Default `1.0`). The value will be clamped in range `1.0` and `ctx.max_anisotropy`.

Any value greater than 1.0 counts as a use of anisotropic filtering:

```
# Disable anisotropic filtering
texture.anisotropy = 1.0

# Enable anisotropic filtering suggesting 16 samples as a maximum
texture.anisotropy = 16.0
```

Type float

`TextureCube.`**`glo`**

The internal OpenGL object. This values is provided for debug purposes only.

Type int

`TextureCube.`**`mglo`**

Internal representation for debug purposes only.

`TextureCube.`**`extra`**

Any - Attribute for storing user defined objects

`TextureCube.`**`ctx`**

The context this object belongs to

3.10 Framebuffer

`class` `moderngl.`**`Framebuffer`**

A *`Framebuffer`* is a collection of buffers that can be used as the destination for rendering. The buffers for Framebuffer objects reference images from either Textures or Renderbuffers.

Create a *Framebuffer* using *Context.framebuffer()*.

3.10.1 Create

Context.**simple_framebuffer**(*size*, *components=4*, *samples=0*, *dtype='f1'*) → Framebuffer
Creates a *Framebuffer* with a single color attachment and depth buffer using *moderngl.Renderbuffer* attachments.

Parameters

- **size** (*tuple*) – The width and height of the renderbuffer.
- **components** (*int*) – The number of components 1, 2, 3 or 4.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

Returns *Framebuffer* object

Context.**framebuffer**(*color_attachments=()*, *depth_attachment=None*) → Framebuffer
A *Framebuffer* is a collection of buffers that can be used as the destination for rendering. The buffers for Framebuffer objects reference images from either Textures or Renderbuffers.

Parameters

- **color_attachments** (*list*) – A list of *Texture* or *Renderbuffer* objects.
- **depth_attachment** (*Renderbuffer or Texture*) – The depth attachment.

Returns *Framebuffer* object

3.10.2 Methods

Framebuffer.**clear**(*red=0.0*, *green=0.0*, *blue=0.0*, *alpha=0.0*, *depth=1.0*, *viewport=None*, *color=None*)
Clear the framebuffer.

If a *viewport* passed in, a scissor test will be used to clear the given viewport. This viewport take prescense over the framebuffers *scissor*. Clearing can still be done with scissor if no viewport is passed in.

This method also respects the *color_mask* and *depth_mask*. It can for example be used to only clear the depth or color buffer or specific components in the color buffer.

If the *viewport* is a 2-tuple it will clear the `(0, 0, width, height)` where `(width, height)` is the 2-tuple.

If the *viewport* is a 4-tuple it will clear the given viewport.

Parameters

- **red** (*float*) – color component.
- **green** (*float*) – color component.
- **blue** (*float*) – color component.
- **alpha** (*float*) – alpha component.
- **depth** (*float*) – depth value.

Keyword Arguments

- **viewport** (*tuple*) – The viewport.
- **color** (*tuple*) – Optional tuple replacing the red, green, blue and alpha arguments

Framebuffer.**read**(*viewport=None*, *components=3*, *attachment=0*, *alignment=1*, *dtype='f1'*) → bytes
Read the content of the framebuffer.

Parameters

- **viewport** (*tuple*) – The viewport.
- **components** (*int*) – The number of components to read.

Keyword Arguments

- **attachment** (*int*) – The color attachment.
- **alignment** (*int*) – The byte alignment of the pixels.
- **dtype** (*str*) – Data type.

Returns bytes

Framebuffer.**read_into**(*buffer*, *viewport=None*, *components=3*, *attachment=0*, *alignment=1*, *dtype='f1'*, *write_offset=0*)
Read the content of the framebuffer into a buffer.

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the pixels.
- **viewport** (*tuple*) – The viewport.
- **components** (*int*) – The number of components to read.

Keyword Arguments

- **attachment** (*int*) – The color attachment.
- **alignment** (*int*) – The byte alignment of the pixels.
- **dtype** (*str*) – Data type.
- **write_offset** (*int*) – The write offset.

Framebuffer.**use**()
Bind the framebuffer. Sets the target for rendering commands.

Framebuffer.**release**()
Release the ModernGL object.

3.10.3 Attributes

Framebuffer.**viewport**
Get or set the viewport of the framebuffer.

Type tuple

Framebuffer.**scissor**
Get or set the scissor box of the framebuffer.

When scissor testing is enabled fragments outside the defined scissor box will be discarded. This applies to rendered geometry or *Framebuffer.clear()*.

Setting is value enables scissor testing in the framebuffer. Setting the scissor to `None` disables scissor testing and reverts the scissor box to match the framebuffer size.

Example:

```
# Enable scissor testing
>>> ctx.scissor = 100, 100, 200, 100
# Disable scissor testing
>>> ctx.scissor = None
```

Type tuple

`Framebuffer.`**`color_mask`**
The color mask of the framebuffer.

Color masking controls what components in color attachments will be affected by fragment write operations. This includes rendering geometry and clearing the framebuffer.

Default value: `(True, True, True, True)`.

Examples:

```
# Block writing to all color components (rgba) in color attachments
fbo.color_mask = False, False, False, False

# Re-enable writing to color attachments
fbo.color_mask = True, True, True, True

# Block fragment writes to alpha channel
fbo.color_mask = True, True, True, False
```

Type tuple

`Framebuffer.`**`depth_mask`**
The depth mask of the framebuffer.

Depth mask enables or disables write operations to the depth buffer. This also applies when clearing the framebuffer. If depth testing is enabled fragments will still be culled, but the depth buffer will not be updated with new values. This is a very useful tool in many rendering techniques.

Default value: `True`

Type bool

`Framebuffer.`**`width`**
The width of the framebuffer.

Framebuffers created by a window will only report its initial size. It's better get size information from the window itself.

Type int

`Framebuffer.`**`height`**
The height of the framebuffer.

Framebuffers created by a window will only report its initial size. It's better get size information from the window itself.

Type int

`Framebuffer.`**`size`**
The size of the framebuffer.

Framebuffers created by a window will only report its initial size. It's better get size information from the window itself.

Type tuple

`Framebuffer.`**`samples`**
The samples of the framebuffer.

Type int

`Framebuffer.`**`bits`**
The bits of the framebuffer.

Type dict

`Framebuffer.`**`color_attachments`**
The color attachments of the framebuffer.

Type tuple

`Framebuffer.`**`depth_attachment`**
The depth attachment of the framebuffer.

Type *Texture* or *Renderbuffer*

`Framebuffer.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`Framebuffer.`**`mglo`**
Internal representation for debug purposes only.

`Framebuffer.`**`extra`**
Any - Attribute for storing user defined objects

`Framebuffer.`**`ctx`**
The context this object belongs to

3.11 Renderbuffer

`class` `moderngl.`**`Renderbuffer`**
Renderbuffer objects are OpenGL objects that contain images. They are created and used specifically with `Framebuffer` objects. They are optimized for use as render targets, while `Texture` objects may not be, and are the logical choice when you do not need to sample from the produced image. If you need to resample, use Textures instead. Renderbuffer objects also natively accommodate multisampling.

A Renderbuffer object cannot be instantiated directly, it requires a context. Use `Context.renderbuffer()` or `Context.depth_renderbuffer()` to create one.

3.11.1 Create

`Context.`**`renderbuffer`**(*size*, *components=4*, *samples=0*, *dtype='f1'*) → Renderbuffer
`Renderbuffer` objects are OpenGL objects that contain images. They are created and used specifically with `Framebuffer` objects.

Parameters

- **`size`** (`tuple`) – The width and height of the renderbuffer.

- **components** (*int*) – The number of components 1, 2, 3 or 4.

Keyword Arguments

- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

Returns *Renderbuffer* object

Context.**depth_renderbuffer**(*size*, *samples=0*) → Renderbuffer

Renderbuffer objects are OpenGL objects that contain images. They are created and used specifically with *Framebuffer* objects.

Parameters **size** (*tuple*) – The width and height of the renderbuffer.

Keyword Arguments **samples** (*int*) – The number of samples. Value 0 means no multisample format.

Returns *Renderbuffer* object

3.11.2 Methods

Renderbuffer.**release**()

Release the ModernGL object.

3.11.3 Attributes

Renderbuffer.**width**

The width of the renderbuffer.

Type int

Renderbuffer.**height**

The height of the renderbuffer.

Type int

Renderbuffer.**size**

The size of the renderbuffer.

Type tuple

Renderbuffer.**samples**

The samples of the renderbuffer.

Type int

Renderbuffer.**components**

The components of the renderbuffer.

Type int

Renderbuffer.**depth**

Is the renderbuffer a depth renderbuffer?

Type bool

Renderbuffer.**dtype**

Data type.

Type str

`Renderbuffer.`**`glo`**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

`Renderbuffer.`**`mglo`**
Internal representation for debug purposes only.

`Renderbuffer.`**`extra`**
Any - Attribute for storing user defined objects

`Renderbuffer.`**`ctx`**
The context this object belongs to

3.12 Scope

`class` `moderngl.`**`Scope`**
This class represents a Scope object.

Responsibilities on enter:

- Set the enable flags.
- Bind the framebuffer.
- Assigning textures to texture locations.
- Assigning buffers to uniform buffers.
- Assigning buffers to shader storage buffers.

Responsibilities on exit:

- Restore the enable flags.
- Restore the framebuffer.

3.12.1 Create

`Context.`**`scope`**(*framebuffer=None*, *enable_only=None*, *textures=()*, *uniform_buffers=()*, *storage_buffers=()*, *samplers=()*, *enable=None*) → Scope
Create a *Scope* object.

Parameters

- **`framebuffer`** (`Framebuffer`) – The framebuffer to use when entering.
- **`enable_only`** (*int*) – The enable_only flags to set when entering.

Keyword Arguments

- **`textures`** (*list*) – List of (texture, binding) tuples.
- **`uniform_buffers`** (*list*) – List of (buffer, binding) tuples.
- **`storage_buffers`** (*list*) – List of (buffer, binding) tuples.
- **`samplers`** (*list*) – List of sampler bindings
- **`enable`** (*int*) – Flags to enable for this vao such as depth testing and blending

3.12.2 Attributes

`Scope.extra`
Any - Attribute for storing user defined objects

`Scope.mglo`
Internal representation for debug purposes only.

`Scope.ctx`
The context this object belongs to

3.12.3 Examples

Simple scope example

```
scope1 = ctx.scope(fbo1, moderngl.BLEND)
scope2 = ctx.scope(fbo2, moderngl.DEPTH_TEST | moderngl.CULL_FACE)

with scope1:
    # do some rendering

with scope2:
    # do some rendering
```

Scope for querying

```
query = ctx.query(samples=True)
scope = ctx.scope(ctx.screen, moderngl.DEPTH_TEST | moderngl.RASTERIZER_DISCARD)

with scope, query:
    # do some rendering

print(query.samples)
```

Understanding what scope objects do

```
scope = ctx.scope(
    framebuffer=framebuffer1,
    enable_only=moderngl.BLEND,
    textures=[
        (texture1, 4),
        (texture2, 3),
    ],
    uniform_buffers=[
        (buffer1, 6),
        (buffer2, 5),
    ],
    storage_buffers=[
        (buffer3, 8),
    ],
)
```

(continues on next page)

(continued from previous page)

```
# Let's assume we have some state before entering the scope
some_random_framebuffer.use()
some_random_texture.use(3)
some_random_buffer.bind_to_uniform_block(5)
some_random_buffer.bind_to_storage_buffer(8)
ctx.enable_only(moderngl.DEPTH_TEST)

with scope:
    # on __enter__
    #     framebuffer1.use()
    #     ctx.enable_only(moderngl.BLEND)
    #     texture1.use(4)
    #     texture2.use(3)
    #     buffer1.bind_to_uniform_block(6)
    #     buffer2.bind_to_uniform_block(5)
    #     buffer3.bind_to_storage_buffer(8)

    # do some rendering

    # on __exit__
    #     some_random_framebuffer.use()
    #     ctx.enable_only(moderngl.DEPTH_TEST)

# Originally we had the following, let's see what was changed
some_random_framebuffer.use()                   # This was restored hurray!
some_random_texture.use(3)                      # Have to restore it manually.
some_random_buffer.bind_to_uniform_block(5)     # Have to restore it manually.
some_random_buffer.bind_to_storage_buffer(8)    # Have to restore it manually.
ctx.enable_only(moderngl.DEPTH_TEST)            # This was restored too.

# Scope objects only do as much as necessary.
# Restoring the framebuffer and enable flags are lowcost operations and
# without them you could get a hard time debugging the application.
```

3.13 Query

class moderngl.**Query**
This class represents a Query object.

3.13.1 Create

Context.**query**(*samples=False*, *any_samples=False*, *time=False*, *primitives=False*) → Query
Create a *Query* object.

Keyword Arguments

- **samples** (*bool*) – Query GL_SAMPLES_PASSED or not.
- **any_samples** (*bool*) – Query GL_ANY_SAMPLES_PASSED or not.
- **time** (*bool*) – Query GL_TIME_ELAPSED or not.
- **primitives** (*bool*) – Query GL_PRIMITIVES_GENERATED or not.

3.13.2 Attributes

`Query.`**`samples`**
The number of samples passed.

Type int

`Query.`**`primitives`**
The number of primitives generated.

Type int

`Query.`**`elapsed`**
The time elapsed in nanoseconds.

Type int

`Query.`**`crender`**
Can be used in a `with` statement.

Type *ConditionalRender*

`Query.`**`extra`**
Any - Attribute for storing user defined objects

`Query.`**`mglo`**
Internal representation for debug purposes only.

`Query.`**`ctx`**
The context this object belongs to

3.13.3 Examples

Simple query example

```
import moderngl
import numpy as np

ctx = moderngl.create_standalone_context()
prog = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 in_vert;

        void main() {
            gl_Position = vec4(in_vert, 0.0, 1.0);
        }
    ''',
    fragment_shader='''
        #version 330

        out vec4 color;

        void main() {
            color = vec4(1.0, 0.0, 0.0, 1.0);
        }
    ''',
)
```

(continues on next page)

(continued from previous page)

```

vertices = np.array([
    0.0, 0.0,
    1.0, 0.0,
    0.0, 1.0,
], dtype='f4')

vbo = ctx.buffer(vertices.tobytes())
vao = ctx.simple_vertex_array(prog, vbo, 'in_vert')

fbo = ctx.simple_framebuffer((64, 64))
fbo.use()

query = ctx.query(samples=True, time=True)

with query:
    vao.render()

print('It took %d nanoseconds' % query.elapsed)
print('to render %d samples' % query.samples)
```

Output

```
It took 13529 nanoseconds
to render 496 samples
```

3.14 ConditionalRender

class moderngl.**ConditionalRender**

This class represents a ConditionalRender object.

ConditionalRender objects can only be accessed from *Query* objects.

3.14.1 Attributes

ConditionalRender.**mglo**

Internal representation for debug purposes only.

3.14.2 Examples

Simple conditional rendering example

```
query = ctx.query(any_samples=True)

with query:
    vao1.render()

with query.crender:
    print('This will always get printed')
    vao2.render()  # But this will be rendered only if vao1 has passing samples.
```

3.15 ComputeShader

class moderngl.**ComputeShader**

A Compute Shader is a Shader Stage that is used entirely for computing arbitrary information. While it can do rendering, it is generally used for tasks not directly related to drawing.

- Compute shaders support uniforms are other member object just like a *moderngl.Program*.
- Storage buffers can be bound using *Buffer.bind_to_storage_buffer()*.
- Uniform buffers can be bound using *Buffer.bind_to_uniform_block()*.
- Images can be bound using *Texture.bind_to_image()*.

3.15.1 Create

Context.**compute_shader**(*source*) → ComputeShader

A *ComputeShader* is a Shader Stage that is used entirely for computing arbitrary information. While it can do rendering, it is generally used for tasks not directly related to drawing.

Parameters **source** (*str*) – The source of the compute shader.

Returns *ComputeShader* object

3.15.2 Methods

ComputeShader.**run**(*group_x=1*, *group_y=1*, *group_z=1*)

Run the compute shader.

Parameters

- **group_x** (*int*) – The number of work groups to be launched in the X dimension.
- **group_y** (*int*) – The number of work groups to be launched in the Y dimension.
- **group_z** (*int*) – The number of work groups to be launched in the Z dimension.

ComputeShader.**get**(*key*, *default*) → Union[Uniform, UniformBlock, Subroutine, Attribute, Varying]

Returns a Uniform, UniformBlock, Subroutine, Attribute or Varying.

Parameters **default** – This is the value to be returned in case key does not exist.

Returns *Uniform*, *UniformBlock*, *Subroutine*, *Attribute* or *Varying*

ComputeShader.**release**()

Release the ModernGL object.

ComputeShader.**__eq__**(*other*)

Compares to compute shaders ensuring the internal opengl name/id is the same

ComputeShader.**__getitem__**(*key*) → Union[Uniform, UniformBlock, Subroutine, Attribute, Varying]

Get a member such as uniforms, uniform blocks, subroutines, attributes and varyings by name.

```
# Get a uniform
uniform = program['color']

# Uniform values can be set on the returned object
# or the `__setitem__` shortcut can be used.
program['color'].value = 1.0, 1.0, 1.0, 1.0
```

(continues on next page)

(continued from previous page)

```
# Still when writing byte data we need to use the `write()` method
program['color'].write(buffer)
```

ComputeShader.**__setitem__**(*key*, *value*)
Set a value of uniform or uniform block

```
# Set a vec4 uniform
uniform['color'] = 1.0, 1.0, 1.0, 1.0

# Optionally we can store references to a member and set the value directly
uniform = program['color']
uniform.value = 1.0, 0.0, 0.0, 0.0

uniform = program['cameraMatrix']
uniform.write(camera_matrix)
```

ComputeShader.**__iter__**() → Generator[str, NoneType, NoneType]
Yields the internal members names as strings. This includes all members such as uniforms, attributes etc.

3.15.3 Attributes

ComputeShader.**glo**
The internal OpenGL object. This values is provided for debug purposes only.

Type int

ComputeShader.**mglo**
Internal representation for debug purposes only.

ComputeShader.**extra**
Any - Attribute for storing user defined objects

ComputeShader.**ctx**
The context this object belongs to

CHAPTER 4

Miscellaneous

4.1 Differences between ModernGL5 and ModernGL4

4.1.1 Package Name

ModernGL4

```
import ModernGL  # mixed case
```

ModernGL5

```
import moderngl  # this is pep8 style
```

4.1.2 Program Creation

ModernGL4

```
my_program = ctx.program([  # extra list
    # vertex_shader returned a Shader object
    ctx.vertex_shader('''
        ...
    '''),
```

(continues on next page)

(continued from previous page)

```
    # fragment_shader returned a Shader object
    ctx.fragment_shader('''
        ...
    '''),
])
```

ModernGL5

```
my_program = ctx.program(  # no list needed
    # vertex_shader is a keyword argument
    vertex_shader='''
        ...
    ''',
    # fragment_shader is a keyword argument
    fragment_shader='''
        ...
    ''',
)
```

4.1.3 Program Varyings

ModernGL4

```
my_program = ctx.program(
    ctx.vertex_shader('''
        ...
    '''),
    ['out_vert', 'out_norm']  # no keyword argument needed
])
```

ModernGL5

```
my_program = ctx.program(
    vertex_shader='''
        ...
    ''',
    varyings=['out_vert', 'out_norm'],  # varyings are explicitly given
)
```

4.1.4 Program Members

ModernGL4

```
my_program.uniforms['ModelViewMatrix'].value = ...
my_program.uniform_buffers['UniformBuffer'].binding = ...
```

ModernGL5

```
my_program['ModelViewMatrix'].value = ...
my_program['UniformBuffer'].binding = ...
```

4.1.5 Texture Pixel Types

ModernGL4

```
my_texture = ctx.texture(size, 4, floats=True)  # floats or not floats
```

ModernGL5

```
my_texture = ctx.texture(size, 4, dtype='f4')  # floats=True
my_texture = ctx.texture(size, 4, dtype='f2')  # half-floats
my_texture = ctx.texture(size, 4, dtype='f1')  # floats=False
my_texture = ctx.texture(size, 4, dtype='i4')  # integers
```

This also apply for *Texture3D*, *TextureCube* and *Renderbuffer*.

4.1.6 Buffer Format

ModernGL4

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f3f', ['in_vert', 'in_norm']),  # extra list object
    #        ^ no space between the attributes
    ...
])
```

ModernGL5

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f 3f', 'in_vert', 'in_norm'),  # no list needed
    #         ^ space is obligatory
    ...
])
```

4.1.7 Buffer Format Half-Floats

ModernGL4

Not available in ModernGL4

ModernGL5

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f2 3f2', 'in_vert', 'in_norm'),  # '3f2' means '3' of 'f2', where 'f2'␣
↪is a half-float
    ...
])
```

4.1.8 Buffer Format Padding

ModernGL4

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f12x', ['in_vert']),  # same as above, in_norm was replaced with padding
    ...
])
```

ModernGL5

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f 3x4', ['in_vert']),  # '3x4' means '3' of 'x4', where 'x4' means 4␣
↪bytes of padding
    ...
])
```

4.1.9 Buffer Format Errors

Let's assume `in_vert` was declared as: `in vec4 in_vert`

ModernGL4

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f', ['in_vert']),  # throws an error (3 != 4)
    ...
])

my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '4i', ['in_vert']),  # throws an error (float != int)
    ...
])
```

ModernGL5

```
my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '3f', 'in_vert'),  # totally fine
    ...
])

my_vertex_array = ctx.vertex_array(prog, [
    (vbo1, '4i', 'in_vert'),  # totally fine
    ...
])
```

Found something not covered here? Please file an issue.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z