ModernGL Documentation

Release 5.12.0

Szabolcs Dombi

Nov 02, 2025

CONTENTS

ModernGL is a high performance rendering module for Python.

CHAPTER

ONE

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 experience issues it's probably related to context creation. More configuration might be needed to run moderngl in some cases. This is especially true on linux running without X. See the context section.

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

- To compile moderngl: `python setup.py build_ext --inplace`

Package and dev dependencies:

- Install `requirements.txt`, `tests/requirements.txt` and `docs/requirements.txt`
- Install the package in editable mode: `pip install -e .`

1.3 Using with Mesa 3D on Windows

If you have an old Graphics Card that raises errors when running moderngl, you can try using this method, to make Moderngl work.

There are essentially two ways,

- Compiling Mesa yourselves see https://docs.mesa3d.org/install.html.
- Using msys2, which provides pre-compiled Mesa binaries.

1.3.1 Using MSYS2

- Download and Install https://www.msys2.org/#installation
- Check whether you have 32-bit or 64-bit python.

32-bit python

If you have 32-bit python, then open `C:\msys64\mingw32.exe` and type the following

```
pacman -S mingw-w64-i686-mesa
```

It will install mesa and its dependencies. Then you can add `C:\msys64\mingw32\bin` to PATH before `C:\Windows` and moderngl should be working. Also, you should set an environment variable called `GLCONTEXT_WIN_LIBGL` which contains the path to opengl32 dll from mesa. In this case it should be `GLCONTEXT_WIN_LIBGL=C:\msys64\mingw32\bin\opengl32.dll`.

64-bit python

If you have 64-bit python, then open `C:\msys64\mingw64.exe` and type the following

```
pacman -S mingw-w64-x86_64-mesa
```

It will install mesa and it's dependencies. Then you can add `C:\msys64\mingw64\bin` to PATH before `C:\Windows` and moderngl should be working. Also, you should set an environment variable called `GLCONTEXT_WIN_LIBGL` which contains the path to opengl32 dll from mesa. In this case it should be `GLCONTEXT_WIN_LIBGL=C:\msys64\mingw64\bin\opengl32.dll`

1.4 Using ModernGL in CI

1.4.1 Windows CI Configuration

ModernGL can't be run directly on Windows CI without the use of Mesa. To get ModernGL running you should first install Mesa from the MSYS2 project and adding it to the `PATH`.

Steps

1. Usually MSYS2 project should be installed by default by your CI provider in `C:\msys64`. You can refer the documentation on how to get it installed and make sure to update it.
2. Then login through bash and enter `pacman -S --noconfirm mingw-w64-x86_64-mesa`.

   ```
   C:\msys64\usr\bin\bash -lc "pacman -S --noconfirm mingw-w64-x86_64-mesa"
   ```

   This will install Mesa binary, which moderngl would be using.
3. Then add `C:\msys64\mingw64\bin` to `PATH`.

   ```
   $env:PATH = "C:\msys64\mingw64\bin;$env:PATH"
   ```

   Warning: Make sure to delete `C:\msys64\mingw64\bin\python.exe` if it exists because the python provided by them would then be added to Global and some unexpected things may happen.
4. Then set an environment variable `GLCONTEXT_WIN_LIBGL=C:\msys64\mingw64\bin\opengl32.dll`. This will make glcontext use `C:\msys64\mingw64\bin\opengl32.dll` for opengl drivers.
5. Then you can run moderngl as you want to.

Example Configuration

A example configuration for Github Actions:

```
name: Hello World
on: [push, pull_request]

jobs:
  build:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - uses: msys2/setup-msys2@v2
        with:
          msystem: MINGW64
          release: false
          install: mingw-w64-x86_64-mesa
```

(continues on next page)

(continued from previous page)

```
      - name: Test using ModernGL
        shell: pwsh
        run: |
          Remove-Item C:\msys64\mingw64\bin\python.exe -Force
          $env:GLCONTEXT_WIN_LIBGL = "C:\msys64\mingw64\bin\opengl32.dll"
          python -m pip install -r requirements.txt
          python -m pytest
```

1.4.2 Linux

For running ModernGL on Linux CI, you would need to configure `xvfb` so that it starts a Window in the background. After that, you should be able to use ModernGL directly.

Steps

1. Install `xvfb` from Package Manager.

   ```
   sudo apt-get -y install xvfb
   ```

2. The run the below command, to start Xvfb from background.

   ```
   sudo /usr/bin/Xvfb :0 -screen 0 1280x1024x24 &
   ```

3. You can run ModernGL now.

Example Configuration

A example configuration for Github Actions:

```
name: Hello World
on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - name: Prepare
        run: |
            sudo apt-get -y install xvfb
            sudo /usr/bin/Xvfb :0 -screen 0 1280x1024x24 &
      - name: Test using ModernGL
        run: |
          python -m pip install -r requirements.txt
          python -m pytest
```

1.4.3 macOS

You won't need any special configuration to run on macOS.

CHAPTER

TWO

THE GUIDE

Consistent learning of ModernGL.

2.1 An introduction to OpenGL

2.1.1 The simplified story

OpenGL (Open Graphics Library) has a long history reaching all the way back to 1992 when it was created by Silicon Graphics. It was partly based in their proprietary IRIS GL (Integrated Raster Imaging System Graphics Library) library.

Today OpenGL is managed by the Khronos Group, an open industry consortium of over 150 leading hardware and software companies creating advanced, royalty-free, acceleration standards for 3D graphics, Augmented and Virtual Reality, vision and machine learning

The purpose of OpenGL is to provide a standard way to interact with the graphics processing unit to achieve hardware accelerated rendering across several platforms. How this is done under the hood is up to the vendors (AMD, Nvidia, Intel, ARM .. etc) as long as the the specifications are followed.

OpenGL has gone though many versions and it can be confusing when looking up resources. Today we separate "Old OpenGL" and "Modern OpenGL". From 2008 to 2010 version 3.x of OpenGL evolved until version 3.3 and 4.0 was released simultaneously.

In 2010 version 3.3, 4.0 and 4.1 was released to modernize the api (simplified explanation) creating something that would be able to utilize Direct3D 11-class hardware. **OpenGL 3.3 is the first "Modern OpenGL" version** (simplified explanation). Everything from this version is forward compatible all the way to the latest 4.x version. An optional deprecation mechanism was introduced to disable outdated features. Running OpenGL in **core mode** would remove all old features while running in **compatibility mode** would still allow mixing the old and new api.

Note: OpenGL 2.x, 3.0, 3.1 and 3.2 can of course access some modern OpenGL features directly, but for simplicity we are are focused on version 3.3 as it created the final standard we are using today. Older OpenGL was also a pretty wild world with countless vendor specific extensions. Modern OpenGL cleaned this up quite a bit.

In OpenGL we often talk about the **Fixed Pipeline** and the **Programmable Pipeline**.

OpenGL code using the **Fixed Pipeline** (Old OpenGL) would use functions like `glVertex`, `glColor`, `glMaterial` `glMatrixMode`, `glLoadIdentity`, `glBegin`, `glEnd`, `glVertexPointer`, `glColorPointer`, `glPushMatrix` and `glPopMatrix`. The api had strong opinions and limitations on what you could do, hiding what really went on under the hood.

OpenGL code using the **Programmable Pipeline** (Modern OpenGL) would use functions like `glCreateProgram`, `UseProgram`. `glCreateShader`, `VertexAttrib*`, `glBindBuffer*`, and `glUniform*`. This API mainly works with buffers of data and smaller programs called "shaders" running on the GPU to process this data using the **OpenGL**

Shading Language (GLSL). This gives enormous flexibility but requires that we understand the OpenGL pipeline (actually not that complicated).

2.1.2 Beyond OpenGL

OpenGL has a lot of "baggage" after 25 years and hardware has drastically changed since its inception. Plans for "OpenGL 5" was started as the **Next Generation OpenGL Initiative (glNext)**. This Turned into the Vulkan API and was a grounds-up redesign to unify OpenGL and OpenGL ES into one common API that will not be backwards compatible with existing OpenGL versions.

This doesn't mean OpenGL is not worth learning today. In fact learning 3.3+ shaders and understanding the rendering pipeline will greatly help you understand Vulkan. In most cases you can pretty much copy paste the shaders over to Vulkan.

2.1.3 Where does ModernGL fit into all this?

The ModernGL library exposes the **Programmable Pipeline** using OpenGL 3.3 core or higher. However, we don't expose OpenGL functions directly. Instead we expose features though various objects like *Buffer* and *Program* in a much more "pythonic" way. It's in other words a higher level wrapper making OpenGL much easier to reason with. We try to hide most of the complicated details to make the user more productive. There are a lot of pitfalls with OpenGL and we remove most of them.

Learning ModernGL is more about learning shaders and the OpenGL pipeline.

2.2 Getting started with ModernGL

A comprehensive guide to getting you started with ModernGL. No experience with OpenGL is required, just an understanding of how it works.

2.2.1 Low start

Low start in learning ModernGL. Coverage of all ModernGL objects with a glance and first use of GPU.

Creating a Context

Before we can do anything with ModernGL we need a *Context*. The *Context* object makes us able to create OpenGL resources. ModernGL can only create headless contexts (no window), but it can also detect and use contexts from a large range of window libraries. The moderngl-window library is a good start or reference for rendering to a window.

Most of the example code here assumes a `ctx` variable exists with a headless context:

```
# standalone=True makes a headless context
ctx = moderngl.create_context(standalone=True)
```

Detecting an active context created by a window library is simply:

```
ctx = moderngl.create_context()
```

More details about context creation can be found in the *Context Creation* section.

ModernGL Types

Before throwing you into doing shaders we'll go through some of the most important types/objects in ModernGL.

- *Buffer* is an OpenGL buffer we can for example write vertex data into. This data will reside in graphics memory.
- *Program* is a shader program. We can feed it GLSL source code as strings to set up our shader program
- *VertexArray* is a light object responsible for communication between *Buffer* and *Program* so it can understand how to access the provided buffers and do the rendering call. These objects are currently immutable but are cheap to make.
- *Texture*, *TextureArray*, *Texture3D* and *TextureCube* represents the different texture types. *Texture* is a 2d texture and is most commonly used.
- *Framebuffer* is an offscreen render target. It supports different attachments types such as a *Texture* and a depth texture/buffer.

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

- *Context.buffer()*
- *Context.program()*
- *Context.vertex_array()*
- *Context.texture()*
- *Context.texture_array()*
- *Context.texture3d()*
- *Context.texture_cube()*
- *Context.framebuffer()*

The ModernGL types cannot be extended as in; you cannot subclass them. Extending them must be done through substitution and not inheritance. This is related to performance. Most objects have an `extra` property that can contain any python object.

Shader Introduction

Shaders are small programs running on the GPU (Graphics Processing Unit). We are using a fairly simple language called GLSL (OpenGL Shading Language). This is a C-style language, so it covers most of the features you would expect with such a language. Control structures (for-loops, if-else statements, etc) exist in GLSL, including the switch statement.

Note: The name "shader" comes from the fact that these small GPU programs was originally created for shading (lighting) 3D scenes. This started as per-vertex lighting when the early shaders could only process vertices and evolved into per-pixel lighting when the fragment shader was introduced. They are used in many other areas today, but the name have stuck around.

Examples of types are:

```
bool value = true;
int value = 1;
uint value = 1;
float value = 0.0;
double value = 0.0;
```

Each type above also has a 2, 3 and 4 component version:

```
// float (default) type
vec2 value = vec2(0.0, 1.0);
vec3 value = vec3(0.0, 1.0, 2.0);
vec4 value = vec4(0.0);

// signed and unsigned integer vectors
ivec3 value = ivec3(0);
uvec3 value = ivec3(0);
// etc ..
```

More about GLSL data types can be found in the Khronos wiki.

The available functions are for example: `radians`, `degrees` `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `pow` `exp`, `log`, `exp2`, `log2`, `sqrt`, `inversesqrt`, `abs`, `sign`, `floor`, `ceil`, `fract`, `mod`, `min`, `max`, `clamp`, `mix`, `step`, `smoothstep`, `length`, `distance`, `dot`, `cross`, `normalize`, `faceforward`, `reflect`, `refract`, `any`, `all` etc.

All functions can be found in the OpenGL Reference Page (exclude functions starting with `gl`). Most of the functions exist in several overloaded versions supporting different data types.

The basic setup for a shader is the following:

```
#version 330

void main() {
}
```

The `#version` statement is mandatory and should at least be 330 (GLSL version 3.3 matching OpenGL version 3.3). The version statement **should always be the first line in the source code**. Higher version number is only needed if more fancy features are needed. By the time you need those you probably know what you are doing.

What we also need to realize when working with shaders is that they are executed in parallel across all the cores on your GPU. This can be everything from tens, hundreds, thousands or more cores. Even integrated GPUs today are very competent.

For those who have not worked with shaders before it can be mind-boggling to see the work they can get done in a matter of microseconds. All shader executions / rendering calls are also asynchronous running in the background while your python code is doing other things (but certain operations can cause a "sync" stalling until the shader program is done).

Let's try to use the shader in the *simplest way (next step)*.

Vertex Shader (transforms)

Let's get our hands dirty right away and jump into it by showing the simplest forms of shaders in OpenGL. These are called transforms or transform feedback. Instead of drawing to the screen we simply capture the output of a shader into a *Buffer*.

The example below shows shader program with only a vertex shader. It has no input data, but we can still force it to run N times. The `gl_VertexID` (int) variable is a built-in value in vertex shaders containing an integer representing the vertex number being processed.

Input variables in vertex shaders are called **attributes** (we have no inputs in this example) while output values are called **varyings**.

```
import struct
import moderngl

ctx = moderngl.create_context(standalone=True)

program = ctx.program(
    vertex_shader="""
    #version 330

    // Output values for the shader. They end up in the buffer.
    out float value;
    out float product;

    void main() {
        // Implicit type conversion from int to float will happen here
        value = gl_VertexID;
        product = gl_VertexID * gl_VertexID;
    }
    """,
    # What out varyings to capture in our buffer!
    varyings=["value", "product"],
)

NUM_VERTICES = 10

# We always need a vertex array in order to execute a shader program.
# Our shader doesn't have any buffer inputs, so we give it an empty array.
vao = ctx.vertex_array(program, [])

# Create a buffer allocating room for 20 32 bit floats
# num of vertices (10) * num of varyings per vertex (2) * size of float in bytes (4)
buffer = ctx.buffer(reserve=NUM_VERTICES * 2 * 4)

# Start a transform with buffer as the destination.
# We force the vertex shader to run 10 times
vao.transform(buffer, vertices=NUM_VERTICES)

# Unpack the 20 float values from the buffer (copy from graphics memory to system␣
↪memory).
# Reading from the buffer will cause a sync (the python program stalls until the shader␣
↪is done)
data = struct.unpack("20f", buffer.read())
for i in range(0, 20, 2):
    print("value = {}, product = {}".format(*data[i:i+2]))
```

Output of the program is:

```
value = 0.0, product = 0.0
value = 1.0, product = 1.0
value = 2.0, product = 4.0
value = 3.0, product = 9.0
value = 4.0, product = 16.0
value = 5.0, product = 25.0
```

(continues on next page)

(continued from previous page)

```
value = 6.0, product = 36.0
value = 7.0, product = 49.0
value = 8.0, product = 64.0
value = 9.0, product = 81.0
```

The GPU is at the very least slightly offended by the meager amount work we assigned it, but this at least shows the basic concept of transforms. We would in most situations also not read the results back into system memory because it's slow, but sometimes it is needed.

This shader program could for example be modified to generate some geometry or data for any other purpose you might imagine useful. Using modulus (`mod(number, divisor)`) on `gl_VertexID` can get you pretty far.

Warning: One known bug in many OpenGL drivers is related to the `mod(number, divisor)` function. It lies in the fact that if the first argument is exactly 2 times the second, then instead of `0` you will receive the value of the second argument [divisor]. To avoid this error, it is recommended to use the following additional function (insert it into the shader before `void main()` and then always call `mod2()` instead of `mod()`):

```
float mod2(float number, float divisor) {
    float result = mod(number, divisor);
    if (result >= divisor) {
        result = 0.0;
    }
    return result;
}
```

2.2.2 First rendering

Unlike OpenGL, ModernGL requires far fewer lines due to robust function generalization, which also reduces the likelihood of bugs when porting code to different platforms.

In this example, we will draw a broken line, which will look different each time we run the code, and at the same time we will slightly expand our horizons in understanding the already mentioned ModernGL objects.

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.

Previously, we looked at creating a *vertex shader-only program* that could only read data from the input buffer and write the converted data to the output buffer. Now let's add a fragment shader to our program; it will allow us to create an algorithm for writing pixels into a texture, that is, perform the *main function* of the shader.

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.

The program will also process a color information.

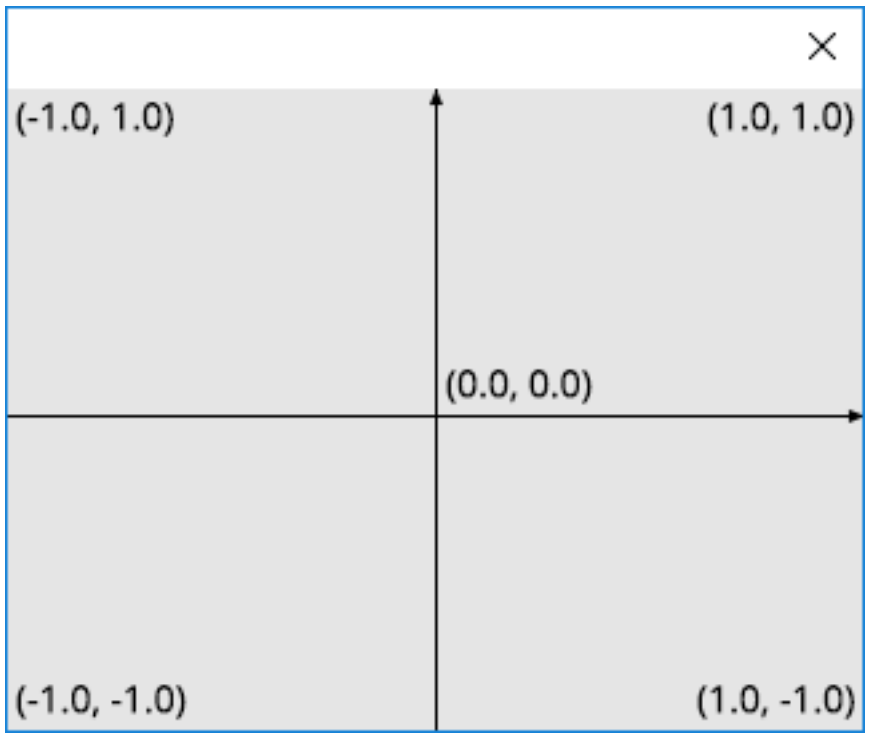


Fig. 1: The screen coordinates

Entire source

```
import moderngl

ctx = moderngl.create_context(standalone=True)

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;
```

(continues on next page)

(continued from previous page)

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

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;
}
```

Proceed to the *next step*.

Buffer

Buffer is a dedicated area of GPU memory. Can store anything in bytes.

Creating a buffer is essentially allocating an area of memory into which data is later written. Therefore, when creating an empty buffer, it may contain memory fragments of deleted objects. Example of creating an empty buffer:

```
buf = ctx.buffer(reserve=1)
```

The `reserve` parameter specifies how many bytes should be allocated, the minimum number is 1.

The buffer is cleared by writing zeros to the allocated memory area. However, this must be done manually:

```
buf.clear()
```

ModernGL allows you to create 2 types of buffer: dynamic and non-dynamic. To do this, when creating a buffer, use the keyword argument `dynamic=True/False` (by default `False`):

```
buf = ctx.buffer(reserve=32, dynamic=True)
```

Note: Using the `dynamic=True` parameter tells the GPU that actions with this *Buffer* will be performed very often. This parameter is optional, but is recommended if the *Buffer* is used frequently.

Later, using the `Buffer.orphan()` function, you can change the buffer size at any time:

```
buf.orphan(size=64)
```

After changing the buffer size, you will need to write data there. This is done using the *Buffer.write()* function. This function will write data from RAM to GPU memory:

```
buf.write(b'any bytes data')
```

However, if the size of this buffer was changed after it was added to *VertexArray*, then when calling *VertexArray.render()* you will need to specify the new number of vertices in the `vertices` parameter. For example:

```
# If from the contents of this buffer every 12 bytes fall on one vertex.
vao.render(vertices=buf.size // 4 // 3)
```

The same will need to be done when calling the *VertexArray.transform()* function:

```
# If from the contents of this buffer every 12 bytes fall on one vertex.
# output_buf - the buffer into which the transformation will be performed.
vao.transform(output_buf, vertices=buf.size // 4 // 3)
```

After *VertexArray.transform()* writes data to `output_buf` using a *vertex shader*, you may need to read it — use *Buffer.read()*. This function will read data from GPU memory into RAM:

```
bytes_data = buf.read()
```

Warning: Transferring data between RAM and GPU memory comes at a huge performance cost. It is recommended to use *Buffer.write()* and *Buffer.read()* as little as possible.

If you just need to copy data between buffers, look towards the *Context.copy_buffer()* function:

```
ctx.copy_buffer(destination_buf, source_buf)
```

In our example, we simply create a static buffer and write data immediately when it is created:

```
vbo = ctx.buffer(vertices.astype("f4").tobytes())
```

We called it VBO (Vertex Buffer Object) because we will store vertex data in this buffer.

Note: For the convenience of transferring data to the GPU memory [in a dedicated *Buffer* area], here we use the NumPy library.

NumPy installation:

```
pip install numpy
```

If you want fewer dependencies, you can try Python's built-in struct module with the `struct.pack()` and `struct.unpack()` methods.

Entire source

```
import moderngl
import numpy as np

ctx = moderngl.create_context(standalone=True)

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.zeros(50)
g = np.ones(50)
b = np.zeros(50)

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

vbo = ctx.buffer(vertices.astype("f4").tobytes())
```

Proceed to the *next step*.

Vertex Array

VertexArray is something like a pipeline, where as arguments *Context.vertex_array()* takes a *Program*, a *Buffer* with input data, and the names of input variables for this program.

VertexArray in ModernGL can be initialized in two ways: one buffer for all input variables or multiple buffers for all input variables.

One input buffer for all input variables (simple *VertexArray* version):

```
vao = ctx.vertex_array(program, buffer, 'input_var1', 'input_var2')
```

Multiple input buffers for all input variables:

```
vao = ctx.vertex_array(
    program,
    [
        (buffer1, '3f 2f', 'input_var1', 'input_var2'),
        (buffer2, '4f', 'input_var3')
    ]
)
```

You can understand `'3f 2f'` and `'4f'` as the type of the input variable (or variables), that is, 3 floats and 2 floats, which form, for example, `vec3 + vec2` and 4 floats, which form `vec4`.

In our example we use a simple implementation of this method.

Entire source

```
import moderngl
import numpy as np

ctx = moderngl.create_context(standalone=True)

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;
```

(continues on next page)

(continued from previous page)

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

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

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

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

Proceed to the *next step*.

Standalone rendering

Standalone (offline) rendering allows you to render without using a window, and is included in ModernGL by default.

Rendering occurs when *VertexArray.render()* is called. By default, the `mode` parameter is *moderngl.TRIANGLES*, but since we need to draw a line, we change the mode value to *moderngl.LINES*:

```
vao.render(moderngl.LINES)  # "mode" is the first optional argument
```

To display the rendering result, we use the Pillow (PIL) library that comes with Python. Let's return the texture from the GPU memory to RAM and call the `PIL.Image.show()` method to show it.

Entire source

```
import moderngl
import numpy as np

from PIL import Image

ctx = moderngl.create_context(standalone=True)

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

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

        void main() {
```

(continues on next page)

(continued from previous page)

```
            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.zeros(50)
g = np.ones(50)
b = np.zeros(50)

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

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

fbo = ctx.framebuffer(
    color_attachments=[ctx.texture((512, 512), 3)]
)
fbo.use()
fbo.clear(0.0, 0.0, 0.0, 1.0)
vao.render(moderngl.LINE_STRIP)

Image.frombytes(
    "RGB", fbo.size, fbo.color_attachments[0].read(),
    "raw", "RGB", 0, -1
).show()
```

The result will be something like this:

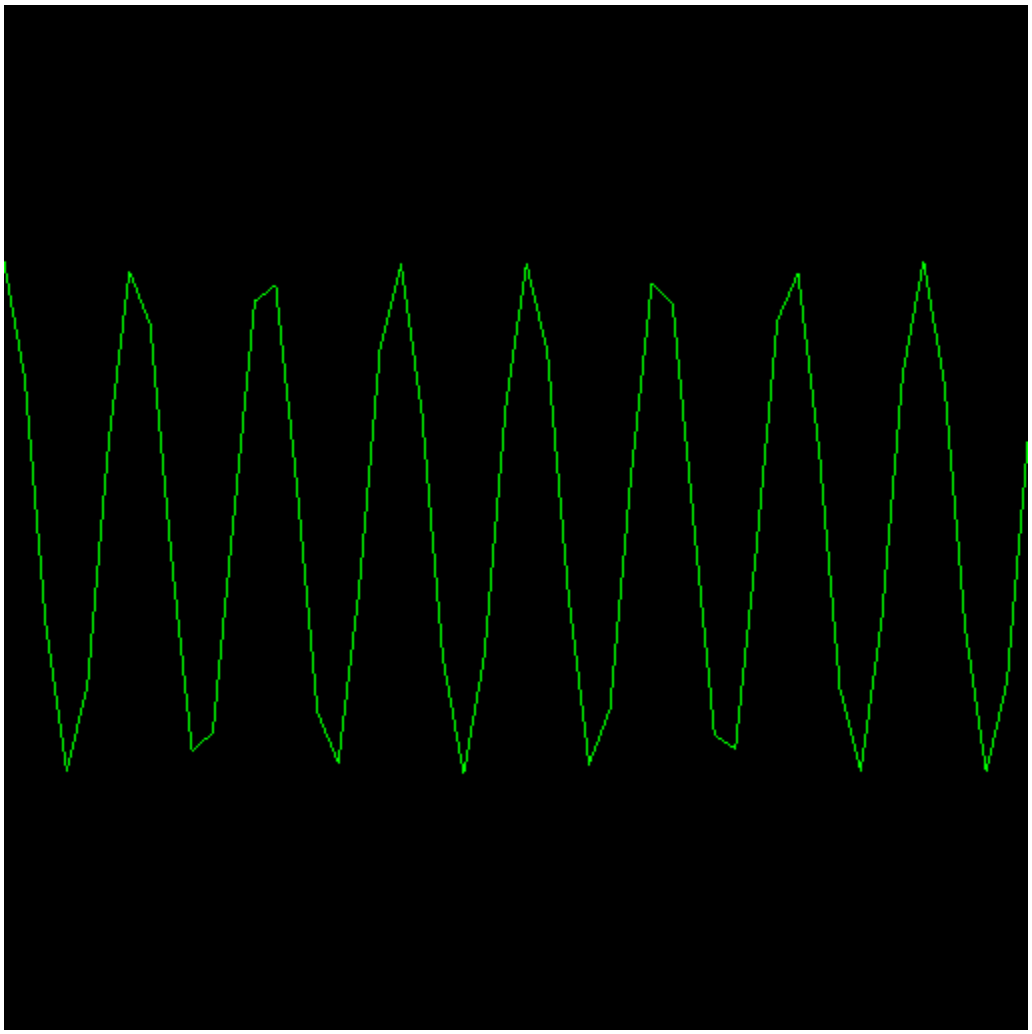


Fig. 2: Rendering result

2.2.3 Triangles drawing

Drawing the first triangles. Closer to real rendering.

One familiar triangle

As with any graphics library guide, we also have a guide on how to draw a triangle. Below is a slightly modified line drawing code from the *previous tutorial*. The following code draws one triangle:

Entire source

```
import moderngl
import numpy as np

from PIL import Image

ctx = moderngl.create_context(standalone=True)

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;
        }
    """,
)

vertices = np.asarray([

    -0.75, -0.75,  1, 0, 0,
    0.75, -0.75,  0, 1, 0,
    0.0, 0.649,  0, 0, 1

```

(continues on next page)

(continued from previous page)

```
], dtype='f4')

vbo = ctx.buffer(vertices.tobytes())
vao = ctx.vertex_array(prog, vbo, "in_vert", "in_color")

fbo = ctx.framebuffer(
    color_attachments=[ctx.texture((512, 512), 3)]
)
fbo.use()
fbo.clear(0.0, 0.0, 0.0, 1.0)
vao.render()  # "mode" is moderngl.TRIANGLES by default

Image.frombytes(
    "RGB", fbo.size, fbo.color_attachments[0].read(),
    "raw", "RGB", 0, -1
).show()
```

When you run the code you will see this:

As you may have noticed, we only specified three colors for each vertex, but OpenGL interpolates our triangle and we see a soft transition of colors.

At this point you can try out the fragment shader, for example, let's draw a lighting effect:

Fragment shader

```
#version 330

in vec3 v_color;

out vec3 f_color;

void main() {
    vec2 pixel_coord = (gl_FragCoord.xy-256.0)/256.0;
    f_color = v_color*(1.0-length(pixel_coord));
}
```

Shaders are not very fond of branching algorithms and operations such as `if (condition) {action} else {action}`. It is recommended to use formulas more often.

Uniform

At some point you will need to make a lot of small changes to your rendering. Changing the screen aspect ratio, viewing angle, changing perspective/orthographic projection and much more. And in many situations it will be very convenient to use *Uniform* -s.

Uniform -s can be specified and used at any time during rendering. They allow you to replace all constants in the shader with variables and change them as needed. The *Uniform* is initialized in the shader as follows:

```
uniform float var1;
```

Changing the *Uniform* -s value in ModernGL is very easy. For example, setting the value for our variable `140.02` is done as follows:

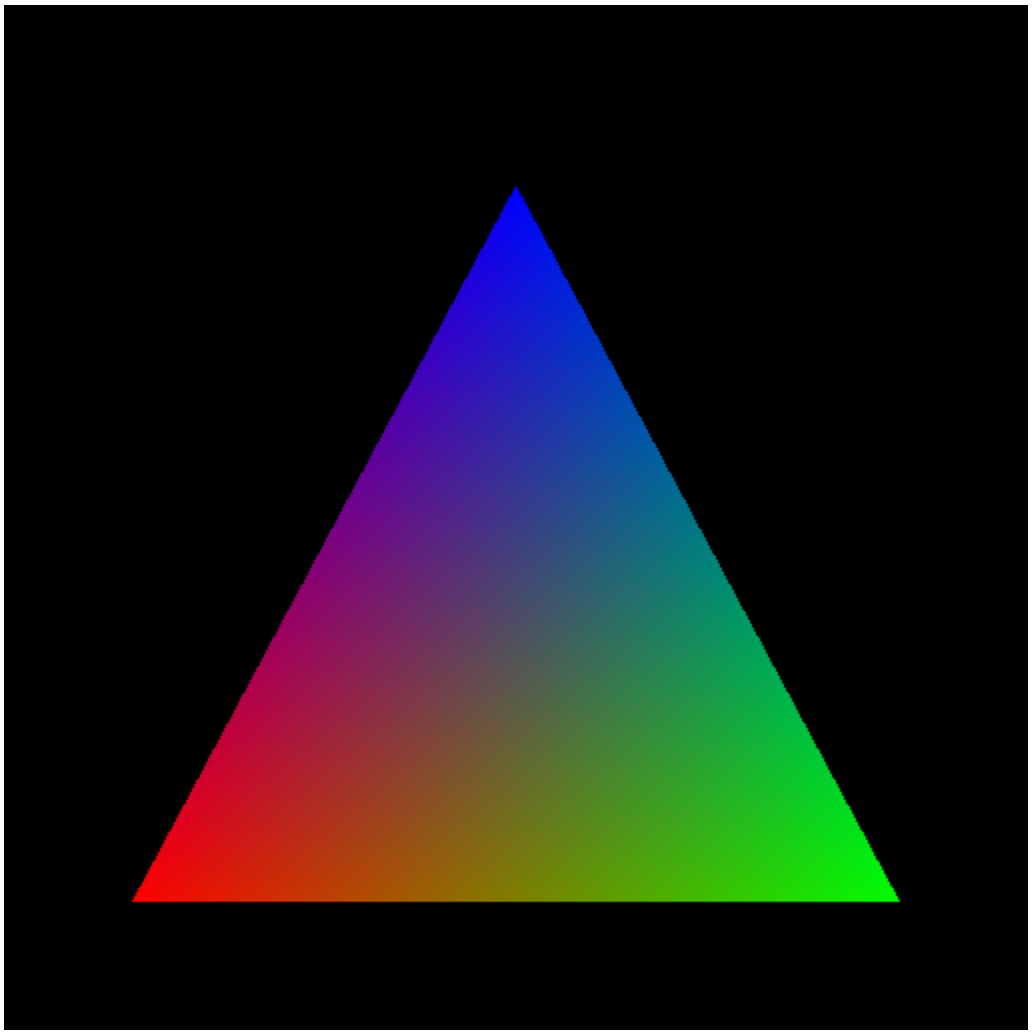


Fig. 3: Triangle

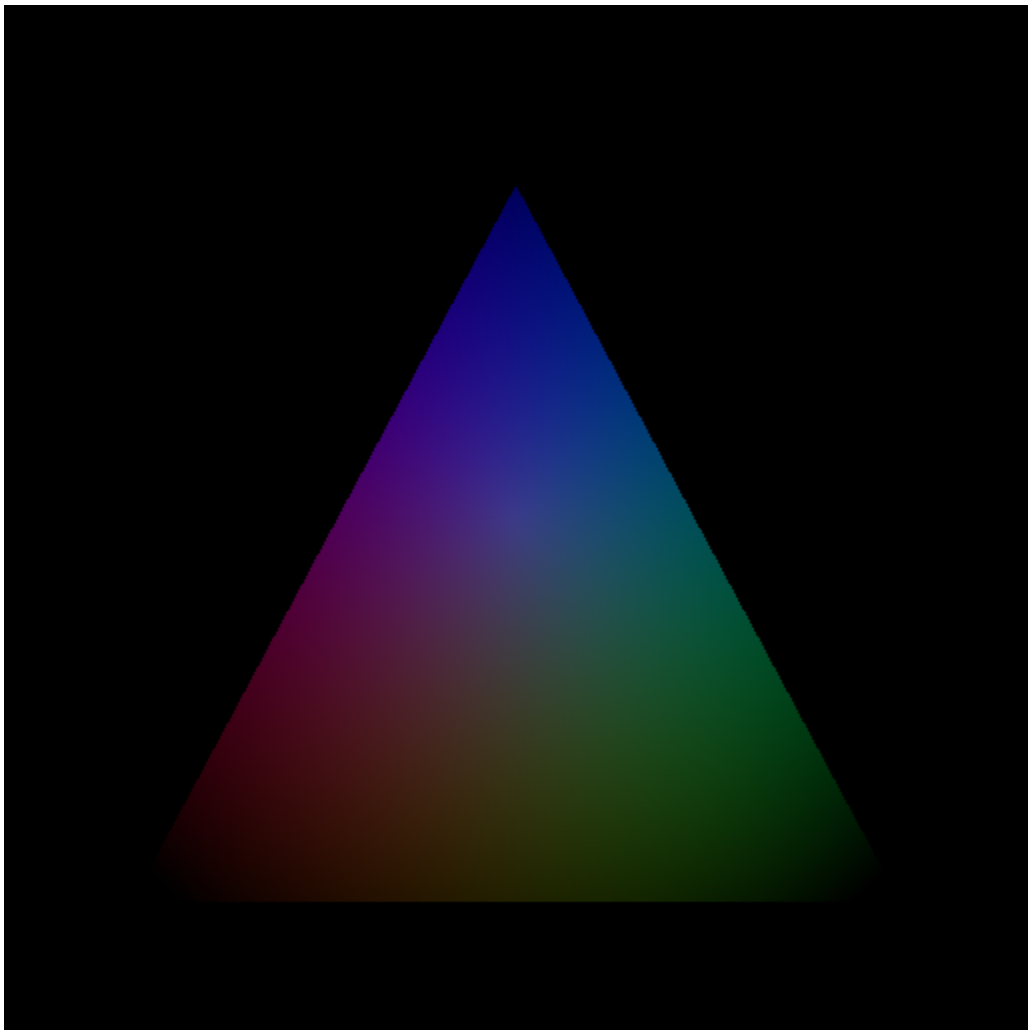


Fig. 4: Lighted triangle

```
vao.program['var1'].value = 140.02

# or (using `__setitem__` shortcut)
vao.program['var1'] = 140.02
```

If the variable type is not `float`, but `vec4`, simply list the values separated by commas:

```
vao.program['var2'] = 1, 2, 3, 4
# or
vao.program['var2'] = [1, 2, 3, 4]
# or
vao.program['var2'] = (1, 2, 3, 4)
```

You need to list as many values as the *value type* takes: `float` will take 1 number, `vec2` will take 2 numbers, `vec4` will take 4 numbers, `mat4` will take 16 numbers, etc.

Let's consider a case where we need to change the size of our triangle. Take the *original triangle drawing code* and make the following changes.

To change the scale (size) of the triangle, add a `scale` *`Uniform`*. In the vertex shader it will be multiplied by all vertices and thus allow us to control the size of all triangles.

Entire source

```
import moderngl
import numpy as np

from PIL import Image

ctx = moderngl.create_context(standalone=True)

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

        in vec2 in_vert;
        in vec3 in_color;

        uniform float scale;

        out vec3 v_color;

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

        in vec3 v_color;

        out vec3 f_color;
```

(continues on next page)

(continued from previous page)

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

vertices = np.asarray([

    -0.75, -0.75,  1, 0, 0,
    0.75, -0.75,  0, 1, 0,
    0.0, 0.649,  0, 0, 1

], dtype='f4')

vbo = ctx.buffer(vertices.tobytes())
vao = ctx.vertex_array(prog, vbo, "in_vert", "in_color")

fbo = ctx.framebuffer(
    color_attachments=[ctx.texture((512, 512), 3)]
)
fbo.use()
fbo.clear(0.0, 0.0, 0.0, 1.0)

vao.program['scale'] = 2

vao.render()  # "mode" is moderngl.TRIANGLES by default

Image.frombytes(
    "RGB", fbo.size, fbo.color_attachments[0].read(),
    "raw", "RGB", 0, -1
).show()
```

We set the scale value to `2.0`, which means our triangle will be enlarged by 2 times.

Now let's set the scale value to `0.5` to reduce the triangle by 2 times:

```
vao.program['scale'] = 0.5
```

Uniforms can not only be set, but also read. This is done as follows:

```
scale = vao.program['scale'].value
```

Also Uniforms can be written or read directly, in the form of bytes:

```
# write
scale = 2
b_scale = numpy.asarray([scale], dtype='f4').tobytes()
vao.program['scale'].write(b_scale)

# read
b_scale = vao.program['scale'].read()
scale = numpy.frombuffer(b_scale, dtype='f4')[0]
```

(continues on next page)

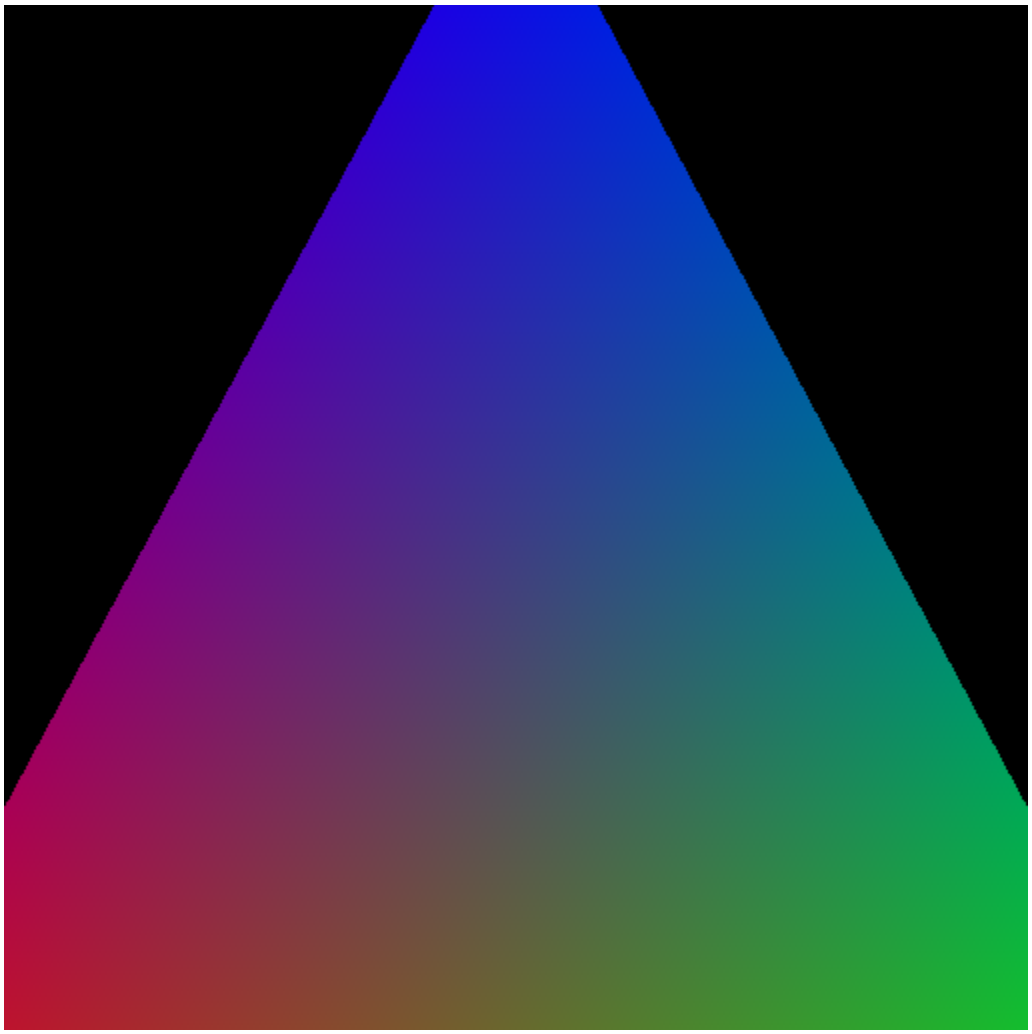


Fig. 5: Enlarged triangle

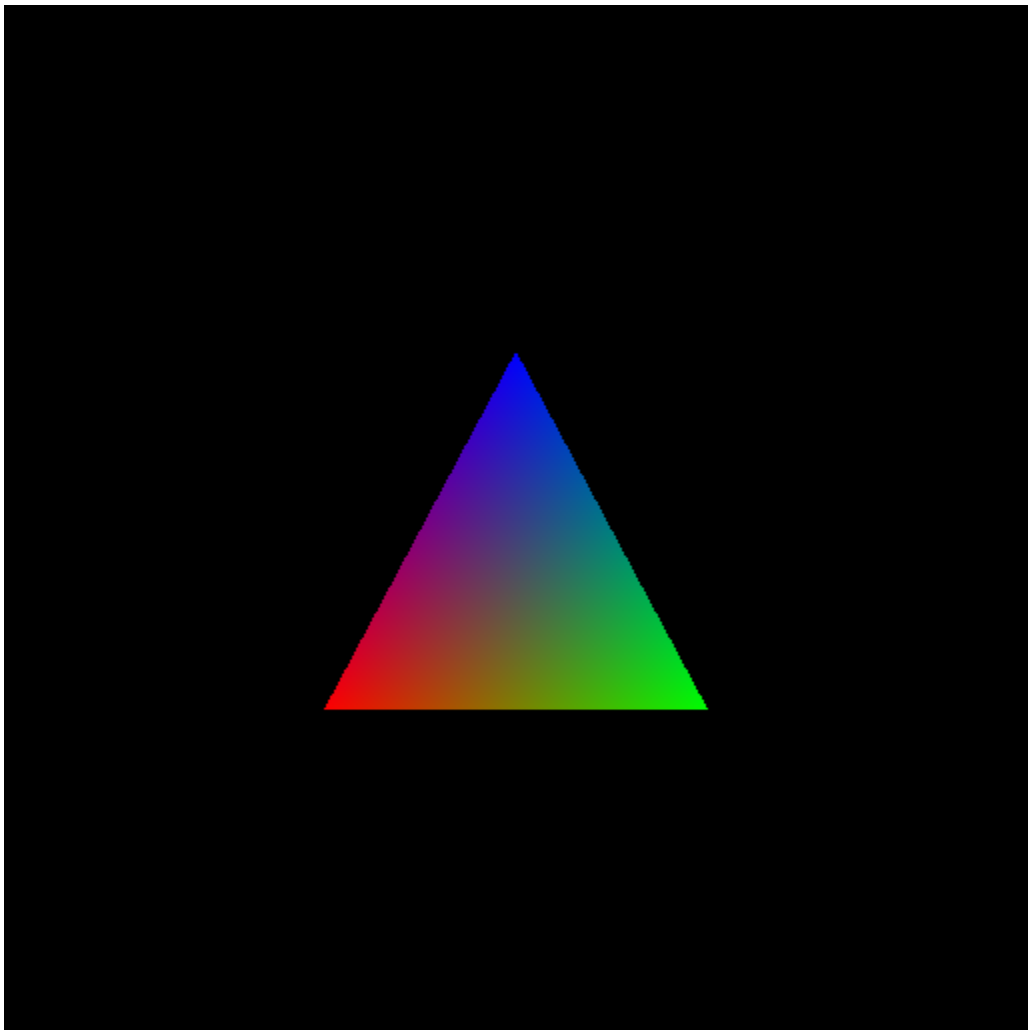


Fig. 6: Reduced triangle

(continued from previous page)

```
# `numpy.frombuffer()` converts a byte string into an array,
# since we have one number, we select it from the array.
```

In most cases, directly using *Uniform* -s `.read()/.write()` methods can speed up the code, but constantly manually converting variables into bytes does not make sense, since ModernGL already does it in the most optimized way.

2.3 Functionality expansion

Opening up new possibilities with ModernGL.

2.3.1 Rendering to a window

By default, ModernGL does not have a window, but the moderngl-window module allows you to use one. The installation is as follows:

```
pip install moderngl-window
```

moderngl-window uses pyglet as its default backend. It is installed automatically along with moderngl-window. However, its use is limited to the supported functionality in `moderngl_window.WindowConfig`.

Entire source

```
import moderngl_window as glw
import numpy as np

window_cls = glw.get_local_window_cls('pyglet')
window = window_cls(
    size=(512, 512), fullscreen=False, title='ModernGL Window',
    resizable=False, vsync=True, gl_version=(3, 3)
)
ctx = window.ctx
glw.activate_context(window, ctx=ctx)
window.clear()
window.swap_buffers()

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);
        }
```

(continues on next page)

(continued from previous page)

```
    """,
    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.zeros(50)
g = np.ones(50)
b = np.zeros(50)

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

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

fbo = ctx.framebuffer(
    color_attachments=[ctx.texture((512, 512), 3)]
)


while not window.is_closing:
    fbo.use()
    fbo.clear(0.0, 0.0, 0.0, 1.0)
    vao.render(gl.LINE_STRIP)

    ctx.copy_framebuffer(window.fbo, fbo)

    window.swap_buffers()
```

You can read the full usage of moderngl-window in its documentation.

CHAPTER

THREE

TOPICS

3.1 The Lifecycle of a ModernGL Object

From moderngl 5.7 we support three different garbage collection modes. This should be configured using the *Context.gc_mode* attribute preferably right after the context is created.

The current supported modes are:

- `None`: (default) No garbage collection is performed. Objects needs to to be manually released like in previous versions of moderngl.
- `"context_gc"`: Dead objects are collected in *Context.objects*. These can periodically be released using *Context.gc()*.
- `"auto"`: Dead objects are destroyed automatically like we would expect in python.

It's important to realize here that garbage collection is not about the python objects itself, but the underlying OpenGL objects. ModernGL operates in many different environments were garbage collection can be a challenge. This depends on factors like who is controlling the existence of the OpenGL context and challenges around threading in python.

3.1.1 Standalone / Headless Context

In this instance we control when the context is created and destroyed. Using `"auto"` garbage collection is perfectly reasonable in this situation.

3.1.2 Context Detection

When detecting an existing context from some window library we have no direct control over the existence of the context. Using `"auto"` mode is dangerous can can cause crashes especially on application exit. The window and context is destroyed and closed, then moderngl will try to destroy resources in a context that no longer exists. Use `"context_gc"` mode to avoid this.

It can be possible to switch the `gc_mode` to `None` when the window is closed. This can still be a problem if you have race conditions due to resources being created in the render loop.

3.1.3 The Threading Issue

When using threads in python the garbage collector can run in any thread. This is a problem for OpenGL because only the main thread is allowed to interact with the context. When using threads in your application you should be using `"context_gc"` mode and periodically call `Context.gc` for example during every frame swap.

3.1.4 Manually Releasing Objects

Objects in moderngl don't automatically release the OpenGL resources when `gc_mode=None` is used. Each type has a `release()` method that needs to be called to properly clean up everything:

```
# Create a texture
texture = ctx.texture((10, 10), 4)

# Properly release the opengl resources
texture.release()
```

3.1.5 Detecting Released Objects

If you for some reason need to detect if a resource was released it can be done by checking the type of the internal moderngl object (`.mglo` property):

```
>> import moderngl
>> ctx = moderngl.create_standalone_context()
>> buffer = ctx.buffer(reserve=1024)
>> type(buffer.mglo)
<class 'mgl.Buffer'>
>> buffer.release()
>> type(buffer.mglo)
<class '_moderngl.InvalidObject'>
>> type(buffer.mglo) == moderngl.mgl.InvalidObject
True
```

3.2 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.

3.2.1 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)

Detecting an existing active context created by a window library:

```
import moderngl
# Create the window with an OpenGL context (Most window libraries support this)
ctx = moderngl.create_context()
# If successful we can now render to the window
print("Default framebuffer is:", ctx.screen)
```

A great reference using various window libraries can be found here: https://github.com/moderngl/moderngl-window/tree/master/moderngl_window/context

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()
```

3.2.2 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).

3.2.3 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.

3.2.4 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
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

3.2.5 Context Info

Various information such as limits and driver information can be found in the `info` property. It can often be useful to know the vendor and render for the context:

```
>>> import moderngl
>>> ctx = moderngl.create_context(standalone=True, gl_version=(4.6))
>>> ctx.info["GL_VENDOR"]
'NVIDIA Corporation'
>>> ctx.info["GL_RENDERER"]
'GeForce RTX 2080 SUPER/PCIe/SSE2'
>>> ctx.info["GL_VERSION"]
'3.3.0 NVIDIA 456.71'
```

Note that it reports version 3.3 here because ModernGL by default requests a version 3.3 context (minimum requirement).

3.3 Texture Format

3.3.1 Description

The format of a texture can be described by the `dtype` parameter during texture creation. For example the `moderngl.Context.texture()`. The default `dtype` is `f1`. Each component is an unsigned byte (0-255) that is normalized when read in a shader into a value from 0.0 to 1.0.

The formats are based on the string formats used in numpy.

Some quick example of texture creation:

```
# RGBA (4 component) f1 texture
texture = ctx.texture((100, 100), 4)  # dtype f1 is default

# R (1 component) f4 texture (32 bit float)
texture = ctx.texture((100, 100), 1, dtype="f4")

# RG (2 component) u2 texture (16 bit unsigned integer)
texture = ctx.texture((100, 100), 2, dtype="u2")
```

Texture contents can be passed in using the `data` parameter during creation or by using the `write()` method. The object passed in `data` can be bytes or any object supporting the buffer protocol.

When writing data to texture the data type can be derived from the internal format in the tables below. `f1` textures takes unsigned bytes (`u1` or `numpy.uint8` in numpy) while `f2` textures takes 16 bit floats (`f2` or `numpy.float16` in numpy).

3.3.2 Float Textures

`f1` textures are just unsigned bytes (8 bits per component) (`GL_UNSIGNED_BYTE`)

The `f1` texture is the most commonly used textures in OpenGL and is currently the default. Each component takes 1 byte (4 bytes for RGBA). This is not really a "real" float format, but a shader will read normalized values from these textures. `0-255` (byte range) is read as a value from `0.0` to `1.0` in shaders.

In shaders the sampler type should be `sampler2D`, `sampler2DArray` `sampler3D`, `samplerCube` etc.

dtype	*Components*	*Base Format*	*Internal Format*
f1	1	GL_RED	GL_R8
f1	2	GL_RG	GL_RG8
f1	3	GL_RGB	GL_RGB8
f1	4	GL_RGBA	GL_RGBA8

`f2` textures stores 16 bit float values (`GL_HALF_FLOAT`).

dtype	*Components*	*Base Format*	*Internal Format*
f2	1	GL_RED	GL_R16F
f2	2	GL_RG	GL_RG16F
f2	3	GL_RGB	GL_RGB16F
f2	4	GL_RGBA	GL_RGBA16F

`f4` textures store 32 bit float values. (`GL_FLOAT`) Note that some drivers do not like 3 components because of alignment.

dtype	*Components*	*Base Format*	*Internal Format*
f4	1	GL_RED	GL_R32F
f4	2	GL_RG	GL_RG32F
f4	3	GL_RGB	GL_RGB32F
f4	4	GL_RGBA	GL_RGBA32F

3.3.3 Integer Textures

Integer textures come in a signed and unsigned version. The advantage with integer textures is that shader can read the raw integer values from them using for example `usampler*` (unsigned) or `isampler*` (signed).

Integer textures do not support `LINEAR` filtering (only `NEAREST`).

Unsigned

`u1` textures store unsigned byte values (`GL_UNSIGNED_BYTE`).

In shaders the sampler type should be `usampler2D`, `usampler2DArray` `usampler3D`, `usamplerCube` etc.

dtype	*Components*	*Base Format*	*Internal Format*
u1	1	GL_RED_INTEGER	GL_R8UI
u1	2	GL_RG_INTEGER	GL_RG8UI
u1	3	GL_RGB_INTEGER	GL_RGB8UI
u1	4	GL_RGBA_INTEGER	GL_RGBA8UI

`u2` textures store 16 bit unsigned integers (`GL_UNSIGNED_SHORT`).

dtype	*Components*	*Base Format*	*Internal Format*
u2	1	GL_RED_INTEGER	GL_R16UI
u2	2	GL_RG_INTEGER	GL_RG16UI
u2	3	GL_RGB_INTEGER	GL_RGB16UI
u2	4	GL_RGBA_INTEGER	GL_RGBA16UI

`u4` textures store 32 bit unsigned integers (`GL_UNSIGNED_INT`)

dtype	*Components*	*Base Format*	*Internal Format*
u4	1	GL_RED_INTEGER	GL_R32UI
u4	2	GL_RG_INTEGER	GL_RG32UI
u4	3	GL_RGB_INTEGER	GL_RGB32UI
u4	4	GL_RGBA_INTEGER	GL_RGBA32UI

Signed

`i1` textures store signed byte values (`GL_BYTE`).

In shaders the sampler type should be `isampler2D`, `isampler2DArray` `isampler3D`, `isamplerCube` etc.

dtype	*Components*	*Base Format*	*Internal Format*
i1	1	GL_RED_INTEGER	GL_R8I
i1	2	GL_RG_INTEGER	GL_RG8I
i1	3	GL_RGB_INTEGER	GL_RGB8I
i1	4	GL_RGBA_INTEGER	GL_RGBA8I

`i2` textures store 16 bit integers (`GL_SHORT`).

dtype	*Components*	*Base Format*	*Internal Format*
i2	1	GL_RED_INTEGER	GL_R16I
i2	2	GL_RG_INTEGER	GL_RG16I
i2	3	GL_RGB_INTEGER	GL_RGB16I
i2	4	GL_RGBA_INTEGER	GL_RGBA16I

`i4` textures store 32 bit integers (`GL_INT`)

dtype	*Components*	*Base Format*	*Internal Format*
i4	1	GL_RED_INTEGER	GL_R32I
i4	2	GL_RG_INTEGER	GL_RG32I
i4	3	GL_RGB_INTEGER	GL_RGB32I
i4	4	GL_RGBA_INTEGER	GL_RGBA32I

3.3.4 Normalized Integer Textures

Normalized integers are integer texture, but texel reads in a shader returns normalized values (`[0.0, 1.0]`). For example an unsigned 16 bit fragment with the value `2**16-1` will be read as `1.0`.

Normalized integer textures should use the *sampler2D* sampler type. Also note that there's no standard for normalized 32 bit integer textures because a float32 doesn't have enough precision to express a 32 bit integer as a number between 0.0 and 1.0.

Unsigned

`nu1` textures is really the same as an `f1`. Each component is a `GL_UNSIGNED_BYTE`, but are read by the shader in normalized form `[0.0, 1.0]`.

dtype	*Components*	*Base Format*	*Internal Format*
nu1	1	GL_RED	GL_R8
nu1	2	GL_RG	GL_RG8
nu1	3	GL_RGB	GL_RGB8
nu1	4	GL_RGBA	GL_RGBA8

`nu2` textures store 16 bit unsigned integers (`GL_UNSIGNED_SHORT`). The value range `[0, 2**16-1]` will be normalized into `[0.0, 1.0]`.

dtype	*Components*	*Base Format*	*Internal Format*
nu2	1	GL_RED	GL_R16
nu2	2	GL_RG	GL_RG16
nu2	3	GL_RGB	GL_RGB16
nu2	4	GL_RGBA	GL_RGBA16

Signed

`ni1` textures store 8 bit signed integers (`GL_BYTE`). The value range `[0, 127]` will be normalized into `[0.0, 1.0]`. Negative values will be clamped.

dtype	*Components*	*Base Format*	*Internal Format*
ni1	1	GL_RED	GL_R8
ni1	2	GL_RG	GL_RG8
ni1	3	GL_RGB	GL_RGB8
ni1	4	GL_RGBA	GL_RGBA8

`ni2` textures store 16 bit signed integers (`GL_SHORT`). The value range `[0, 2**15-1]` will be normalized into `[0.0, 1.0]`. Negative values will be clamped.

dtype	*Components*	*Base Format*	*Internal Format*
ni2	1	GL_RED	GL_R16
ni2	2	GL_RG	GL_RG16
ni2	3	GL_RGB	GL_RGB16
ni2	4	GL_RGBA	GL_RGBA16

3.3.5 Overriding internalformat

`Context.texture()` supports overriding the internalformat of the texture. This is only necessary when needing a different internal formats from the tables above. This can for example be `GL_SRGB8 = 0x8C41` or some compressed format. You may also need to look up in *`Context.extensions`* to ensure the context supports internalformat you are using. We do not provide the enum values for these alternative internalformats. They can be looked up in the registry : https://raw.githubusercontent.com/KhronosGroup/OpenGL-Registry/master/xml/gl.xml

Example:

```
texture = ctx.texture(image.size, 3, data=srbg_data, internal_format=GL_SRGB8)
```

3.4 Buffer Format

3.4.1 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.

3.4.2 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.

3.4.3 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 construct 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"),
```

(continues on next page)

(continued from previous page)

```
    ]
    index_buffer_object
)
```

So, the vertices and normals, using `/v`, are passed to each vertex within an instance. This fulfills the rule that 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.

CHAPTER

FOUR

TECHNIQUES

4.1 Headless on Ubuntu 18 Server

4.1.1 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

4.1.2 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 that you have these libraries installed. ModernGL through the glcontext library will use `ctype.find_library` to locate the latest installed version.

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,
    # These are OPTIONAL if you want to load a specific version
    libgl='libGL.so.1',
    libx11='libX11.so.6',
```

(continues on next page)

(continued from previous page)

```
)

# EGL
import moderngl
ctx = moderngl.create_context(
    standalone=True,
    backend='egl',
    # These are OPTIONAL if you want to load a specific version
    libgl='libGL.so.1',
    libegl='libEGL.so.1',
)
```

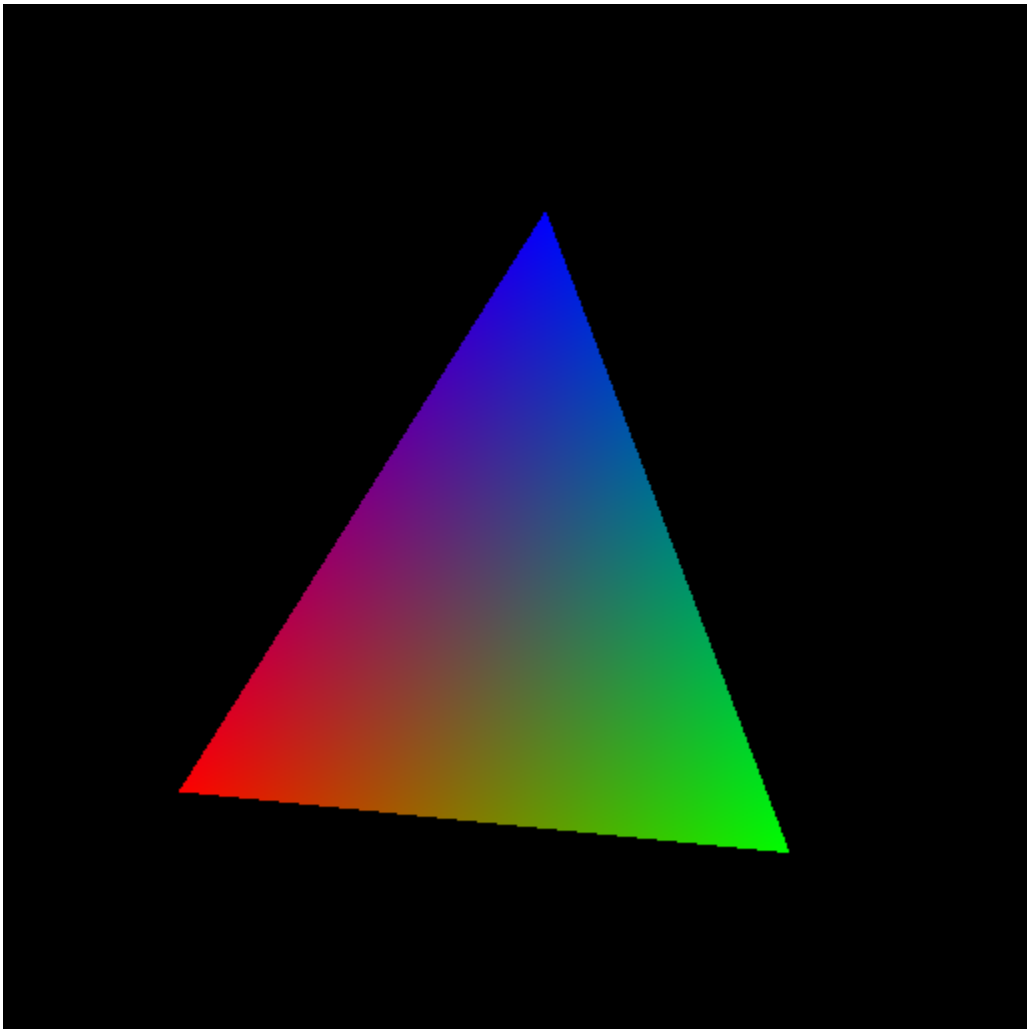
4.1.3 Running an example

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;
    out vec3 color;
    void main() {
        gl_Position = model * vec4(in_vert, 0.0, 1.0);
        color = in_color;
    }
    """,
```

(continues on next page)

(continued from previous page)

```
    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

FIVE

REFERENCE

5.1 moderngl

```
import moderngl

window = ...
ctx = moderngl.create_context()
# store a ref to ctx
```

The module object itself is responsible for creating a *Context* object.

moderngl.**create_context**(*require: int = 330*, *standalone: bool = False*) → *Context*

Create a ModernGL context by loading OpenGL functions from an existing OpenGL context. An OpenGL context must exist. Call this after a window is created or opt for the windowless standalone mode. Other backend specific settings are passed as keyword arguments.

Context sharing is known to not work properly, please avoid using it. There is a paramter *share* for that to attempt to create a shared context.

Parameters

- **require** (*int*) – OpenGL version code
- **standalone** (*bool*) – Headless flag

Example:

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

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

# Create a windowless context
ctx = moderngl.create_context(standalone=True)
```

moderngl.**create_standalone_context**(...) → *Context*

Deprecated, use *moderngl.create_context()* with the standalone parameter set.

moderngl.**get_context**() → *Context*

Returns the previously created context object.

Example:

```
# my_app.py

from moderngl import create_context

ctx = create_context(...)

# my_renderer.py

from moderngl import get_context

class MyRenderer:
    def __init__(self):
        self.ctx = get_context()
        self.program = ...
        self.vao = ...
```

5.1.1 Context Flags

These were moved to *Context*.

moderngl.**NOTHING**
: See *Context.NOTHING*

moderngl.**BLEND**
: See *Context.BLEND*

moderngl.**DEPTH_TEST**
: See *Context.DEPTH_TEST*

moderngl.**CULL_FACE**
: See *Context.CULL_FACE*

moderngl.**RASTERIZER_DISCARD**
: See *Context.RASTERIZER_DISCARD*

moderngl.**PROGRAM_POINT_SIZE**
: See *Context.PROGRAM_POINT_SIZE*

moderngl.**POINTS**
: See *Context.POINTS*

moderngl.**LINES**
: See *Context.LINES*

moderngl.**LINE_LOOP**
: See *Context.LINE_LOOP*

moderngl.**LINE_STRIP**
: See *Context.LINE_STRIP*

moderngl.**TRIANGLES**
: See *Context.TRIANGLES*

moderngl.**TRIANGLE_STRIP**
: See *Context.TRIANGLE_STRIP*

moderngl.**TRIANGLE_FAN**
See *Context.TRIANGLE_FAN*

moderngl.**LINES_ADJACENCY**
See *Context.LINES_ADJACENCY*

moderngl.**LINE_STRIP_ADJACENCY**
See *Context.LINE_STRIP_ADJACENCY*

moderngl.**TRIANGLES_ADJACENCY**
See *Context.TRIANGLES_ADJACENCY*

moderngl.**TRIANGLE_STRIP_ADJACENCY**
See *Context.TRIANGLE_STRIP_ADJACENCY*

moderngl.**PATCHES**
See *Context.PATCHES*

moderngl.**NEAREST**
See *Context.NEAREST*

moderngl.**LINEAR**
See *Context.LINEAR*

moderngl.**NEAREST_MIPMAP_NEAREST**
See *Context.NEAREST_MIPMAP_NEAREST*

moderngl.**LINEAR_MIPMAP_NEAREST**
See *Context.LINEAR_MIPMAP_NEAREST*

moderngl.**NEAREST_MIPMAP_LINEAR**
See *Context.NEAREST_MIPMAP_LINEAR*

moderngl.**LINEAR_MIPMAP_LINEAR**
See *Context.LINEAR_MIPMAP_LINEAR*

moderngl.**ZERO**
See *Context.ZERO*

moderngl.**ONE**
See *Context.ONE*

moderngl.**SRC_COLOR**
See *Context.SRC_COLOR*

moderngl.**ONE_MINUS_SRC_COLOR**
See *Context.ONE_MINUS_SRC_COLOR*

moderngl.**SRC_ALPHA**
See *Context.SRC_ALPHA*

moderngl.**ONE_MINUS_SRC_ALPHA**
See *Context.ONE_MINUS_SRC_ALPHA*

moderngl.**DST_ALPHA**
See *Context.DST_ALPHA*

moderngl.**ONE_MINUS_DST_ALPHA**
See *Context.ONE_MINUS_DST_ALPHA*

moderngl.**DST_COLOR**

See *Context.DST_COLOR*

moderngl.**ONE_MINUS_DST_COLOR**

See *Context.ONE_MINUS_DST_COLOR*

moderngl.**DEFAULT_BLENDING**

See *Context.DEFAULT_BLENDING*

moderngl.**ADDITIVE_BLENDING**

See *Context.ADDITIVE_BLENDING*

moderngl.**PREMULTIPLIED_ALPHA**

See *Context.PREMULTIPLIED_ALPHA*

moderngl.**FUNC_ADD**

See *Context.FUNC_ADD*

moderngl.**FUNC_SUBTRACT**

See *Context.FUNC_SUBTRACT*

moderngl.**FUNC_REVERSE_SUBTRACT**

See *Context.FUNC_REVERSE_SUBTRACT*

moderngl.**MIN**

See *Context.MIN*

moderngl.**MAX**

See *Context.MAX*

moderngl.**FIRST_VERTEX_CONVENTION**

See *Context.FIRST_VERTEX_CONVENTION*

moderngl.**LAST_VERTEX_CONVENTION**

See *Context.LAST_VERTEX_CONVENTION*

5.2 Context

class Context

Returned by `moderngl.create_context()`

Class exposing OpenGL features.

ModernGL objects can be created from this class.

5.2.1 Objects

Context.**program**(*vertex_shader: str*, *fragment_shader: str*, *geometry_shader: str*, *tess_control_shader: str*, *tess_evaluation_shader: str*, *varyings: Tuple[str, ...]*, *fragment_outputs: Dict[str, int]*, *varyings_capture_mode: str = 'interleaved'*) → *Program*

Create a *Program* object.

The `varyings` are only used when a transform program is created to specify the names of the output varyings to capture in the output buffer.

`fragment_outputs` can be used to programmatically map named fragment shader outputs to a framebuffer attachment numbers. This can also be done by using `layout(location=N)` in the fragment shader.

Parameters

- **vertex_shader** (*str*) – The vertex shader source.
- **fragment_shader** (*str*) – The fragment shader source.
- **geometry_shader** (*str*) – The geometry shader source.
- **tess_control_shader** (*str*) – The tessellation control shader source.
- **tess_evaluation_shader** (*str*) – The tessellation evaluation shader source.
- **varyings** (*list*) – A list of varyings.
- **fragment_outputs** (*dict*) – A dictionary of fragment outputs.

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

Returns a new *Buffer* object.

The *data* can be anything supporting the buffer interface.

The *data* and *reserve* parameters are mutually exclusive.

Parameters

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

Context.**vertex_array**(*program: Program*, *content: list*, *index_buffer: Buffer = None*, *index_element_size: int = 4*, *mode: int = ...*) → *VertexArray*

Returns a new *VertexArray* object.

A VertexArray describes how buffers are read by a shader program. The content is a list of tuples containing a buffer, a format string and any number of attribute names. Attribute names are defined by the user in the Vertex Shader program stage.

The default *mode* is *TRIANGLES*.

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 (optional)
- **index_element_size** (*int*) – byte size of each index element, 1, 2 or 4.
- **skip_errors** (*bool*) – Ignore errors during creation
- **mode** (*int*) – The default draw mode (for example: `TRIANGLES`)

Examples:

```
# Empty vertext array (no attribute input)
vao = ctx.vertex_array(program)

# Multiple buffers
vao = ctx.vertex_array(program, [
    (buffer1, '3f', 'in_position'),
```

(continues on next page)

(continued from previous page)

```
    (buffer2, '3f', 'in_normal'),
])
vao = ctx.vertex_array(
    program,
    [
        (buffer1, '3f', 'in_position'),
        (buffer2, '3f', 'in_normal'),
    ],
    index_buffer=ibo,
    index_element_size=2,  # 16 bit / 'u2' index buffer
)
```

Backward Compatible Version:

```
# Simple version with a single buffer
vao = ctx.vertex_array(program, buffer, 'in_position', 'in_normal')
vao = ctx.vertex_array(program, buffer, 'in_position', 'in_normal', index_
↪buffer=ibo)
```

Context.**simple_vertex_array**(...)

Deprecated, use *Context.vertex_array()* instead.

Context.**texture**(*size: Tuple[int, int]*, *components: int*, *data: Any = None*, *samples: int = 0*, *alignment: int = 1*, *dtype: str = 'f1'*) → *Texture*

Returns a new *Texture* object.

A Texture is a 2D image that can be used for sampler2D uniforms or as render targets if framebuffers.

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.
- **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.
- **internal_format** (*int*) – Override the internalformat of the texture (IF needed)

Example:

```
from PIL import Image

img = Image.open(...).convert('RGBA')
texture = ctx.texture(img.size, components=4, data=img.tobytes())

# float texture
texture = ctx.texture((64, 64), components=..., dtype='f4')

# integer texture
texture = ctx.texture((64, 64), components=..., dtype='i4')
```

Note: Do not play with `internal_format` unless you know exactly you are doing. This is an override to support sRGB and compressed textures if needed.

`Context.framebuffer`(*color_attachments: List[*Texture*]*, *depth_attachment:* Texture = *None*) → *Framebuffer*

Returns a new *Framebuffer* object.

A Framebuffer is a collection of images that can be used as render targets. The images of the Framebuffer object can be either Textures or Renderbuffers.

Parameters

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

`Context.sampler`(*repeat_x: bool*, *repeat_y: bool*, *repeat_z: bool*, *filter: tuple*, *anisotropy: float*, *compare_func: str*, *border_color: tuple*, *min_lod: float*, *max_lod: float*, *texture:* Texture) → *Sampler*

Returns a new *Sampler* object.

Samplers bind Textures to uniform samplers within a Program object. Binding a Sampler object also binds the texture object attached to it.

Parameters

- **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** (*str*) – 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.depth_texture`(*size: Tuple[int, int]*, *data: Any* = *None*, *samples: int* = *0*, *alignment: int* = *4*) → *Texture*

Returns a new *Texture* object.

A depth texture can be used for sampler2D and sampler2DShadow uniforms and as a depth attachment for framebuffers.

Parameters

- **size** (*tuple*) – The width and height of the texture.
- **data** (*bytes*) – Content of the texture.
- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.

Context.**texture3d**(*size: Tuple[int, int, int]*, *components: int*, *data: Any = None*, *alignment: int = 1*, *dtype: str = 'f1'*) → *Texture3D*

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

Context.**texture_array**(*size: Tuple[int, int, int]*, *components: int*, *data: Any = None*, ***, *alignment: int = 1*, *dtype: str = 'f1'*) → *TextureArray*

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

Context.**texture_cube**(*size: Tuple[int, int]*, *components: int*, *data: Any = None*, *alignment: int = 1*, *dtype: str = 'f1'*) → *TextureCube*

Returns a new *TextureCube* object.

Note that the width and height of the cubemap must be the same.

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
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.
- **dtype** (*str*) – Data type.
- **internal_format** (*int*) – Override the internalformat of the texture (IF needed)

Context.**depth_texture_cube**(*size: Tuple[int, int]*, *data: Any | None = None*, *alignment: int = 4*) → *TextureCube*

Returns a new *TextureCube* object.

Parameters

- **size** (*tuple*) – The width and height of the texture.
- **data** (*bytes*) – Content of the texture.
- **alignment** (*int*) – The byte alignment 1, 2, 4 or 8.

Context.**simple_framebuffer**(...)

Deprecated, use *Context.framebuffer()* instead.

Context.**renderbuffer**(*size: Tuple[int, int]*, *components: int = 4*, *samples: int = 0*, *dtype: str = 'f1'*) → *Renderbuffer*

Returns a new *Renderbuffer* object.

Similar to textures, renderbuffers can be attached to framebuffers as render targets, but they cannot be sampled as textures.

Parameters

- **size** (*tuple*) – The width and height of the renderbuffer.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

Context.**depth_renderbuffer**(*size: Tuple[int, int]*, *samples: int = 0*) → *Renderbuffer*

Returns a new *Renderbuffer* object.

Parameters

- **size** (*tuple*) – The width and height of the renderbuffer.
- **samples** (*int*) – The number of samples. Value 0 means no multisample format.

Context.**scope**(*framebuffer*, *enable_only*, *textures*, *uniform_buffers*, *storage_buffers*, *samplers*)

Returns a new *Scope* object.

Scope objects can be attached to VertexArray objects to minimize the possibility of rendering within the wrong scope. VertexArrays with an attached scope always have the scope settings at render time.

Parameters

- **framebuffer** (*Framebuffer*) – The framebuffer to use when entering.
- **enable_only** (*int*) – The enable_only flags to set when entering.
- **textures** (*tuple*) – List of (texture, binding) tuples.
- **uniform_buffers** (*tuple*) – Tuple of (buffer, binding) tuples.
- **storage_buffers** (*tuple*) – Tuple of (buffer, binding) tuples.
- **samplers** (*tuple*) – Tuple of sampler bindings

Context.**query**(*samples: bool*, *any_samples: bool*, *time: bool*, *primitives: bool*) → *Query*

Returns a new *Query* object.

Parameters

- **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**(...)

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.

5.2.2 External Objects

External objects are only useful for interoperability with other libraries.

`Context.external_buffer`(*glo: int*, *size: int*, *dynamic: bool*) → *Buffer*
TBD

`Context.external_texture`(*glo: int*, *size: Tuple[int, int]*, *components: int*, *samples: int*, *dtype: str*) → *Texture*
Returns a new *Texture* object from an existing OpenGL texture object.

The content of the texture is referenced and it is not copied.

Parameters

- **glo** (*int*) – External OpenGL texture object.
- **size** (*tuple*) – The width and height of the texture.
- **components** (*int*) – The number of components 1, 2, 3 or 4.
- **samples** (*int*) – The number of samples. Value 0 means no multisample format.
- **dtype** (*str*) – Data type.

5.2.3 Methods

`Context.clear`()
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.

Args:
red (float): color component. green (float): color component. blue (float): color component. alpha (float): alpha component. depth (float): depth value.

Keyword Args:
viewport (tuple): The viewport. color (tuple): Optional rgba color tuple

`Context.enable_only`()
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)
```

Args:
flags (EnableFlag): The flags to enable

`Context.enable()`

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)
```

Args:
flag (int): The flags to enable.

`Context.disable()`

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)
```

Args:
flag (int): The flags to disable.

`Context.enable_direct()`

Gives direct access to `glEnable` so unsupported capabilities in ModernGL can be enabled.

Do not use this to set already supported context flags.

Example:

```
# Enum value from the opengl registry
GL_CONSERVATIVE_RASTERIZATION_NV = 0x9346
ctx.enable_direct(GL_CONSERVATIVE_RASTERIZATION_NV)
```

Context.**disable_direct**()

Gives direct access to `glDisable` so unsupported capabilities in ModernGL can be disabled.

Do not use this to set already supported context flags.

Example:

```
# Enum value from the opengl registry
GL_CONSERVATIVE_RASTERIZATION_NV = 0x9346
ctx.disable_direct(GL_CONSERVATIVE_RASTERIZATION_NV)
```

Context.**finish**()

Wait for all drawing commands to finish.

Context.**clear_samplers**()

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.

Parameters

- **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.**copy_buffer**()

Copy buffer content.

Args:
dst (Buffer): The destination buffer. src (Buffer): The source buffer. size (int): The number of bytes to copy.

Keyword Args:
read_offset (int): The read offset. write_offset (int): The write offset.

Context.**copy_framebuffer**()

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.

Args:
dst (Framebuffer or Texture): Destination framebuffer or texture. src (Framebuffer): Source framebuffer.

`Context.detect_framebuffer()`

Detect a framebuffer.

This is already done when creating a context, but if the underlying window library for some changes the default framebuffer during the lifetime of the application this might be necessary.

Args:
glo (int): Framebuffer object.

`Context.memory_barrier()`

Applying a memory barrier.

The memory barrier is needed in particular to correctly change buffers or textures between each shader. If the same buffer is changed in two shaders, it can cause an effect like 'depth fighting' on a buffer or texture.

The method should be used between *Program* -s, between *ComputeShader* -s, and between *Program* -s and *ComputeShader* -s.

Keyword Args:
barriers (int): Affected barriers, default moderngl.ALL_BARRIER_BITS. by_region (bool): Memory barrier mode by region. More read on https://registry.khronos.org/OpenGL-Refpages/gl4/html/glMemoryBarrier.xhtml

`Context.gc()` → int

Deletes OpenGL objects. Returns the number of objects deleted.

This method must be called to garbage collect OpenGL resources when `gc_mode` is `'context_gc'`.

Calling this method with any other `gc_mode` configuration has no effect and is perfectly safe.

`Context.release()`

5.2.4 Attributes

`Context.gc_mode: str`

The garbage collection mode.

The default mode is `None` meaning no automatic garbage collection is done. Other modes are `auto` and `context_gc`. Please see documentation for the appropriate configuration.

Examples:

```
# Disable automatic garbage collection.
# Each objects needs to be explicitly released.
ctx.gc_mode = None

# Collect all dead objects in the context and
# release them by calling Context.gc()
ctx.gc_mode = 'context_gc'
ctx.gc()  # Deletes the collected objects

# Enable automatic garbage collection like
# we normally expect in python.
ctx.gc_mode = 'auto'
```

Context.**objects: deque**

Moderngl objects scheduled for deletion.

These are deleted when calling *Context.gc()*.

Context.**line_width: float**

Set the default line width.

> **Warning:** A line width other than 1.0 is not guaranteed to work across different OpenGL implementations. For wide lines you should be using geometry shaders.

Context.**point_size: float**

Set/get the point size.

Point size changes the pixel size of rendered points. The min and max values are limited by `POINT_SIZE_RANGE`. This value usually at least `(1, 100)`, but this depends on the drivers/vendors.

If variable point size is needed you can enable `PROGRAM_POINT_SIZE` and write to `gl_PointSize` in the vertex or geometry shader.

Note: Using a geometry shader to create triangle strips from points is often a safer way to render large points since you don't have have any size restrictions.

Context.**depth_func: str**

Set the default depth func.

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
```

Context.**depth_clamp_range: Tuple[float, float]**

Setting up depth clamp range (write only, by default `None`).

`ctx.depth_clamp_range` offers uniform use of GL_DEPTH_CLAMP and glDepthRange.

`GL_DEPTH_CLAMP` is needed to disable clipping of fragments outside near limit of projection matrix. For example, this will allow you to draw between 0 and 1 in the Z (depth) coordinate, even if `near` is set to 0.5 in the projection matrix.

Note: All fragments outside the `near` of the projection matrix will have a depth of `near`.

See https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Depth_clamping for more info.

`glDepthRange(nearVal, farVal)` is needed to specify mapping of depth values from normalized device coordinates to window coordinates. See https://registry.khronos.org/OpenGL-Refpages/gl4/html/glDepthRange.xhtml for more info.

Example:

```
# For glDisable(GL_DEPTH_CLAMP) and glDepthRange(0, 1)
ctx.depth_clamp_range = None

# For glEnable(GL_DEPTH_CLAMP) and glDepthRange(near, far)
ctx.depth_clamp_range = (near, far)
```

Context.**blend_func: tuple**

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

# Shortcuts
moderngl.DEFAULT_BLENDING     # (SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
moderngl.ADDITIVE_BLENDING    # (ONE, ONE)
moderngl.PREMULTIPLIED_ALPHA  # (SRC_ALPHA, ONE)
```

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
)
```

Context.**blend_equation: tuple**

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

Example:

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

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

Context.**multisample: bool**

Enable/disable multisample mode (GL_MULTISAMPLE).

This property is write only.

Example:

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

Context.**viewport: tuple**

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.

Context.**scissor: tuple**

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.

Context.**version_code: int**

Context.**screen: *Framebuffer***

A Framebuffer instance representing the screen.

Normally 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 since there are no default framebuffer.

Context.**fbo:** *Framebuffer*

Context.**front_face:** str

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'
```

Context.**cull_face:** str

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'
```

Context.**wireframe:** bool

Wireframe settings for debugging.

Context.**max_samples:** int

The maximum supported number of samples for multisampling.

Context.**max_integer_samples:** int

The max integer samples.

Context.**max_texture_units:** int

The max texture units.

Context.**max_anisotropy:** float

The maximum value supported for anisotropic filtering.

Context.**default_texture_unit:** int

The default texture unit.

Context.**patch_vertices:** int

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

Context.**provoking_vertex:** int

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
```

Context.**polygon_offset: tuple**

Get or set the current polygon offset.

The tuple values represents two float values: `unit` and a `factor`:

```
ctx.polygon_offset = unit, factor
```

When drawing polygons, lines or points directly on top of exiting geometry the result is often not visually pleasant. We can experience z-fighting or partially fading fragments due to different primitives not being rasterized in the exact same way or simply depth buffer precision issues.

For example when visualizing polygons drawing a wireframe version on top of the original mesh, these issues are immediately apparent. Applying decals to surfaces is another common example.

The official documentation states the following:

```
When enabled, the depth value of each fragment is added
to a calculated offset value. The offset is added before
the depth test is performed and before the depth value
is written into the depth buffer. The offset value o is calculated by:
o = m * factor + r * units
where m is the maximum depth slope of the polygon and r is the smallest
value guaranteed to produce a resolvable difference in window coordinate
depth values. The value r is an implementation-specific int.
```

In simpler terms: We use polygon offset to either add a positive offset to the geometry (push it away from you) or a negative offset to geometry (pull it towards you).

- **`units` is a int offset to depth and will do the job alone**
  if we are working with geometry parallel to the near/far plane.
- The `factor` helps you handle sloped geometry (not parallel to near/far plane).

In most cases you can get away with `[-1.0, 1.0]` for both factor and units, but definitely play around with the values. When both values are set to `0` polygon offset is disabled internally.

To just get started with something you can try:

```
# Either push the geomtry away or pull it towards you
# with support for handling small to medium sloped geometry
ctx.polygon_offset = 1.0, 1.0
ctx.polygon_offset = -1.0, -1.0

# Disable polygon offset
ctx.polygon_offset = 0, 0
```

Context.**error: str**

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.

Context.**extensions: Set[str]**

The extensions supported by the context.

All extensions names have a `GL_` prefix, so if the spec refers to `ARB_compute_shader` we need to look for `GL_ARB_compute_shader`:

```
# If compute shaders are supported ...
>> 'GL_ARB_compute_shader' in ctx.extensions
True
```

Example data:

```
{
    'GL_ARB_multi_bind',
    'GL_ARB_shader_objects',
    'GL_ARB_half_float_vertex',
    'GL_ARB_map_buffer_alignment',
    'GL_ARB_arrays_of_arrays',
    'GL_ARB_pipeline_statistics_query',
    'GL_ARB_provoking_vertex',
    'GL_ARB_gpu_shader5',
    'GL_ARB_uniform_buffer_object',
    'GL_EXT_blend_equation_separate',
    'GL_ARB_tessellation_shader',
    'GL_ARB_multi_draw_indirect',
    'GL_ARB_multisample',
    .. etc ..
}
```

Context.**info: Dict[str, Any]**

OpenGL Limits and information about the context.

Example:

```
# The maximum width and height of a texture
>> ctx.info['GL_MAX_TEXTURE_SIZE']
16384

# Vendor and renderer
>> ctx.info['GL_VENDOR']
NVIDIA Corporation
>> ctx.info['GL_RENDERER']
NVIDIA GeForce GT 650M OpenGL Engine
```

Example data:

```
{
    '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),
    'GL_ALIASED_LINE_WIDTH_RANGE': (1.0, 1.0),
    'GL_POINT_FADE_THRESHOLD_SIZE': 1.0,
    'GL_POINT_SIZE_GRANULARITY': 0.125,
```

(continues on next page)

(continued from previous page)

```
    '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,
```

(continues on next page)

(continued from previous page)

```
    'GL_MAX_VERTEX_ATTRIBS': 16,
    'GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS': 16,
    '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
}
```

Context.**includes: Dict[str, str]**

Mapping used for include statements.

Context.**extra: Any**

User defined data.

5.2.5 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: int**

Represents no states. Can be used with *Context.enable_only()* to disable all states.

Context.**BLEND: int**

Enable/disable blending

Context.**DEPTH_TEST: int**

Enable/disable depth testing

Context.**CULL_FACE: int**

Enable/disable face culling

Context.**RASTERIZER_DISCARD: int**

Enable/disable rasterization

Context flag: Enables `gl_PointSize` in vertex or geometry shaders.

When enabled we can write to `gl_PointSize` in the vertex shader to specify the point size for each individual point.

If this value is not set in the shader the behavior is undefined. This means the points may or may not appear depending if the drivers enforce some default value for `gl_PointSize`.

`Context.PROGRAM_POINT_SIZE: int`

When disabled *Context.point_size* is used.

5.2.6 Primitive Modes

`Context.POINTS: int`

Each vertex represents a point

`Context.LINES: int`

Vertices 0 and 1 are considered a line. Vertices 2 and 3 are considered a line. And so on. If the user specifies a non-even number of vertices, then the extra vertex is ignored.

`Context.LINE_LOOP: int`

As line strips, except that the first and last vertices are also used as a line. Thus, you get n lines for n input vertices. If the user only specifies 1 vertex, the drawing command is ignored. The line between the first and last vertices happens after all of the previous lines in the sequence.

`Context.LINE_STRIP: int`

The adjacent vertices are considered lines. Thus, if you pass n vertices, you will get n-1 lines. If the user only specifies 1 vertex, the drawing command is ignored.

`Context.TRIANGLES: int`

Vertices 0, 1, and 2 form a triangle. Vertices 3, 4, and 5 form a triangle. And so on.

`Context.TRIANGLE_STRIP: int`

Every group of 3 adjacent vertices forms a triangle. The face direction of the strip is determined by the winding of the first triangle. Each successive triangle will have its effective face order reversed, so the system compensates for that by testing it in the opposite way. A vertex stream of n length will generate n-2 triangles.

`Context.TRIANGLE_FAN: int`

The first vertex is always held fixed. From there on, every group of 2 adjacent vertices form a triangle with the first. So with a vertex stream, you get a list of triangles like so: (0, 1, 2) (0, 2, 3), (0, 3, 4), etc. A vertex stream of n length will generate n-2 triangles.

`Context.LINES_ADJACENCY: int`

These are special primitives that are expected to be used specifically with geomtry shaders. These primitives give the geometry shader more vertices to work with for each input primitive. Data needs to be duplicated in buffers.

`Context.LINE_STRIP_ADJACENCY: int`

These are special primitives that are expected to be used specifically with geomtry shaders. These primitives give the geometry shader more vertices to work with for each input primitive. Data needs to be duplicated in buffers.

`Context.TRIANGLES_ADJACENCY: int`

These are special primitives that are expected to be used specifically with geomtry shaders. These primitives give the geometry shader more vertices to work with for each input primitive. Data needs to be duplicated in buffers.

`Context.TRIANGLE_STRIP_ADJACENCY: int`

These are special primitives that are expected to be used specifically with geomtry shaders. These primitives give the geometry shader more vertices to work with for each input primitive. Data needs to be duplicated in buffers.

Context.**PATCHES: int**

primitive type can only be used when Tessellation is active. It is a primitive with a user-defined number of vertices, which is then tessellated based on the control and evaluation shaders into regular points, lines, or triangles, depending on the TES's settings.

Texture Filters

Also available in the *Context* instance including mode details.

Context.**NEAREST: int**

Returns the value of the texture element that is nearest (in Manhattan distance) to the specified texture coordinates.

Context.**LINEAR: int**

Returns the weighted average of the four texture elements that are closest to the specified texture coordinates. These can include items wrapped or repeated from other parts of a texture, depending on the values of texture repeat mode, and on the exact mapping.

Context.**NEAREST_MIPMAP_NEAREST: int**

Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `NEAREST` criterion (the texture element closest to the specified texture coordinates) to produce a texture value.

Context.**LINEAR_MIPMAP_NEAREST: int**

Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `LINEAR` criterion (a weighted average of the four texture elements that are closest to the specified texture coordinates) to produce a texture value.

Context.**NEAREST_MIPMAP_LINEAR: int**

Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `NEAREST` criterion (the texture element closest to the specified texture coordinates) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

Context.**LINEAR_MIPMAP_LINEAR: int**

Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `LINEAR` criterion (a weighted average of the texture elements that are closest to the specified texture coordinates) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

5.2.7 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: int**

(0,0,0,0)

Context.**ONE: int**

(1,1,1,1)

Context.**SRC_COLOR: int**

(Rs0/kR,Gs0/kG,Bs0/kB,As0/kA)

Context.**ONE_MINUS_SRC_COLOR: int**

(1,1,1,1) - (Rs0/kR,Gs0/kG,Bs0/kB,As0/kA)

Context.**SRC_ALPHA: int**
(As0/kA,As0/kA,As0/kA,As0/kA)

Context.**ONE_MINUS_SRC_ALPHA: int**
(1,1,1,1) - (As0/kA,As0/kA,As0/kA,As0/kA)

Context.**DST_ALPHA: int**
(Ad/kA,Ad/kA,Ad/kA,Ad/kA)

Context.**ONE_MINUS_DST_ALPHA: int**
(1,1,1,1) - (Ad/kA,Ad/kA,Ad/kA,Ad/kA)

Context.**DST_COLOR: int**
(Rd/kR,Gd/kG,Bd/kB,Ad/kA)

Context.**ONE_MINUS_DST_COLOR: int**
(1,1,1,1) - (Rd/kR,Gd/kG,Bd/kB,Ad/kA)

5.2.8 Blend Function Shortcuts

Context.**DEFAULT_BLENDING: tuple**
Shotcut for the default blending `SRC_ALPHA, ONE_MINUS_SRC_ALPHA`

Context.**ADDITIVE_BLENDING: tuple**
Shotcut for additive blending `ONE, ONE`

Context.**PREMULTIPLIED_ALPHA: tuple**
Shotcut for blend mode when using premultiplied alpha `SRC_ALPHA, ONE`

5.2.9 Blend Equations

Used with *Context.blend_equation*.

Context.**FUNC_ADD: int**
source + destination

Context.**FUNC_SUBTRACT: int**
source - destination

Context.**FUNC_REVERSE_SUBTRACT: int**
destination - source

Context.**MIN: int**
Minimum of source and destination

Context.**MAX: int**
Maximum of source and destination

5.2.10 Other Enums

Context.**FIRST_VERTEX_CONVENTION: int**

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: int**

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

Context.**VERTEX_ATTRIB_ARRAY_BARRIER_BIT: int**

VERTEX_ATTRIB_ARRAY_BARRIER_BIT

Context.**ELEMENT_ARRAY_BARRIER_BIT: int**

ELEMENT_ARRAY_BARRIER_BIT

Context.**UNIFORM_BARRIER_BIT: int**

UNIFORM_BARRIER_BIT

Context.**TEXTURE_FETCH_BARRIER_BIT: int**

TEXTURE_FETCH_BARRIER_BIT

Context.**SHADER_IMAGE_ACCESS_BARRIER_BIT: int**

SHADER_IMAGE_ACCESS_BARRIER_BIT

Context.**COMMAND_BARRIER_BIT: int**

COMMAND_BARRIER_BIT

Context.**PIXEL_BUFFER_BARRIER_BIT: int**

PIXEL_BUFFER_BARRIER_BIT

Context.**TEXTURE_UPDATE_BARRIER_BIT: int**

TEXTURE_UPDATE_BARRIER_BIT

Context.**BUFFER_UPDATE_BARRIER_BIT: int**

BUFFER_UPDATE_BARRIER_BIT

Context.**FRAMEBUFFER_BARRIER_BIT: int**

FRAMEBUFFER_BARRIER_BIT

Context.**TRANSFORM_FEEDBACK_BARRIER_BIT: int**

TRANSFORM_FEEDBACK_BARRIER_BIT

Context.**ATOMIC_COUNTER_BARRIER_BIT: int**

ATOMIC_COUNTER_BARRIER_BIT

Context.**SHADER_STORAGE_BARRIER_BIT: int**

SHADER_STORAGE_BARRIER_BIT

Context.**ALL_BARRIER_BITS: int**

ALL_BARRIER_BITS

5.2.11 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)
```

5.3 Buffer

class Buffer

Returned by *Context.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()*.

5.3.1 Methods

Buffer.**write**(*data: Any*, *, *offset: int = 0*) → None:

Write the content.

Parameters

- **data** (*bytes*) – The data.
- **offset** (*int*) – The offset in bytes.

Buffer.**read**(*size: int = -1*, *, *offset: int = 0*) → bytes:

Read the content.

Parameters

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

Buffer.**read_into**(*buffer: Any*, *size: int = -1*, *, *offset: int = 0*, *write_offset: int = 0*) → None:

Read the content into a buffer.

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the content.
- **size** (*int*) – The size in bytes. Value -1 means all.
- **offset** (*int*) – The read offset in bytes.
- **write_offset** (*int*) – The write offset in bytes.

Buffer.**clear**(*size: int = -1*, *, *offset: int = 0*, *chunk: Any = None*) → None:

Clear the content.

Parameters

- **size** (*int*) – The size. Value -1 means all.
- **offset** (*int*) – The offset.
- **chunk** (*bytes*) – The chunk to use repeatedly.

Buffer.**bind_to_uniform_block**(*binding: int = 0*, *, *offset: int = 0*, *size: int = -1*) → None:

Bind the buffer to a uniform block.

Parameters

- **binding** (*int*) – The uniform block binding.
- **offset** (*int*) – The offset.
- **size** (*int*) – The size. Value -1 means all.

Buffer.**bind_to_storage_buffer**(*binding: int = 0*, *, *offset: int = 0*, *size: int = -1*) → None:

Bind the buffer to a shader storage buffer.

Parameters

- **binding** (*int*) – The shader storage binding.
- **offset** (*int*) – The offset.
- **size** (*int*) – The size. Value -1 means all.

Buffer.**release**() → None:

Release the ModernGL object

Buffer.**bind**(**attribs*, *layout=None*) → tuple:

Helper method for binding a buffer in *Context.vertex_array()*.

Buffer.**assign**(*index: int*) → tuple:

Helper method for assigning a buffer to an index in *Context.scope()*.

5.3.2 Attributes

Buffer.**size: int**

The size of the buffer in bytes.

Buffer.**dynamic: bool**

The dynamic flag.

Buffer.**ctx: *Context***

The context this object belongs to

Buffer.**glo**: **int**

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

Buffer.**extra**: **Any**

User defined data.

5.4 VertexArray

class VertexArray

Returned by *Context.vertex_array()*

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.

Compared to OpenGL, *VertexArray* also stores a *Program* object.

5.4.1 Methods

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

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

VertexArray.**render_indirect**(*buffer: Buffer*, *mode: int | None = None*, *count: int = -1*, *first: int = 0*) → None

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

VertexArray.**transform**(*buffer: Buffer | List[Buffer]*, *mode: int | None = None*, *vertices: int = -1*, *first: int = 0*, *instances: int = -1*, *buffer_offset: int = 0*) → None

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.
- **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: int*, *cls: str*, *buffer:* Buffer, *fmt: str*, *offset: int = 0*, *stride: int = 0*, *divisor: int = 0*, *normalize: bool = 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.
- **offset** (*int*) – The offset.
- **stride** (*int*) – The stride.
- **divisor** (*int*) – The divisor.
- **normalize** (*bool*) – The normalize parameter, if applicable.

VertexArray.**release**() → None

Release the ModernGL object.

5.4.2 Attributes

VertexArray.**mode**: **int**

Get or set the default rendering mode.

This value is used when `mode` is not passed in rendering calls.

Examples:

```
vao.mode = moderngl.TRIANGLE_STRIPS
```

VertexArray.**program**: *Program*

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

VertexArray.**index_buffer**: *Buffer*

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

VertexArray.**index_element_size**: **int**

The byte size of each element in the index buffer.

VertexArray.**scope**: *Scope*

The scope to use while rendering.

VertexArray.**vertices**: **int**

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.

VertexArray.**instances: int**

Get or set the number of instances to render.

VertexArray.**ctx:** *Context*

The context this object belongs to

VertexArray.**glo: int**

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

VertexArray.**extra: Any**

User defined data.

5.5 Program

class Program

Returned by *Context.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`.

5.5.1 Methods

Program.**get**(*key: str*, *default: Any*) → *Uniform* | *UniformBlock* | *Attribute* | *Varying*

Returns a Uniform, UniformBlock, Attribute or Varying.

Parameters

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

Program.**__getitem__**()

Get a member such as uniforms, uniform blocks, 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__**()

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)
```

Program.**__iter__**()

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.**release**() → None

Release the ModernGL object.

5.5.2 Attributes

`Program.`**`geometry_input: int`**

The geometry input primitive.

The GeometryShader's input primitive if the GeometryShader exists. The geometry input primitive will be used for validation. (from `layout(input_primitive) in;`)

This can only be `POINTS`, `LINES`, `LINES_ADJACENCY`, `TRIANGLES`, `TRIANGLE_ADJACENCY`.

`Program.`**`geometry_output: int`**

The geometry output primitive.

The GeometryShader's output primitive if the GeometryShader exists. This can only be `POINTS`, `LINE_STRIP` and `TRIANGLE_STRIP` (from `layout(output_primitive, max_vertices = vert_count) out;`)

`Program.`**`geometry_vertices: int`**

The maximum number of vertices that the geometry shader will output. (from `layout(output_primitive, max_vertices = vert_count) out;`)

`Program.`**`is_transform: int`**

If this is a tranform program (no fragment shader).

`Program.`**`ctx: Context`**

The context this object belongs to

`Program.`**`glo: int`**

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

`Program.`**`extra: Any`**

User defined data.

5.5.3 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']
)
```

5.5.4 Program Members

Uniform

class Uniform

Available in *Program.__getitem__()*

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

read() → bytes:

Read the value of the uniform.

write(*data: Any*) → None:

Write the value of the uniform.

Attributes

Uniform.**location: int**

The location of the uniform. The result of the glGetUniformLocation.

Uniform.**array_length: int**

If the uniform is an array the array_length is the length of the array otherwise *1*.

Uniform.**dimension: int**

The uniform dimension.

Uniform.**name: str**

The uniform name.

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

Uniform.**value: Any**

The uniform value stored in the program object.

Uniform.**extra: Any**

User defined data.

UniformBlock

class UniformBlock

Available in *Program.__getitem__()*

UniformBlock.**binding: int**

The binding of the uniform block. Same as the value.

UniformBlock.**value: int**

The value of the uniform block. Same as the binding.

UniformBlock.**name: str**

The name of the uniform block.

UniformBlock.**index: int**

The index of the uniform block.

UniformBlock.**size: int**

The size of the uniform block.

UniformBlock.**extra: Any**

User defined data.

StorageBlock

class StorageBlock

Available in *Program.__getitem__()*

Storage Blocks are OpenGL 4.3+ Program accessible data blocks. Compared to UniformBlocks they can be larger in size and also support write operations. For less than one page (64KB) read-only data use UniformBlocks.

StorageBlock.**binding: int**

The binding of the Storage block. Same as the value.

`StorageBlock.value: int`
The value of the Storage block. Same as the binding.

`StorageBlock.name: str`
The name of the Storage block.

`StorageBlock.index: int`
The index of the Storage block.

`StorageBlock.size: int`
The size of the Storage block.

`StorageBlock.extra: Any`
User defined data.

Attribute

`class Attribute`
Available in *Program.__getitem__()*

Represents a program input attribute.

`Attribute.location: int`
The location of the attribute. The result of the glGetAttribLocation.

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

`Attribute.dimension: int`
The attribute dimension.

`Attribute.shape: str`
The shape is a single character, representing the scalar type of the attribute. It is either `'i'` (int), `'f'` (float), `'I'` (unsigned int), `'d'` (double).

`Attribute.name: str`
The attribute name.

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

`Attribute.extra: Any`
User defined data.

Varying

`class Varying`
Available in *Program.__getitem__()*

Represents a program output varying.

`Varying.name: str`
The name of the varying.

`Varying.number: int`
The output location of the varying.

`Varying.extra: Any`
User defined data.

5.6 Sampler

class Sampler

Returned by *Context.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.

5.6.1 Methods

Sampler.**use**(*location: int = 0*) → None:

Bind the sampler to a texture unit.

Parameters
location (*int*) – The texture unit

Sampler.**clear**(*location: int = 0*) → None:

Clear the sampler binding on a texture unit.

Parameters
location (*int*) – The texture unit

Sampler.**assign**(*index: int*) → tuple

Helper method for assigning samplers to scopes.

Example:

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

Sampler.**release**()

5.6.2 Attributes

Sampler.**texture:** *Texture*

The texture object to sample.

Sampler.**repeat_x:** **bool**

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
```

Sampler.**repeat_y: bool**

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
```

Sampler.**repeat_z: bool**

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
```

Sampler.**filter: tuple**

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)
```

Sampler.**compare_func: tuple**

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
```

(continues on next page)

(continued from previous page)

```
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
```

Sampler.**anisotropy: float**

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
```

Sampler.**border_color: tuple**

The (r, g, b, a) color for the texture border (Default `(0.0, 0.0, 0.0, 0.0)`).

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: float**

Minimum level-of-detail parameter (Default `-1000.0`).

This floating-point value limits the selection of highest resolution mipmap (lowest mipmap level)

Sampler.**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)

Sampler.**ctx: *Context***

The context this object belongs to

Sampler.**glo: int**

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

Sampler.**extra: Any**

User defined data.

5.7 Texture

class Texture

Returned by *Context.texture()* and *Context.depth_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.

5.7.1 Methods

Texture.**read**(*alignment: int = 1*) → bytes

Read the pixel data as bytes into system memory.

Parameters

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

Texture.**read_into**(*buffer: Any*, *alignment: int = 1*, *write_offset: int = 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, 2), 1)
texture.read_into(data)
```

Parameters

- **buffer** (*bytearray*) – The buffer that will receive the pixels.
- **alignment** (*int*) – The byte alignment of the pixels.
- **write_offset** (*int*) – The write offset.

Texture.**write**(*data: Any*, *viewport: tuple*, *alignment: int = 1*)

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

Examples:

```
# 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
```

(continues on next page)

(continued from previous page)

```
data = b'\xff\xff\xff\xff\xff\xff\xff\xff'
texture = ctx.texture3d((2, 2), 1)
texture.write(data)
```

Parameters

- **data** (*bytes*) – The pixel data.
- **viewport** (*tuple*) – The viewport.
- **alignment** (*int*) – The byte alignment of the pixels.

`Texture.`**build_mipmaps**(*base: int = 0*, *max_level: int = 1000*) → None

Generate mipmaps.

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

Parameters

- **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*) → None

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.**

Note that we bind the 3D textured layered making the entire texture readable and writable. It is possible to bind a specific 2D section in the future.

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** (`Texture`) – The texture to bind
- **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: int = 0*) → None

Better to use *Sampler* objects and avoid this call on the Texture object directly.

Bind the texture to a texture unit.

Parameters
location (*int*) – The texture location/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 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)
```

`Texture.get_handle`(*resident: bool = True*) → int

Handle for Bindless Textures.

Parameters
resident (*bool*) – Make the texture resident.

Once a handle is created its parameters cannot be changed. Attempting to do so will have no effect. (filter, wrap etc). There is no way to undo this immutability.

Handles cannot be used by shaders until they are resident. This method can be called multiple times to move a texture in and out of residency:

```
>> texture.get_handle(resident=False)
4294969856
>> texture.get_handle(resident=True)
4294969856
```

Ths same handle is returned if the handle already exists.

Note: Limitations from the OpenGL wiki

The amount of storage available for resident images/textures may be less than the total storage for textures that is available. As such, you should attempt to minimize the time a texture spends being resident. Do not attempt to take steps like making textures resident/unresident every frame or something. But if you are finished using a texture for some time, make it unresident.

`Texture.release`()

5.7.2 Attributes

`Texture.`**`width: int`**

The width of the texture.

`Texture.`**`height: int`**

The height of the texture.

`Texture.`**`size: Tuple[int, int]`**

The size of the texture.

`Texture.`**`components: int`**

The number of components of the texture.

`Texture.`**`samples: int`**

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

`Texture.`**`depth: bool`**

Determines if the texture contains depth values.

`Texture.`**`dtype: str`**

Data type.

`Texture.`**`swizzle: str`**

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'
```

`Texture.`**`repeat_x`**

See *Sampler.repeat_x*

`Texture.`**`repeat_y`**

See *Sampler.repeat_y*

`Texture.`**`filter`**

See *Sampler.filter*

Texture.**compare_func**

See *Sampler.compare_func*

Texture.**anisotropy**

See *Sampler.anisotropy*

Texture.**ctx: *Context***

The context this object belongs to

Texture.**glo: int**

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

Texture.**extra: Any**

User defined data.

5.8 TextureArray

class TextureArray

Returned by *Context.texture_array()*

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.

5.8.1 Methods

TextureArray.**read**()

TextureArray.**read_into**()

TextureArray.**write**()

TextureArray.**bind_to_image**()

TextureArray.**build_mipmaps**()

TextureArray.**use**()

TextureArray.**release**()

TextureArray.**get_handle**()

5.8.2 Attributes

TextureArray.**repeat_x**

TextureArray.**repeat_y**

TextureArray.**filter**

TextureArray.**swizzle**

TextureArray.**anisotropy**

TextureArray.**width**

TextureArray.**height**

TextureArray.**layers**

TextureArray.**size**

TextureArray.**dtype**

TextureArray.**components**

TextureArray.**ctx:** *Context*
The context this object belongs to

TextureArray.**glo:** **int**
The internal OpenGL object. This values is provided for interoperability and debug purposes only.

TextureArray.**extra:** **Any**
User defined data.

5.9 Texture3D

class Texture3D
Returned by *Context.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.

5.9.1 Methods

Texture3D.**read**()

Texture3D.**read_into**()

Texture3D.**write**()

Texture3D.**build_mipmaps**()

Texture3D.**bind_to_image**()

Texture3D.**use**()

Texture3D.**release**()

Texture3D.**get_handle**()

5.9.2 Attributes

Texture3D.**repeat_x**

Texture3D.**repeat_y**

Texture3D.**repeat_z**

Texture3D.**filter**

Texture3D.**swizzle**

Texture3D.**width**

Texture3D.**height**

Texture3D.**depth**

Texture3D.**size**

Texture3D.**dtype**

Texture3D.**components**

Texture3D.**ctx:** *Context*

The context this object belongs to

Texture3D.**glo:** **int**

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

Texture3D.**extra:** **Any**

User defined data.

5.10 TextureCube

class TextureCube

Returned by *Context.texture_cube()* and *Context.depth_texture_cube()*

Cubemaps are a texture using the type GL_TEXTURE_CUBE_MAP.

They are similar to 2D textures in that they have two dimensions. However, each mipmap level has 6 faces, with each face having the same size as the other faces.

The width and height of a cubemap must be the same (ie: cubemaps are squares), but these sizes need not be powers of two.

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.

5.10.1 Methods

TextureCube.**read**()

TextureCube.**read_into**()

TextureCube.**write**()

TextureCube.**bind_to_image**()

TextureCube.**use**()

TextureCube.**release**()

TextureCube.**get_handle**()

5.10.2 Attributes

TextureCube.**size**

TextureCube.**dtype**

TextureCube.**components**

TextureCube.**filter**

TextureCube.**swizzle**

TextureCube.**anisotropy**

TextureCube.**ctx**: *Context*
The context this object belongs to

TextureCube.**glo**: **int**
The internal OpenGL object. This values is provided for interoperability and debug purposes only.

TextureCube.**extra**: **Any**
User defined data.

5.11 Framebuffer

class Framebuffer
Returned by *Context.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.

5.11.1 Methods

`Framebuffer.clear`(*red: float = 0.0*, *green: float = 0.0*, *blue: float = 0.0*, *alpha: float = 0.0*, *depth: float = 1.0*, *viewport=...*, *color=...*) → None

Clear the framebuffer.

If a *viewport* passed in, a scissor test will be used to clear the given viewport. This viewport take presence 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.
- **viewport** (*tuple*) – The viewport.
- **color** (*tuple*) – Optional tuple replacing the red, green, blue and alpha arguments

`Framebuffer.read`(*viewport=...*, *components: int = 3*, *attachment: int = 0*, *alignment: int = 1*, *dtype: str = 'f1'*, *clamp: bool = False*) → bytes

Read the content of the framebuffer.

Parameters

- **viewport** (*tuple*) – The viewport.
- **components** (*int*) – The number of components to read.
- **attachment** (*int*) – The color attachment number. -1 for the depth attachment
- **alignment** (*int*) – The byte alignment of the pixels.
- **dtype** (*str*) – Data type.
- **clamp** (*bool*) – Clamps floating point values to `[0.0, 1.0]`

```
# Read the first color attachment's RGBA data
data = fbo.read(components=4)
# Read the second color attachment's RGB data
data = fbo.read(attachment=1)
# Read the depth attachment
data = fbo.read(attachment=-1)
# Read the lower left 10 x 10 pixels from the first color attachment
data = fbo.read(viewport=(0, 0, 10, 10))
```

`Framebuffer.read_into`(*buffer*, *viewport*, *components: int = 3*, *attachment: int = 0*, *alignment: int = 1*, *dtype: str = 'f1'*, *write_offset: int = 0*) → None

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.
- **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.

`Framebuffer.release()`

5.11.2 Attributes

`Framebuffer.viewport: tuple`

Get or set the viewport of the framebuffer.

`Framebuffer.scissor: tuple`

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
```

`Framebuffer.color_mask: tuple`

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
```

Framebuffer.**depth_mask: bool**

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`

Framebuffer.**width: int**

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.

Framebuffer.**height: int**

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.

Framebuffer.**size: Tuple[int, int]**

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.

Framebuffer.**samples: int**

The samples of the framebuffer.

Framebuffer.**bits: int**

The bits of the framebuffer.

Framebuffer.**color_attachments: tuple**

Framebuffer.**depth_attachment: tuple**

Framebuffer.**ctx: *Context***

The context this object belongs to

Framebuffer.**glo: int**

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

Framebuffer.**extra: Any**

User defined data.

5.12 Renderbuffer

class Renderbuffer

Returned by *Context.renderbuffer()* or *Context.depth_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.

5.12.1 Methods

Renderbuffer.**release**()

5.12.2 Attributes

Renderbuffer.**width: int**

The width of the renderbuffer.

Renderbuffer.**height: int**

The height of the renderbuffer.

Renderbuffer.**size: Tuple[int, int]**

The size of the renderbuffer.

Renderbuffer.**samples: int**

The number of samples for multisampling.

Renderbuffer.**components: int**

The number components.

Renderbuffer.**depth: bool**

Determines if the renderbuffer contains depth values.

Renderbuffer.**dtype: str**

Data type.

Renderbuffer.**ctx: *Context***

The context this object belongs to

Renderbuffer.**glo: int**

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

Renderbuffer.**extra: Any**

User defined data.

5.13 Scope

class Scope

Returned by *Context.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.

5.13.1 Methods

Scope.**release**()

5.13.2 Attributes

Scope.**ctx:** *Context*

The context this object belongs to

Scope.**extra:** **Any**

User defined data.

5.13.3 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),
    ],
)

# 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.
```

5.14 Query

class Query

Returned by *Context.query()*

This class represents a Query object.

5.14.1 Attributes

Query.**samples: int**

The number of samples passed.

Query.**primitives: int**

The number of primitives generated.

Query.**elapsed: int**

The time elapsed in nanoseconds.

Query.**crender:** *ConditionalRender*

Query.**ctx:** *Context*

The context this object belongs to

Query.**extra: Any**

User defined data.

5.14.2 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
```

ConditionalRender

class ConditionalRender

Available in *Query.crender*

This class represents a ConditionalRender object.

ConditionalRender objects can only be accessed from *Query* objects.

Examples

```
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.
```

5.15 ComputeShader

class ComputeShader

Returned by *Context.compute_shader()*

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 similar to `moderngl.Program` objects.
- 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()*.

5.15.1 Methods

ComputeShader.**run**(*group_x: int = 1*, *group_y: int = 1*, *group_z: int = 1*) → None:

Parameters

- **group_x** (*int*) – Workgroup size x.
- **group_y** (*int*) – Workgroup size y.
- **group_z** (*int*) – Workgroup size z.

Run the compute shader.

run_indirect(*self*, *buffer:* Buffer, *offset: int = 0*) → None:

Run the compute shader indirectly from a Buffer object.

Parameters

- **buffer** (Buffer) – the buffer containing a single workgroup size at offset.
- **offset** (*int*) – the offset into the buffer in bytes.

ComputeShader.**get**(*key*, *default*)

Returns a Uniform, UniformBlock or StorageBlock.

Parameters

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

ComputeShader.**__getitem__**(*key*)

Get a member such as uniforms, uniform blocks and storage blocks.

```
# 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)
```

(continues on next page)

(continued from previous page)

```
# Set binding for a storage block (if supported)
program['DataBlock'].binding = 0
```

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)

# Set binding for a storage block (if supported)
program['DataBlock'].binding = 0
```

ComputeShader.**__iter__**()

Yields the internal members names as strings.

Example:

```
for member in program:
    obj = program[member]
    print(member, obj)
    if isinstance(obj, moderngl.StorageBlock):
        print('This is a storage block member')
```

This includes all members such as uniforms, uniform blocks and storage blocks.

5.15.2 Attributes

ComputeShader.**ctx:** *Context*

The context this object belongs to

ComputeShader.**glo:** **int**

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

ComputeShader.**extra:** **Any**

User defined data.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

P

Q

R

S

T

U

V

W

Z