ModernGL Documentation

Release 5.7.0

Szabolcs Dombi

Oct 07, 2022

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, * Compling Mesa yourselves see https://docs.mesa3d.org/install.html. * Using msys2, which provids 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 it's dependencies. Then you can add `C:\msys64\mingw32\bin` to PATH before `C:\Windows` and moderngl should be working.

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.

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:

```yaml
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
      - 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:

```yaml
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 specialy configuration to run on macOS.

CHAPTER

TWO

THE GUIDE

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 have 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 4.1 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*`, `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 have 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 some cases you can pretty much copy paste the shaders over to Vulkan.

2.1.3 Where do 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 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.

2.3 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.texture_3d()`
- `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.

2.4 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)

2.5 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() {
```

(continues on next page)

(continued from previous page)

```
        // 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
buffer = ctx.buffer(reserve=NUM_VERTICES * 8)

# 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 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
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`) on `gl_VertexID` can get you pretty far.

2.6 Rendering

```
import moderngl
import numpy as np

from PIL import Image

ctx = moderngl.create_standalone_context()

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

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

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

        in vec3 v_color;

        out vec3 f_color;

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

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

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

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

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

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

2.7 Program

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

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

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

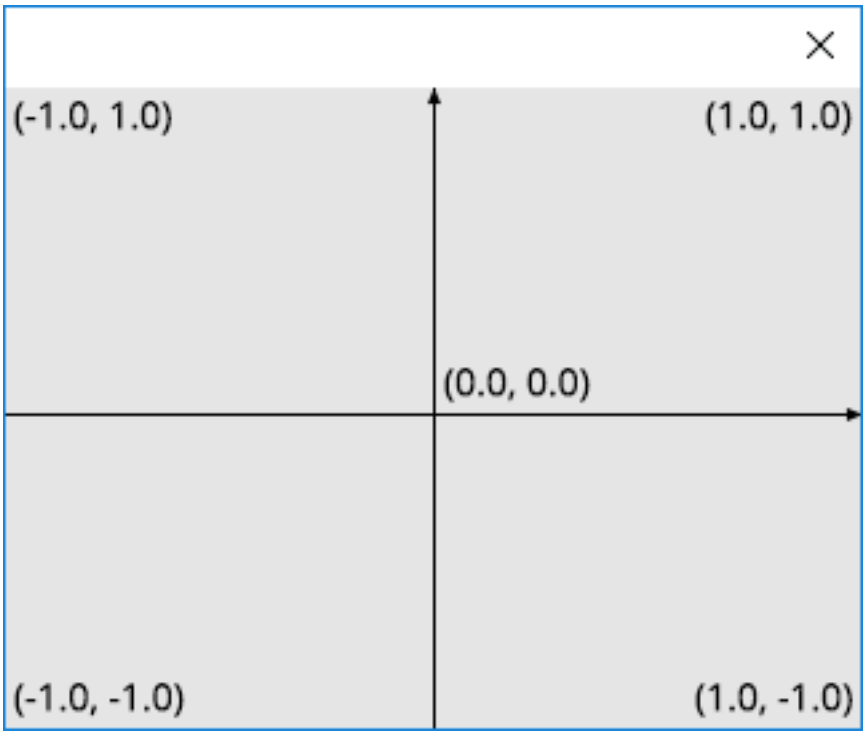


Fig. 1: The screen coordinates

The program will also process a color information.

Entire source

```
import moderngl

ctx = moderngl.create_standalone_context()

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

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

        void main() {
```

(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;
        }
    ''',
)
```

Vertex Shader

```
in vec2 in_vert;
in vec3 in_color;

out vec3 v_color;

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

Fragment Shader

```
in vec3 v_color;

out vec3 f_color;

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

Proceed to the *next step*.

2.8 VertexArray

```
import moderngl
import numpy as np

ctx = moderngl.create_standalone_context()

prog = ctx.program(
    vertex_shader='''
```

(continues on next page)

(continued from previous page)

```
        #version 330

        in vec2 in_vert;
        in vec3 in_color;

        out vec3 v_color;

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

        in vec3 v_color;

        out vec3 f_color;

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

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

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

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

Proceed to the *next step*.

CHAPTER

THREE

TOPICS

3.1 The Lifecycle of a ModernGL Object

Note: Future version of ModernGL might support different GC models. It's an area currently being explored.

3.1.1 Releasing Objects

Objects in moderngl don't automatically release the OpenGL resources it allocated. 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()

# Ensure we don't keep the object around
texture = None
```

This comes as a surprise for most people, but there are a number of reasons moderngl have chosen this approach. Unless you are doing headless rendering we don't even "own" the context itself. It's the window library creating the context for us and we simply detect it. We don't really know exactly when this context is destroyed. There are also other more complicated situations such as contexts with shared resources.

You can create your own `__del__` methods in wrappers if needed, but keep in mind that moderngl types cannot be extended. They only have an `extra` attribute that can contain anything.

3.1.2 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)
```

(continues on next page)

(continued from previous page)

```
<class 'mgl.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
```

(continues on next page)

(continued from previous page)

```
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:
```

(continues on next page)

(continued from previous page)

```
    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, dype="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 rage) 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 contruct it, would then be passed into a VAO using:

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

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

```
in vec2 in_vert;
```

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

```
in vec3 in_color;
```

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

Example of multiple arrays with differing `/usage`

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

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

Our shader will take all the above values as attributes.

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

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

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

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

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

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

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

```
in vec3 in_color;
```

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

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',
)
```

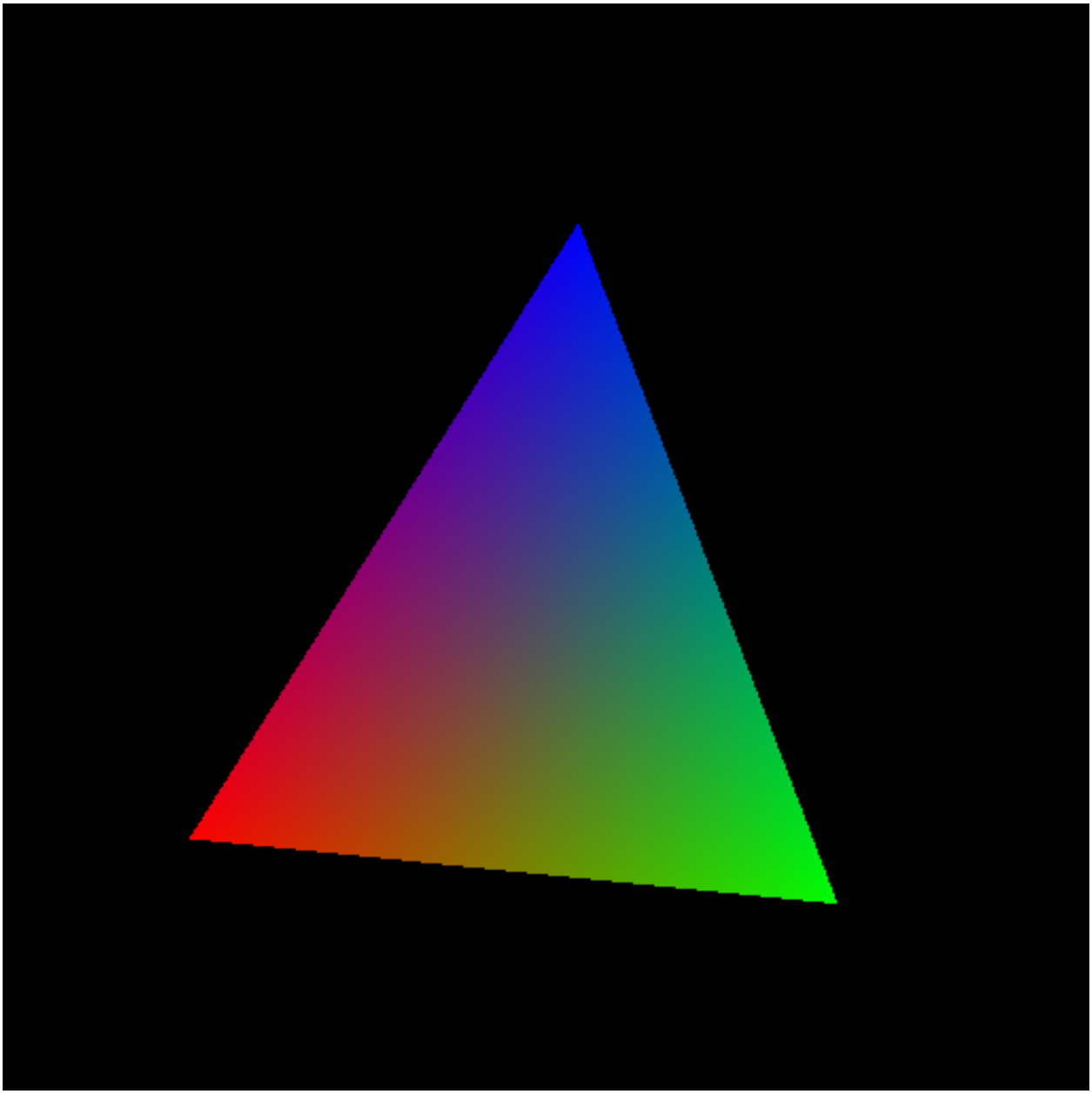
4.1.3 Running an example

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;
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

CHAPTER

FIVE

REFERENCE

5.1 moderngl

5.1.1 Attributes

Attributes available in the root `moderngl` module. Some may be listed in their original sub-module, but they are imported during initialization.

Context Flags

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

Primitive Modes

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

Texture Filters

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

Blend Functions

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

Shortcuts

Blend Equations

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

Provoking Vertex

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

5.1.2 Functions

Also see `Context`.

5.2 Context

5.2.1 Create

5.2.2 ModernGL Objects

5.2.3 Methods

5.2.4 Attributes

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)
```

5.2.6 Primitive Modes

Texture Filters

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

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
```

5.2.8 Blend Function Shortcuts

5.2.9 Blend Equations

Used with `Context.blend_equation`.

5.2.10 Other Enums

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)
```

ContextManager

context_manager.py

example.py

```
from context_manager import ContextManager

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

5.3 Buffer

5.3.1 Create

5.3.2 Methods

5.3.3 Attributes

5.4 VertexArray

5.4.1 Create

5.4.2 Methods

5.4.3 Attributes

5.5 Program

5.5.1 Create

5.5.2 Methods

5.5.3 Attributes

5.5.4 Examples

A simple program designed for rendering

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

        in vec2 vert;

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

        out vec4 color;

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

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']
)
```

5.5.5 Program Members

Uniform

Methods

Attributes

UniformBlock

Subroutine

Attribute

Varying

5.6 Sampler

5.6.1 Create

5.6.2 Methods

5.6.3 Attributes

5.7 Texture

5.7.1 Create

5.7.2 Methods

5.7.3 Attributes

5.8 TextureArray

5.8.1 Create

5.8.2 Methods

5.8.3 Attributes

5.9 Texture3D

5.9.1 Create

5.9.2 Methods

5.9.3 Attributes

5.10 TextureCube

5.10.1 Create

5.10.2 Methods

5.10.3 Attributes

5.11 Framebuffer

5.11.1 Create

5.11.2 Methods

5.11.3 Attributes

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)
```

(continues on next page)

(continued from previous page)

```
    #     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

5.14.1 Create

5.14.2 Attributes

5.14.3 Examples

Simple query example

```
import moderngl
import numpy as np

ctx = moderngl.create_standalone_context()
prog = ctx.program(
    vertex_shader='''
        #version 330

        in vec2 in_vert;

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

        out vec4 color;

        void main() {
            color = vec4(1.0, 0.0, 0.0, 1.0);
        }
    ''',
```

(continues on next page)

(continued from previous page)

```
)

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

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

fbo = ctx.simple_framebuffer((64, 64))
fbo.use()

query = ctx.query(samples=True, time=True)

with query:
    vao.render()

print('It took %d nanoseconds' % query.elapsed)
print('to render %d samples' % query.samples)
```

Output

```
It took 13529 nanoseconds
to render 496 samples
```

5.15 ConditionalRender

5.15.1 Attributes

5.15.2 Examples

Simple conditional rendering example

```
query = ctx.query(any_samples=True)

with query:
    vao1.render()

with query.crender:
    print('This will always get printed')
    vao2.render()  # But this will be rendered only if vao1 has passing samples.
```

5.16 ComputeShader

5.16.1 Create

5.16.2 Methods

5.16.3 Attributes

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

M