Practical 2 - Meshes and modeling

After covering the basic mechanics of OpenGL and communicating with shaders, we now focus on the following goals:

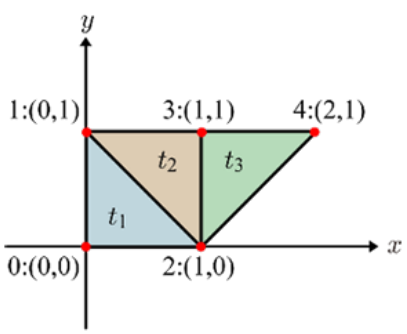
- understanding how to pass several primitives to the GPU
- sending primitives with shared vertices in indexed mode
- creating and drawing meshes with their normals
- making some efficient abstractions to wrap this code for re-use
- drawing a basic canonical shape: cylinder

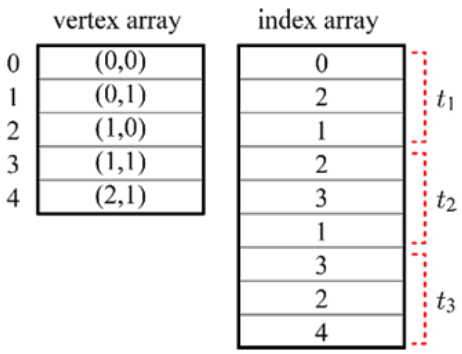
Drawing indexed arrays

Until now we only drew one triangle primitive. To draw more complex surfaces, we need to understand how to pass several triangle primitives to the GPU. The main ways to send and draw a batch of primitives is to use indexed arrays, which we illustrate for 2D triangles:

Indexed vertex array

Indexed vertex arrays allow vertex sharing between triangles, by using two array buffers, one for vertex coordinates, and the second an index array, specifying where to get triangle vertices in the first, in groups of 3:





(_images/vertex_index_array.png)

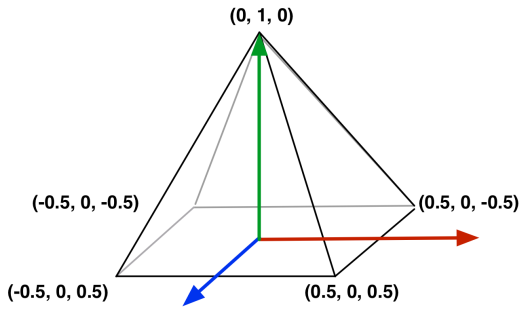
```
# one time initialization
position = np.array(((0, 0), (0, 1), (1, 0), (1, 1), (2, 1)), np.float32)
index = np.array((0, 2, 1, 2, 3, 1, 3, 2, 4), np.uint32)

glid = GL.glGenVertexArrays(1)            # create a vertex array OpenGL identifier
GL.glBindVertexArray(glid)                # make it active for receiving state below

buffers = [GL.glGenBuffers(1)]            # create one OpenGL buffer for our position attri
GL.glEnableVertexAttribArray(0)           # assign state below to shader attribute layout =
GL.glBindBuffer(GL.GL_ARRAY_BUFFER, buffers[0])                         # our created posit
GL.glBufferData(GL.GL_ARRAY_BUFFER, position, GL.GL_STATIC_DRAW)        # upload our vertex
GL.glVertexAttribPointer(0, 2, GL.GL_FLOAT, False, 0, None)             # describe array un
...                                       # optionally add attribute buffers here, same nb

buffers += [GL.glGenBuffers(1)]                                           # create GPU inde
GL.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, buffers[-1])                  # make it active
GL.glBufferData(GL.GL_ELEMENT_ARRAY_BUFFER, index, GL.GL_STATIC_DRAW)     # our index array


# when drawing in the rendering loop: use glDrawElements for index buffer
GL.glBindVertexArray(glid)                                                # activate our ve
GL.glDrawElements(GL.GL_TRIANGLES, index.size, GL.GL_UNSIGNED_INT, None)  # 9 indexed verts
```

Note

The additional index array specified with `GL_ELEMENT_ARRAY_BUFFER` is bound to the vertex array `glid` just like all vertex attributes. Which means its state is also brought along when re-binding the vertex array for drawing with the `glBindVertexArray(glid)` call. There can be only one index array per OpenGL vertex array, obviously.

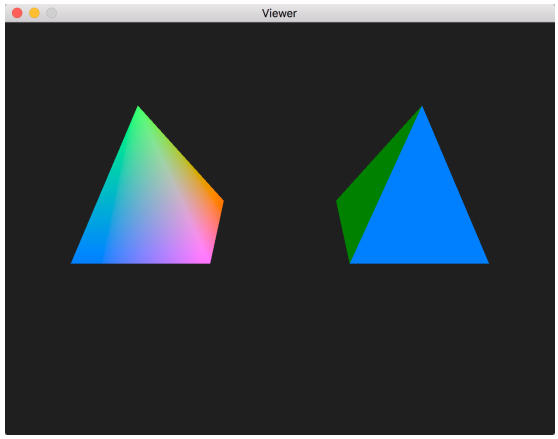
Exercise 1. Draw colored pyramids

- Using everything you learned up until now, create two new classes similar to **`SimpleTriangle`** . They will each draw a pyramid, as per the figure below for example. We want the first pyramid class to have per-vertex colors, interpolated across faces; and the second pyramid class should have uniform per face colors. Add an instance of each object to your viewer in **`main()`** for testing.
- By convention for various tasks such as determining visibility, the vertices should be enumerated in counter-clockwise order when facing the surface. Use `GL.glEnable(GL.GL_CULL_FACE)` at Viewer initialization which enables back face culling, eliminating non-front facing polygons, to check the correctness of your orientation
- Along the way, you may encounter some occlusion issues. Look up the OpenGL documentation about depth testing to solve this problem.



(_images/pyramid.png)

Pyramid coordinates.



(_images/pyramids.png)

Expected result.

Note

- you can statically translate a Numpy position array by just adding a 3-vector to it (any iterable will do, Numpy array, triplet or list of 3 elements). This is useful to visualize your two pyramids without them overlapping:

```
position += (1, 0, 0)
```

Exercise 2. Vertex Array Wrapper

We've only created three object classes until now, and you've already copy-pasted the essentially identical buffer creation code 5 or 6 times. Because of the low-level nature of the OpenGL API, and the error-prone repetition, every OpenGL programmer in the world ends up writing his own refactored wrapper code or using somebody else's at some point.

We provide a **`VertexArray`** class doing just that, with the following usage, with arbitrary N:

```
# one-time initialization code
attribute0 = np.array(..., np.float32)       # 2D array, 1 row per vertex, intended layout
...
attributeN = np.array(..., np.float32)       # 2D array, 1 row per vertex, intended layout
index = np.array(..., np.uint32)             # 1 dimensional array of unsigned integer indi

my_vertex_array = VertexArray([attribute0, ..., attributeN], index)

# draw code in the rendering loop
my_vertex_array.execute(GL.GL_TRIANGLES)
```

You can copy-paste the class code as follows:

```
class VertexArray:
    """ helper class to create and self destroy OpenGL vertex array objects. """
    def __init__(self, attributes, index=None, usage=GL.GL_STATIC_DRAW):
        """ Vertex array from attributes and optional index array. Vertex
            Attributes should be list of arrays with one row per vertex. """

        # create vertex array object, bind it
        self.glid = GL.glGenVertexArrays(1)
        GL.glBindVertexArray(self.glid)
        self.buffers = []  # we will store buffers in a list
        nb_primitives, size = 0, 0

        # load buffer per vertex attribute (in list with index = shader layout)
        for loc, data in enumerate(attributes):
            if data is not None:
                # bind a new vbo, upload its data to GPU, declare size and type
                self.buffers += [GL.glGenBuffers(1)]
                data = np.array(data, np.float32, copy=False)  # ensure format
                nb_primitives, size = data.shape
                GL.glEnableVertexAttribArray(loc)
                GL.glBindBuffer(GL.GL_ARRAY_BUFFER, self.buffers[-1])
                GL.glBufferData(GL.GL_ARRAY_BUFFER, data, usage)
                GL.glVertexAttribPointer(loc, size, GL.GL_FLOAT, False, 0, None)

        # optionally create and upload an index buffer for this object
        self.draw_command = GL.glDrawArrays
        self.arguments = (0, nb_primitives)
        if index is not None:
            self.buffers += [GL.glGenBuffers(1)]
            index_buffer = np.array(index, np.int32, copy=False)  # good format
            GL.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, self.buffers[-1])
            GL.glBufferData(GL.GL_ELEMENT_ARRAY_BUFFER, index_buffer, usage)
            self.draw_command = GL.glDrawElements
            self.arguments = (index_buffer.size, GL.GL_UNSIGNED_INT, None)

        # cleanup and unbind so no accidental subsequent state update
        GL.glBindVertexArray(0)
        GL.glBindBuffer(GL.GL_ARRAY_BUFFER, 0)
        GL.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, 0)

    def execute(self, primitive):
        """ draw a vertex array, either as direct array or indexed array """
        GL.glBindVertexArray(self.glid)
        self.draw_command(primitive, *self.arguments)
        GL.glBindVertexArray(0)

    def __del__(self):  # object dies => kill GL array and buffers from GPU
        GL.glDeleteVertexArrays(1, [self.glid])
        GL.glDeleteBuffers(len(self.buffers), self.buffers)
```

Note

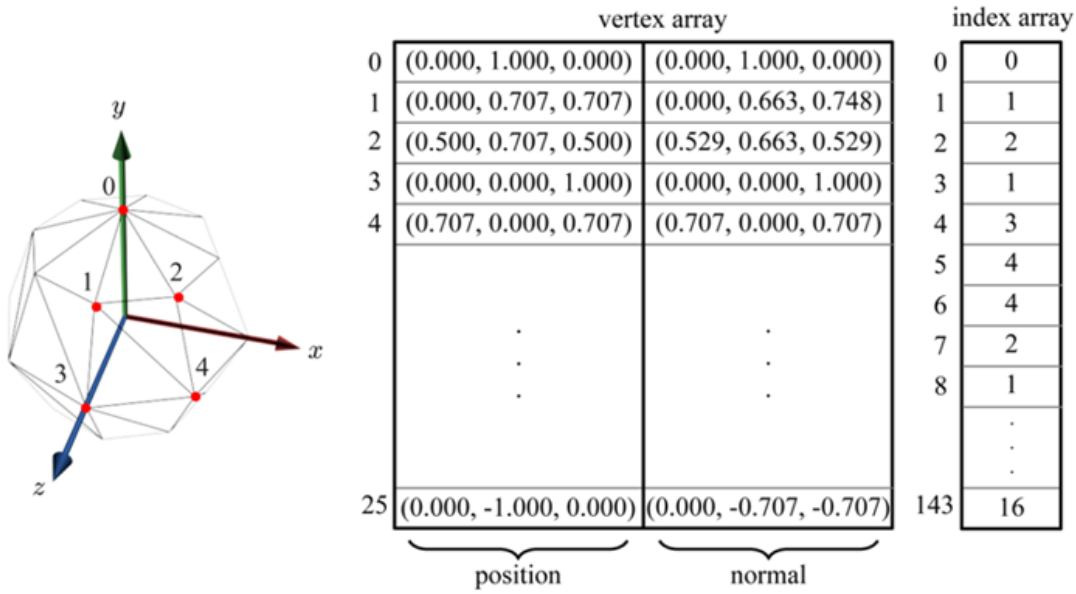
- Thanks to the layout index trick in GLSL and OpenGL syntax, no reference to any shader id needs to be made in this class

Using this newly provided class, revisit your previous implementation of the pyramids by replacing your custom indexed vertex buffer code with simplified wrapper-based code.

Meshes and normals

The main representation for surfaces in 3D Graphics is a **polygon mesh**, which is primarily a set of vertices and their polygon connectivities. While general meshes can support polygons of any arity, OpenGL 4 supports triangle meshes only. As we have seen, they can be passed on to the GPU with vertex arrays or indexed vertex arrays.

Among the various attributes that can be associated to mesh vertices, one stands out and is extensively used throughout the Graphics pipeline, notably for illumination: the **surface normals**. They indicate surface orientation by pointing *outward* with respect to the object. They must be of **unit norm**. Surface normals are typically specified either per-face, or per-vertex with normal interpolation on faces for perceived smoothness, as in this indexed array example:



(_images/normal_array.png)

Loading models from files

There are many formats to store meshes. One of the most popular is the *.obj* format, which describes in text-readable format a succession of vertex positions, vertex normals, then face indices and face attributes. Here is a sample to use and look at:

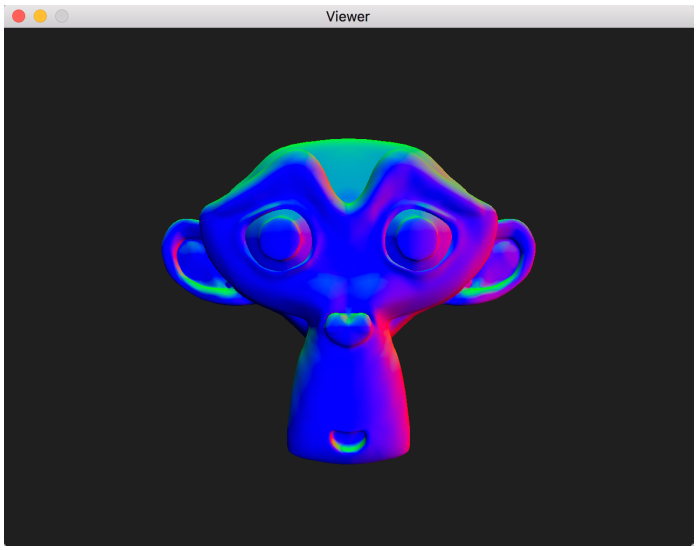
`Download suzanne.obj` (_downloads/711ec76aa7815cd5ba5eae366f6a0041/suzanne.obj)

We provide a simple code to load such a file in your viewer, which you can copy-paste in your code. It is based on the external module pyassimp (https://pypi.python.org/pypi/pyassimp/), a Python wrapper of the assimp (http://www.assimp.org) 3D asset importer library:

```
import pyassimp                     # 3D ressource loader
import pyassimp.errors              # assimp error management + exceptions
...

# -------------- 3D ressource loader ----------------------------------------
def load(file):
    """ load resources from file using pyassimp, return list of ColorMesh """
    try:
        option = pyassimp.postprocess.aiProcessPreset_TargetRealtime_MaxQuality
        scene = pyassimp.load(file, option)
    except pyassimp.errors.AssimpError:
        print('ERROR: pyassimp unable to load', file)
        return []     # error reading => return empty list

    meshes = [ColorMesh([m.vertices, m.normals], m.faces) for m in scene.meshes]
    size = sum((mesh.faces.shape[0] for mesh in scene.meshes))
    print('Loaded %s\t(%d meshes, %d faces)' % (file, len(scene.meshes), size))

    pyassimp.release(scene)
    return meshes
```

Exercise 3. ColorMesh class

The provided loading code is based on the assumption that you write a generic **`ColorMesh`** class (highlighted usage above) for which the initializer takes a list of attributes as first parameter, and an optional index buffer, just like the **`VertexArray`** class described above which it may invoke. The difference is that the **`draw()`** method of your new mesh class should invoke the color shader to draw the object whose attributes have been passed along.



(_images/suzanne_normal.png)

Expected rendering of *suzanne.obj* with interpolated normals rendered as color values

Optional Exercises

Exercise 4 is independent of 5 and 6.

Exercise 4. Refactoring

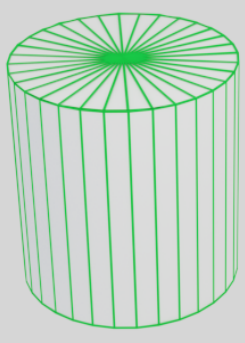
Level: easy

To avoid code redundancy, all your previous classes `SimpleTriangle`, the pyramid classes, (and the `Cylinder` class below) can actually be made to derive from `ColorMesh` and invoke its base constructor to create the geometry. They can thus all share the same `draw()` method from the `ColorMesh` base class, which, up until now, was copy pasted among all these classes.

Exercise 5. Cylinder

Level: medium

To gain further intuition about meshes and normals it is of interest to build at least one canonical object by hand. The lateral surface of a unit cylinder can be described by the following parametric equations:



(_images/cylinder.png)

$$\begin{cases} x &= \cos\theta \\ y &= \sin\theta \\ z &= h \end{cases} \quad \text{for } \theta \in [0, 2\pi[\text{ and } h \in [0, 1].$$

Create a “canonical” cylinder centered along the z axis, with a radius of 1 and bases on $h = 0$ and $h = 1$, with a new Python class with the same layout as your pyramids. Don’t care about normals for now.

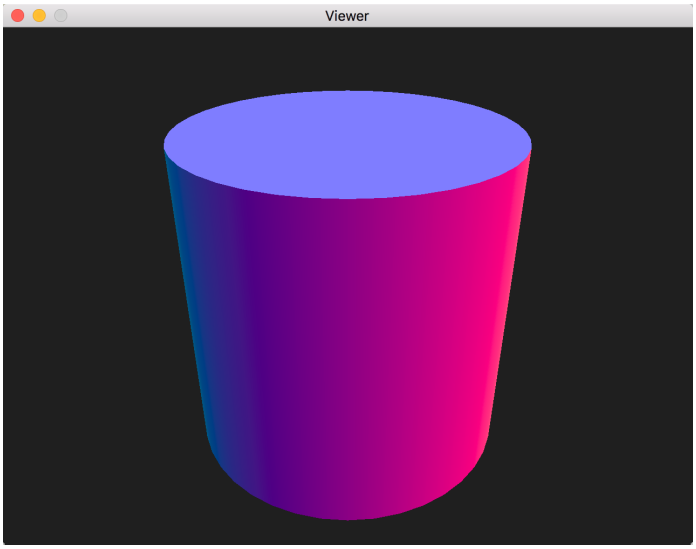
- Discretize the lateral surface of the cylinder into quad facets, then triangles. The number of facets could be parametrized in the constructor.
- Complete the geometry with the bottom and top bases. Ask yourself: how any vertices should be passed to the shader? Are they shared by multiple triangles?
- For simplicity, use a translated version of the position as your color array
- Add an instance of your new class to the viewer in `main()`

Exercise 6. Cylinder normals

Level: medium

Add normals to your previous cylinder. Hint: it is fairly easy to get them from a modified version of the position array.

Use your color shader to display your normals; for appropriate visualization, try to pass a rectified version of the normals as your color buffer, such that normals are on a sphere centered on $(0.5, 0.5, 0.5)$ of radius 0.5.



(_images/cylinder_normal.png)

Expected rendering of the cylinder with interpolated normals rendered as color values

Back to top