CS3388B: Lecture 10

February 28, 2023

10 The Render Pipeline

We are now familiar with Model, View, and Projection matrices. We are also familiar with rasterization, clip space, and Normalized Device Coordinates. Let's put it all together now.

The basic render pipeline follows five main steps:

1. Vertex specification
2. Vertex processing
3. Vertex post-processing:
   - (i) Primitive assembly
   - (ii) Clipping
   - (iii) Face culling
4. Rasterization
5. Sample Processing: depth testing, Blending, etc.

We've already done all of this in one way or another. But let's get the details now.

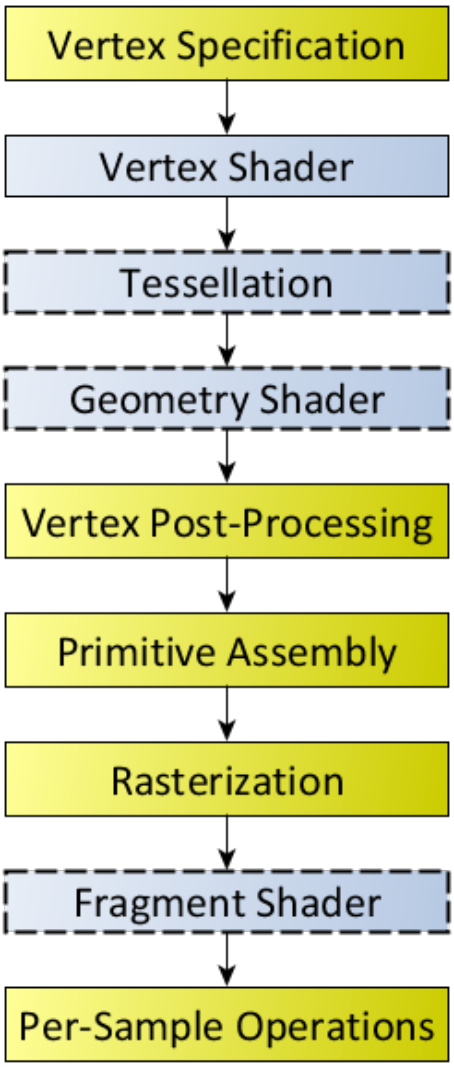
10.1 Vertex Attributes and Specification

Vertices in the most basic sense are points in space. We know that vertices are combined to generate primitives (triangles, quads, fans, etc.). Vertex attributes in general are any piece of information associated with a vertex that is then used to create a primitive. In fact, **vertex position** is the only *required* vertex attribute. Other common attributes are **color**, **normal**, and **texture coordinates**.

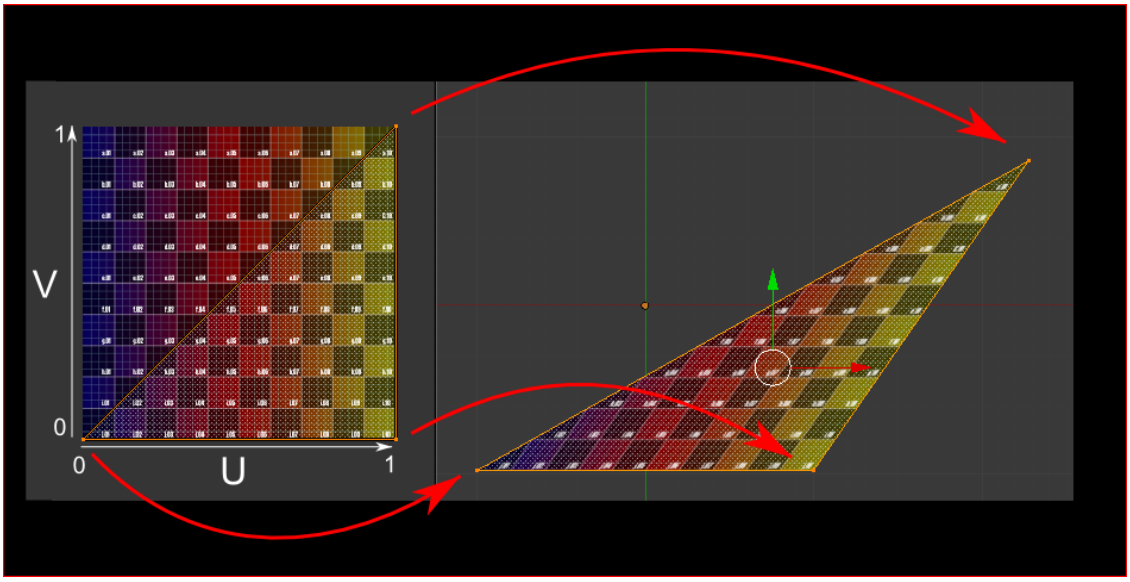
We've seen how to specify these attributes already, using functions like:

- `glColor`: color
- `glNormal`: normal
- `glTexCoord`: texture coordinate
- `glVertex`: vertex position and specification.

Recall that OpenGL is a state machine. Thus, when we specify vertex attributes with `glColor`, `glNormal`, `glTexCoord`, those attributes persist and are applied for every vertex specified with each `glVertex` call.


Vertex Specification
Vertex Shader
Tessellation
Geometry Shader
Vertex Post-Processing
Primitive Assembly
Rasterization
Fragment Shader
Per-Sample Operations

Texture Mapping



Texture mapping is the process of covering a triangle (or many triangles) with an image (a **texture**). Rather than a color, we use textures as a sort of *wallpaper* over the surface of a triangle.

We typically specify **texture coordinates** as a pair of floating point numbers in the range [0, 1]. These coordinates are often called (u, v) or (s, t). The first coordinate u in the range [0, 1] maps to the range [0, w] for an image of width w. The first coordinate v in the range [0, 1] maps to the range [0, h] for an image of width h. Texture coordinates are also commonly referred to as **UV-coordinates**.

A texture coordinate is specified for each vertex and then that image is mapped (transformed, stretched and squished) to fit onto the face of the triangle based on those three coordinates. How that mapping occurs we'll see in Section 10.4.

10.2 Vertex Processing: Fixed Function Pipeline

Vertex processing is how vertices and their attributes go from specification to primitive assembly.

Up to now, we have always been using the **fixed function pipeline** This render pipeline is said to be *fixed* because it has no programmatic capabilities. Rather, the client only specifies particular values for some fixed parameters.

We can feel that the fixed function pipeline is a sort of "default behaviour". Vertex processing in the fixed function pipeline is quite basic; the only modification applied before primitive assembly is matrix multiplication. Each vertex position specified by `glVertex` is transformed by `GL_MODELVIEW` and then `GL_PROJECTION`. Then, primitive assembly and clipping occurs (as discussed in Lecture 9).

10.3 Primitive Assembly & Face Culling

We know that a sequence of vertices can be *interpreted* as certain kinds of primitives: lines, triangles, quads, triangle strips, etc. But, **primitive assembly** is rather the process or breaking down a vertex stream from what it was originally specified as into a sequence of **base primitives**: points, lines, or triangles.

For example, a line strip (polyline) becomes a sequence of line segments. A triangle fan, triangle strip, quad, quad strip becomes a sequence of triangles.

When triangles are the target base primitive, an additional **face culling** step can be applied to remove any triangles from the pipeline whose front face is or is not facing the camera.

Using **winding order** and `glFrontFace`, we specify whether the front face of a triangle is the one where vertices circle clockwise or counterclockwise around its center. Then, we can enable face calling through `glEnable(GL_CULL_FACE)`. Then, we tell which kinds of faces to cull: the ones whose front is facing towards the camera, `glCullFace(GL_FRONT)`; the ones whose back is facing towards the camera, `glCullFace(GL_BACK)`; or both, `glCullFace(GL_FRONT_AND_BACK)`.

10.4 Rasterization & Attribute Interpolation

As we recall from Lecture 6, rasterization is the process of converting base primitives to **fragments**. From OpenGL itself: "Rasterization is the process whereby each individual Primitive is broken down into discrete elements called Fragments, based on the sample coverage of the primitive."

Fragments provide all the information needed to color a pixel. This includes raster position, depth, color, and any other attributes. Unlike vertex attributes, the key part of rasterization which we did not cover in lecture 6 was **interpolation**. Interpolation is the process that rasterization uses to turn vertex attributes into the values for each fragment.

As the name suggests, interpolation is used to *interpolate* multiple vertices' attributes across all fragments between those vertices. Between two vertices, this is a simple **lerp** between the attributes of each vertex. But, since a triangle is composed of three vertices, it's a not-so-simple lerp. It's really a weighted average based on the distance between a fragment's position and each of the vertices.

The simplest example for this is a triangle. Consider a triangle where each vertex is given a different color: red, green, and blue. The resulting fragments of this triangle varies through combinations of those colors. However, this interpolation happens for any and all vertex attributes: normals, texture coordinates, etc.

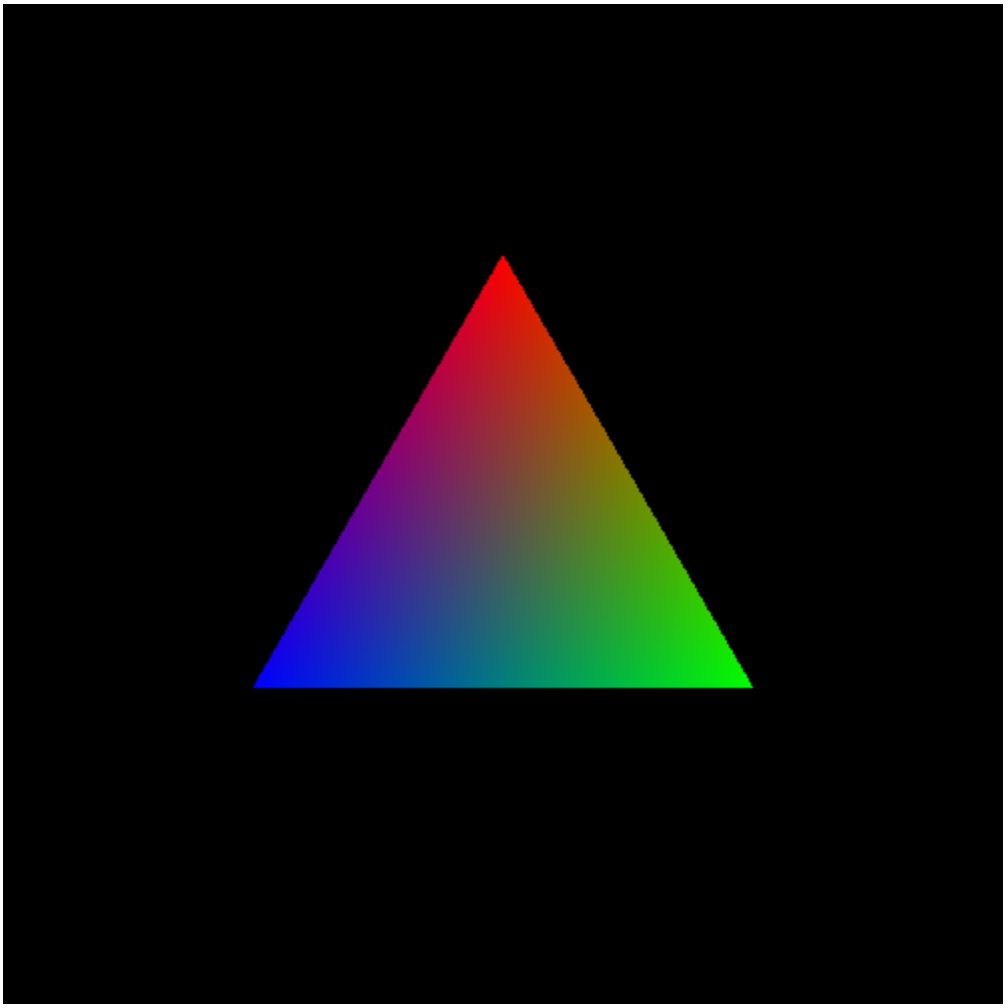


Figure 1: Interpolation of color between vertices

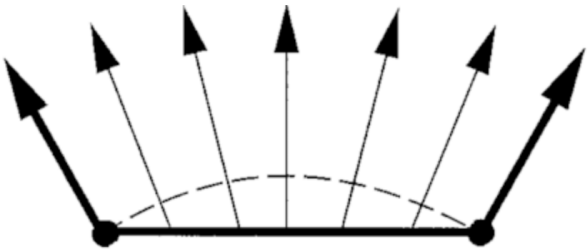


Figure 2: Interpolation of normals between vertices

In fact, this is exactly how texture mapping works. (If a primitive is being texture mapped.) Each fragment gets an interpolated value of texture coordinates. Then, we access a texture's pixel (a **texel**) to determine what color the fragment should be.

But, you ask, images have discrete pixels while texture coordinates are floating point numbers, so how do we determine the correct color? The answer to this question is **texture filtering**. There are two basic strategies. **Nearest neighbour** filtering simply chooses the texel which is closest to the texture coordinate. **(Bi)linear filtering** is essentially a weighted mean based on the distance from a texture coordinate to its nearest four texels.

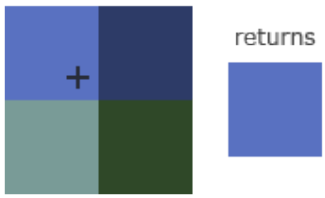


Figure 3: Nearest Neighbour Filtering

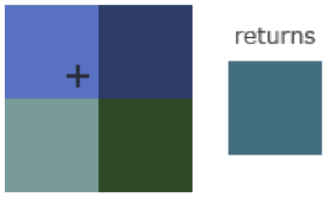


Figure 4: Linear Filtering

10.5 Sample Processing: Depth and Blending

After all this effort, we have many fragments they need to be drawn to the frame buffer. Of course, it is very possible that different primitives overlap and thus may result in more than one fragment covering the same pixel. This is where **per-sample operations** come into play.

The main considerations are **depth** and **blending**. Without either of these being enabled, the last fragment to be drawn is the one that gets displayed in a pixel of a frame buffer. We enable either using `glEnable`.

10.5.1 Depth.

Depth testing proceeds as follows. When a fragment is determined to cover a particular pixel, it first checks if that pixel is already covered. If not, color that pixel. If it is, either color the pixel again (**overdraw**), or discard the current fragment and leave the pixel unmodified.

The choice of to overdraw or to discard the current fragment is the **depth test**. With depth testing enabled, the fragment's raster position uses its z coordinate after perspective divide (its depth) to determine whether the current fragment "passes" the depth test and should be used to overdraw a pixel. What values pass a depth test depends on the chosen depth function.

We specify the depth function in OpenGL with `glDepthFunc`. Its options include `GL_LESS`, `GL_GREATER`, among others. With `GL_LESS`, if the depth value of a particular fragment is *less* than the depth value of the fragment used previously to color the pixel, then the depth test passes and the new fragment colors the pixel again. One proceeds similarly with other comparison methods.

The actual comparison of depth values is facilitated through the **depth buffer**. When we draw a fragment's color to a particular pixel in a frame buffer, only the fragment's color is stored. We need another place to store the fragment's (and thus pixel's) depth. This is the depth buffer. It is another buffer with the same dimensions as the frame buffer, and thus a one-to-one correspondence between the pixels in the depth buffer and the pixels in the frame buffer. When depth testing is enabled, each time we draw a fragment's color to the frame buffer, we similarly draw a fragment's depth to the depth buffer.

10.5.2 Blending.

Blending, as the name suggests, allows fragments to be blended and mixed together to get interesting effects. When a fragment is to be drawn to a frame buffer (and blending is enabled) the **blend function** determines how that fragment's color is mixed with the existing pixel color in the frame buffer.

There are many possible blend functions, but their basic premise that it specifies a weighted sum of the **source color** (the color to be drawn) and the **destination color** (the existing color in the frame buffer). Each color channel is blended separately.

Let $f_{s,R}$, $f_{s,G}$, $f_{s,B}$, $f_{s,A}$ be the scale factors for the source color channels R, G, B, A.
Let $f_{d,R}$, $f_{d,G}$, $f_{d,B}$, $f_{d,A}$ be the scale factors for the destination color channels R, G, B, A.

The new color in the frame buffer will become this weighted sum of source color S_i and destination color D_i:

$$new_i = S_i f_{s,i} + D_i f_{d,i}$$

for the color channels $i \in \{R, G, B, A\}$.

Of course, new_i is also clamped so as to not overflow the maximum value for a particular color channel in the current color space.

We specify the blend function using `glBlendFunc` and 5stating the source scale factor and the destination scale factor. By default, the source scale factors are 1 and the destination scale factors are 0. That is, `glBlendFunc( GL_ONE, GL_ZERO)`. The most common blend function is `glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )`. With this blend function, $f_{s,i} = S_A$ and $f_{d,i} = 1 - S_A$. If you work out the math, this blend function mimics the effect of **translucent** (semi-transparent) objects. However, with this blend function, it also *requires* that primitives are drawn in order **furthest to nearest**. Why? (See problem sets).

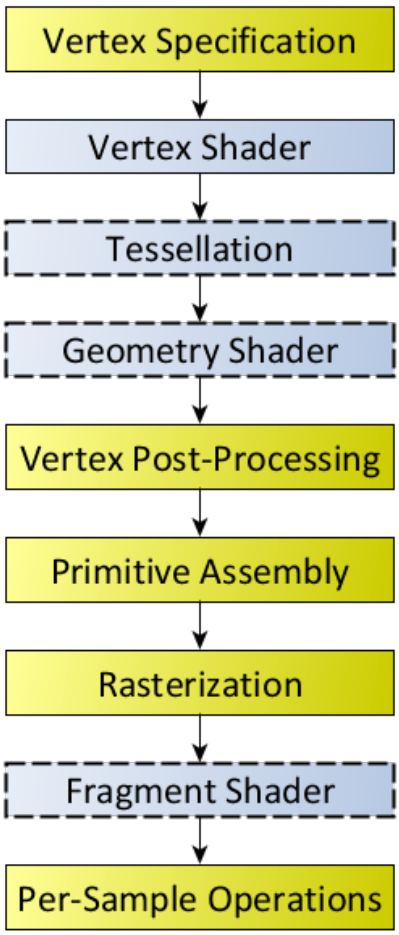
10.5.3 Clearing Buffers

But what happens in blending if this is the first fragment found to cover a particular pixel? There is no destination color to blend with!?

Well, hopefully we used **glClear** to *clear* the frame buffer by writing a constant color to it everywhere. This is the "background color". **glClearColor** specifies the default color which fills the frame buffer when we call `glClear` with the parameter `GL_COLOR_BUFFER_BIT`

We can also use `glClear` to clear the depth buffer and fill it with a particular depth value. That depth value is specified with **glClearDepth**. The default value is 1.

And that's it! Kind of. That's all the automatic and otherwise *fixed functionality* of the rendering pipeline. Through these previous sections we have only been specifying the different parameters of fixed functions (e.g. blending, depth, vertex attributes). In modern graphics libraries we instead have a **programmable pipeline**. Each of the blue boxes below are **shaders**—areas in the render pipeline which allow the execution of arbitrary code (with some requirements). Dashed boxes are optional shaders.



10.6 GLSL: OpenGL Shading Language

GLSL is a C-like programming language used in OpenGL to program shaders.

The (compiled) code defining the functions to be executed as part of a particular shader is called a **shader object**. The linking of several shaders (at least a vertex shader plus any combination of tesselation, geometry, or fragment) is called a **program**.

While GLSL is C-like it does not have all the functionality of a C program. But, it does have a lot of bonus functionality. There are built-in data types for vectors, matrices, textures, and more. Check out the docs: https://docs.gl/sl4/all

Since the rendering pipeline is a **pipeline** the flow of data follows simple input and output mechanics. The basics of a GLSL shader are:

- a version code header: `#version XYZ`
- global **input variables**
- global **output variables**
- a **main** function

```
#version 330 core

in vec3 fromPreviousShader; //input variables are declared with in

out vec3 toNextShader; //output variables are declared with out

void main() {
    toNextShader = fromPreviousShader;
}
```

You can also write code for arbitrary functions and call those functions from main, as needed. Rather than going through all the specific syntax and available functions, let's see the minimum requirements of the shaders and then learn by playing and doing.

10.7 Vertex Shaders

For each draw to the screen, a vertex shader is executed for each vertex specified in a particular draw call. The inputs of a vertex shader come from vertex specification and any vertex attributes you have defined. **If a vertex shader expects a particular attribute you better specify it**.

The outputs of a vertex shader as passed to the next stage of the render pipeline which is specified. Recall tesselation and geometry shaders are optional. If neither is specified, the output of a vertex shader goes directly to vertex post-processing.

A vertex shader, at the very minimum, **must** produce a vertex position. Everything else is up to you. This output position uses an automatically defined output variable **gl_Position**.

The vertex's position is specified to be in **clip space**. For a vertex specified in local coordinates, the output of the shader must be $PVMv_{local} = v_{clip}$. Most commonly, the shader itself does the multiplication of the input vertex position with the MVP matrix. However, it is always possible to do the matrix multiplication on the client side and *specify* the vertex position to already be in clip space.

10.8 Tesselation & Geometry Shaders

Tesselation and Geometry shaders are ways to generate more primitives (not just base primitives) from existing primitives. This is a sort of primitive pre-assembly and extension. Loosely speaking, tesselation shaders are used to sub-divide an existing primitive into more smaller ones. Geometry shaders are used to create new vertices (and thus primitives) from existing vertices and the primitives they represent.

To be continued...

10.9 Fragment Shaders

After vertex post-processing, vertex assembly, and rasterization, we have a number of fragments which cover pixels to be drawn.

Fragment shaders receive some inputs by default, which do not need to be explicitly named as input variables:

- `vec4 gl_FragCoord`: the (x, y) location of the fragment in window space. The z coordinate of this vector is the fragment's depth, the w coordinate is $\frac{1}{w_{clip}}$.
- `bool gl_FrontFacing`: a boolean to say if this fragment was generated from a primitive facing the camera or not, depending on winding order, etc.

A fragment also receives *interpolated vertex attributes* like color, normal, UV-coordinates, etc. Any vertex attribute which is output from the vertex shader.

In order so actually *see* a fragment, the fragment shader must output at least one color to be drawn. Traditionally, the output of a fragment shader uses the built-in output variable `out vec4 gl_FragColor`, which is the color to draw to the frame buffer.

In advanced fragment shaders and very modern OpenGL, you can use a fragment shader to draw to several buffers simultaneously by specifying an array of output colors. If the shader does not assign to `gl_FragColor` then it should specify another `out` variable which holds the color to draw (the number of coordinates in that output variable should equal the number of color channels expected by the buffer to be drawn to). Typically, this is `vec4` for R, G, B, A.

10.10 Interface Matching

For a shader which outputs values, the next shader in the pipeline (if one exists) must take input values which match the outputs. This is **interface matching**. A variable's type, name and

location (to be defined later) must all match.

If a vertex shader has `out vec4 myVertexData`, then the next shader (whether it be the geometry shader or fragment shader or whatever) should have `in vec4 myVertexData`.

10.11 Shaders in practice

Okay, we know the jist and intention of shaders. Now, let's use them in practice.

Shaders are specified as *strings*. Which are compiled into shader objects, and linked into programs dynamically at the runtime of an OpenGL application. To create a shader object, we proceed as follows:

1. **Create** a shader object: `glCreateShader(GL_SHADER_TYPE)`. This returns a **unique ID** specifying the shader object instance.
2. **Specify the source code** of the shader: `glShaderSource(shaderID, count, stringArray, lengthArray)`
3. **Compile** the shader: `glCompileShader(shaderID)`

Specifying the source code with worthy of more details. Typically, we specify a single string which contains all of the GLSL source code of the shader. However, OpenGL allows several strings to be passed (as `stringArray`) and thus a concatenation of all those strings (really a `char*`) creates the source code of the shader. `lengthArray` can either be an array of integers which are the length of each string, or `NULL`. If `lengthArray` is `NULL`, then OpenGL assumes that each string is NULL-terminated (as all strings in C should be).

We do this for each shader we want to use. Recall that must include at least a vertex shader, and then additionally and optionally, tesselation, geometry and fragment.

With a collection of shader objects, we still don't have a *program*. We have to link shader objects to a program and to each other to create a valid program to be used in the programmable render pipeline. This is where interface matching occurs. Moreover, the encapsulation and discretization of shader objects allows shader objects to be re-used several times in different programs. (As we will see, we can change the shader program for each draw call to get different effects for different objects to be drawn.)

To create a program:

1. **Create** the program: `glCreateProgram()` returns a unique ID for the created program instance.
2. **Attach** shader objects to the program: `glAttachShader(programID, shaderID)`. Attach various shader objects of varying types (vertex, fragment, etc.)
3. **Link** the program: `glLinkProgram(programID)`. This creates the executable program to be used as part of the programmable render pipeline.

Once a program is successfully linked, its state is fixed until another call to `glLinkProgram`. Thus, if programs are not likely to change and shader objects are not likely to be reused, shader objects can be detached and deleted to clean up resources.

10.12 Revisiting Vertex Specification

Now we need to **use** the just created shader program. Recall that vertex shaders take in arbitrary collection of attributes. We need to specify where those attributes can be read from to be passed to the shader.

It beings with a **location**. Each attribute in a vertex shader is given a particular location, essentially an attribute index. Each input variable of the vertex shader is implicitly given a location based on its order of declaration.

```
in vec3 vertexPosition_modelspace;
in vec4 color_in;

void main() { ... }
```

In the above, `vertexPosition_modelspace` implicitly has location 0 and `color_in`, location 1. We can also explicitly give a variable a location by prefacing it with `layout(location = i)`.

```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec4 color_in;

void main() { ... }
```

We must tell OpenGL where these vertex attributes can be found **for each draw call**. Indeed, OpenGL is a state machine. So, if we have multiple objects to be drawn (and thus multiple different vertex attribute pointers), the vertex attribute pointer should change for each draw call. Thus, we should set the vertex attribute pointer every time before drawing.

Let's say we have parallel arrays of floats for the vertex attributes of position and color. For example, vertex i should have have (x, y, z) position:
`(vertices[3*i], vertices[3*i + 1], vertices[3*i + 2])`
and RGBA color:
`(colors[3*i], colors[3*i + 1], colors[3*i + 2], colors[3*i + 3])`.

We must **enable** a vertex attribute at a specific location, and then provide a **pointer** to that attribute data.

```
glEnableVertexAttribArray(0);          //enable location 0
glVertexAttribPointer(
    0,                                 // attribute location in shader
    3,                                 // this attribute has 3 coordinates
    GL_FLOAT,                          // type
    GL_FALSE,                          // normalized?
    0,                                 // stride
    vertices                           // vertex attribute pointer
```

```
);

glEnableVertexAttribArray(1);          //enable location 1
    glVertexAttribPointer(
    1,                                 // attribute location
    4,                                 // this attribute has 4 coordinates
    GL_FLOAT,                          // type
    GL_FALSE,                          // normalized?
    0,                                 // stride
    colors                             // vertex attribute pointer
);
```

10.12.1 Shader uniforms

Recall that vertex shaders should output vertex positions in clip space. It is likely that the vertex position which are input to a vertex shader is not in clip space. Usually they are in local coordinates.

Since we are using the programmable pipeline now, our previous `GL_MODELVIEW` and `GL_PROJECTION` matrices no longer work and have to manipulate our vertices into clip space ourselves!

Lucky for us, GLSL provided built-in types for vectors, matrices, and their arithmetic. The only thing we have to do is get our matrices to the shader. This is done with uniforms. A **uniform variable** is a shader variable whose value **does not change** within a single draw call. In contrast, vertex attributes change with each individual vertex passing through a vertex shader. However, a uniform is a variable whose value is constant for all vertices passing through the shader in a particular **draw call**.

A *draw call* is one set of `glBegin()` and `glEnd()`, or one call to `glDrawElements` or one call to `glDrawArrays`. But, we don't use immediate mode (and thus `glBegin` and `glEnd`) with shaders. So we need the other two. But what are those functions and how do we use them? See the next section!

Returning to uniforms, our goal to first get the location of the uniform in the shader and then set its value before we make a draw call:

1. **Find** the uniform: we get the location of a uniform variable of name `varName` using `glGetUniformLocation(ProgramID, "varName")`. This returns a unique ID for the variable in the program. This find only needs to be done one time, after the program is linked.

2. **Set** the uniform: we use `glUniform*(uniformID, ...)`

Since there are many different variable types in GLSL, there are many different functions to set those uniforms.

- `glUniformXf` sends X number of floats to a uniform of type `vecX`
- `glUniformXi` sends X number of integers to a uniform of type `vecX`

- `glUniformMatrixXfv` sends an array of X^2 floats to a uniform of type `matX`.

Note that GLSL uses column-major matrices! Thus, an array like:

$$[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]$$

becomes a matrix like:

$$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

Thankfully, the `glm` library handles a lot of this for us.

With this shader:

```
in vec3 vertexPosition_modelspace;

uniform mat4 MVP;

void main() {
    gl_Position = MVP * vertexPosition_modelspace;
}
```

The following sets the uniform:

```
glm::mat4 Projection = glm::perspective(glm::radians(45.0f),
                                        screenW/screenH, 0.001f, 1000.0f);
glm::mat4 V = glm::lookAt(eye, center, up);
glm::mat4 M = glm::mat4(1.0f); //identity
glm::mat4 MVP = Projection * V * M;

GLuint MatrixID = glGetUniformLocation(ProgramID, "MVP");
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

10.13 Draw Calls

When we want to draw things without using immediate mode, we must use `glDrawArrays` or `glDrawElements`. We'll see the latter in the next lecture. `glDrawArrays` is used to draw many primitives based on the arrays specified as part of `glVertexAttribPointer`.

`glDrawArrays` takes three parameters:

- Primitive mode: lines, points, triangles, line strips, triangle strips, etc.
- The starting index from which to start reading data from the attribute pointers.
- The number of indices to be used for specifying primitives to be drawn.

Assume `glVertexAttribPointer` has already been specified. Then, we can draw 20 triangles, for example, using:

```
glDrawArrays(GL_TRIANGLES, 0, 20*3); //3 vertices per triangle
```

10.14 Revisiting Textures

We now know about the ideas of texture mapping, texture coordinates as vertex attributes, and shaders. Now let's use shaders to actually do texture mapping.

The specification of texture coordinates is rather simple and follows the pattern of any other vertex attribute in a vertex shader. The only difference is the type of a texture coordinate attribute is typically `vec2` for a 2D texture (i.e. a typical image).

The tricky things with textures is understanding all the different pieces which make up a texture.

- A **texture object** is an OpenGL object that is created on the server side, and a **unique ID** is returned to the client side. This is very similar to programs and shaders. `glGenTextures(int num, GLuint* IDs)` creates and allocates `num` texture objects and returns their IDs in the array `IDs`.
- A texture object has **texture storage** for storing that actual image data of the texture. However, we cannot directly access the storage of a texture object.
    1. **Bind** the texture object to a **texture target**. Possible targets are, `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, etc. depending on the dimensions of the image to load. This is done with `glBindTexture(textureTarget, textureID)`.
    2. **Load** the bitmap image into a texture target: `glTexImageXD` must use `GL_TEXTURE_XD` as its target, where `X` is one of 1, 2, 3.
    3. **Generate Mipmaps**. Just call `glGenerateMipmap(GL_TEXTURE_XD)`. Otherwise your texture won't work.
    4. After loading, one should **unbind** the texture object. Since OpenGL is a state machine, we want to avoid accidental side-effects of leaving a texture object bound. We unbind an object to a target by calling `glBindTexture(GL_TEXTURE_XD, 0)`
- **Texture unit**. A texture unit can be considered as a "texture slot". In an OpenGL program, there are least 80 texture slots named `GL_TEXTURE0`, `GL_TEXTURE1`, `GL_TEXTURE2`, etc. **Texture units are the *access point* of draw calls to texture data.**
    1. **Set which texture unit is active** by calling `glActiveTexture(GL_TEXTUREi)`.
    2. **Enable a texture target within the active unit**. We call `glEnable` with one of `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, etc. to get that texture enabled for subsequent draw calls.
    3. **Bind** a texture object to a texture target. We may have previously bound a texture object to a texture target in order to fill it with data, but now we have to bind it again

so that the draw calls can read the texture data.

4. Do your draw calls (details below).

5. **Unbind** the texture object.

To summarize, we have texture objects, texture storage, and texture units. To actually use a texture for texture mapping during drawing, we have to fill a texture object's texture storage with an image, enable a texture target within a texture unit, bind the texture object to that texture object and, finally, **read data from that texture unit** during drawing (which inevitably reads data from the bound texture object's texture storage).

Of course, reading data from the texture unit requires texture coordinates to know the coordinates of of the texture storage to read from. But how do we get access to the texture unit and the bound texture? GLSL uniforms and **samplers**.

A vertex shader should input and output texture coordinates as part of the vertex specification. After interpolation, each fragment will have its own texture coordinates. Now, the fragment shader should use those texture coordinates to actually read data from the texture. This is the job of the sampler.

```
#version 330 core

in vec2 uv2;
uniform sampler2D tex;

void main(){
    gl_FragColor = texture(tex, uv2);
}
```

In GLSL, we use the `texture()` function to read the color data from a texture's texture storage at a particular location. If the texture is 2D, we need 2D coordinates, etc. The uniform variable `tex` is actually a simple integer to describe **which texture unit** to read from. By default, it is 0.

If we want to read from a texture unit other than `GL_TEXTURE0`, we have to set the uniform in the shader appropriately. In our client code, we get the uniform location from the linked program object and then set its value:

```
GLuint texID = glGetUnformLocation(ProgramID, "tex");
glUniform1i(texID, i); // i is an integer corresponding to GL_TEXTUREi
```

Put it all together: see L10.cpp and L10texture.cpp

Note: reading texture data form different image formats is quite tricky. Let some other library for that for you. In CPP, I provide a `loadBMP()` function. In Python, you can use `PIL` (Python Imaging Library) and simply say `im = PIL.Image.open(imageFilePath)`