Lecture 10

The Graphics Pipeline
Caveat About Today’s Lecture

• Today’s focus is on **OpenGL**
  • The cross-platform graphics API for Indie games
  • **Vulkan** may take over, but not there yet

• CUGL uses **OpenGLES 3** for rendering
  • Is a proper subset of **OpenGL 3.x**
  • Designed with mobile devices in mind

• Much of what we say is true in other APIs
  • But the pipeline will be slightly different
  • In the case of Vulkan, a lot different
Graphics Cards Draw Triangles
Triangles Can Be Colored
Triangles Can Be Textured
Triangles Can Be Both
A Sprite is (Often) Two Triangles
Triangles are Drawn with Shaders

The Graphics Pipeline

- Vertex Data
- Vertex Shader
- Pixel Data
- Fragment Shader
- Image

Uniforms
Vertex Data Defines the Triangle

Position (Required)

(0,0)   (25,43)   (0,50)
Vertex Data Defines the Triangle

Position (Required)
Color (Optional)

(0,0) (0,0,1,1) (25,43) (1,0,0,1) (0,50) (0,1,0,1)
Vertex Shader **Interpolates** Pixels

Position (Required)

Color (Optional)

(0,0) (0,0,1,1)

(12,21) (0.49,0,0.48,1)

(25,43) (1,0,0,1)

(25,14) (0.33,0.33,0.33,1)

(0,50) (0,1,0,1)
A Very Simple Shader

Vertex Shader

```cpp
// Positions
in vec4 aPosition;

// Colors
in vec4 aColor;
out vec4 outColor;

uniform mat4 uCamera;

// Interpolate position and color
void main(void) {
    gl_Position = uCamera * aPosition;
    outColor = aColor;
}
```

Fragment Shader

```cpp
// The output color
out vec4 frag_color;

// Color result from vertex shader
in vec4 outColor;

// Just use color computed
void main(void) {
    frag_color = outColor;
}
```
**A Very Simple Shader**

### Vertex Shader

```glsl
// Positions
in vec4 aPosition;

// Colors
in vec4 aColor;
out vec4 outColor;
uniform mat4 uCamera;

// Interpolate position and color
void main(void) {
    gl_Position = uCamera * aPosition;
    outColor = aColor;
}
```

### Fragment Shader

```glsl
// The output color
out vec4 frag_color;

// Color result from vertex shader
in vec4 outColor;

// Just use color computed
void main(void) {
    frag_color = outcolor;
}
```
A Very Simple Shader

### Vertex Shader

```cpp
// Positions
in vec4 aPosition;

// Colors
in vec4 aColor;
out vec4 outColor;

// Interpolate position and color
void main(void) {
  gl_Position = uCamera * aPosition;
  outColor = aColor;
}
```

### Fragment Shader

```cpp
// The output color
out vec4 frag_color;

// Color result from vertex shader
in vec4 outColor;

// Just use color computed
void main(void) {
  frag_color = outColor;
}
```
Uniforms “Never” Change

- We *stream* vertex data to the shader
  - Put all vertex data into a giant array
  - Send it all to graphics card at once

- Changing a uniform *breaks the stream*
  - Have to break up the array into parts
  - Send one part with first value of uniform
  - Send next part with second value of the uniform

- This can *slow down the framerate*
  - Unlikely in this class unless lots of sprites
  - But should be aware of the cost
Uniforms “Never” Change

- We *stream* vertex data to the shader
  - Put all vertex data into a giant array
  - Send it all to graphics card at once

- Changing a uniform breaks the stream
  - Have to break up the array into parts
  - Send one part with first value of uniform
  - Send next part with second value of the uniform

- This can *slow down the framerate*
  - Unlikely in this class unless lots of sprites
  - But should be aware of the cost

Will the camera ever change?
Images Have Texture Coordinates

(0,0) (1,0)

(0,1) (1,1)
Vertex Data Can Include Texture Data

Position (Required)

Texture Coords (Optional)

(0,0)
(-0.37,1)
(25,43)
(0.5,-0.5)
(50,0)
(1.37,1)
Vertex Shader **Interpolates** Pixels

Position (Required)

Texture Coords (Optional)

(0,0)  (25,43)
(-0.37,1)  (0.5,-0.5)
(12,21)  (25,14)
(0.048,0.27)  (0.5,0.51)
(50,0)  (1.37,1)
A Texture Shader

**Vertex Shader**

```cpp
// Positions
in vec4 aPosition;

// Texture Coords
in vec4 aCoord;
out vec4 outCoord;

uniform mat4 uCamera;

// Interpolate position and coords
void main(void) {
    gl_Position = uCamera * aPosition;
    outCoord = aCoord;
}
```

**Fragment Shader**

```cpp
// The output color
out vec4 frag_color;

// Texture coord from vertex shader
in vec4 outCoord;

uniform sampler2D uTexture;

// Use texture to compute color
void main(void) {
    frag_color = texture(uTexture, outCoord);
}
```
A Texture Shader

**Vertex Shader**

```glsl
// Positions
in vec4 aPosition;

// Texture Coords
in vec4 aCoord;
out vec4 outCoord;

uniform mat4 uCamera;

// Interpolate position and coords
void main(void) {
  gl_Position = uCamera * aPosition;
  outCoord = aCoord;
}
```

**Fragment Shader**

```glsl
// The output color
out vec4 frag_color;

// Texture coord from vertex shader
in vec4 outCoord;
uniform sampler2D uTexture;

// Use texture to compute color
void main(void) {
  frag_color = texture(uTexture, outCoord);
}
```

```
texture + coord = color
```
A Texture Shader

**Vertex Shader**

```cpp
// Positions
in vec4 aPosition;

// Texture Coords
in vec4 aCoord;
out vec4 outCoord;

uniform mat4 uCamera;

// Interpolate position and coords
void main(void) {
    gl_Position = uCamera*aPosition;
    outCoord = aCoord;
}
```

**Fragment Shader**

```cpp
// The output color
out vec4 frag_color;

// Texture coord from vertex shader
in vec4 outCoord;

uniform sampler2D uTexture;

// Use texture to compute color
void main(void) {
    frag_color = texture(uTexture, outCoord);
}
```
A Texture Shader

**Vertex Shader**

```glsl
// Positions
in vec4 aPosition;

// Texture Coords
in vec4 aCoord;

out vec4 outCoord;

uniform mat4 uCamera;

// Interpolate position and coords
void main(void) {
    gl_Position = uCamera * aPosition;
    outCoord = aCoord;
}
```

**Fragment Shader**

```glsl```
// The output color
out vec4 frag_color;

// Texture coord from vertex shader
in vec4 outCoord;

uniform sampler2D uTexture;

// Use texture to compute color
void main(void) {
    frag_color = texture(uTexture, outCoord);
}
```

Changing the texture **stalls** the stream
How Does a SpriteBatch Work?

- SpriteBatch has a **shader**
- Methods create vertices
- Vertices have **color, texture**
- Sends vertices to shader

- Groups data by **uniforms**
- Adds all vertices to a set
- Breaks set into **batches**
- Uniforms fixed each batch

- Each texture is a **new batch**
- How often do you switch?
How Does a SpriteBatch Work?

- SpriteBatch has a **shader**
  - Methods create vertices
  - Vertices have **color, texture**
  - Sends vertices to shader

- Groups data by **uniforms**
  - Adds all vertices to a set
  - Breaks set into **batches**
  - Uniforms fixed each batch

- Each texture is a **new batch**
  - How often do you switch?
How Does a SpriteBatch Work?

- SpriteBatch has a **shader**
- Methods create vertices
- Vertices have **color, texture**
- Sends vertices to shader

- Groups data by **uniforms**
- Adds all vertices to a set
- Breaks set into **batches**
- Uniforms fixed each batch

- Each texture is a **new batch**
- How often do you switch?
How Does a SpriteBatch Work?

- SpriteBatch has a **shader**
- Methods create vertices
- Vertices have **color**, **texture**
- Sends vertices to shader

- Groups data by **uniforms**
- Adds all vertices to a set
- Breaks set into **batches**
- Uniforms fixed each batch

- Each texture is a **new batch**
- How often do you switch?
Optimizing Performance: Atlases

- **Idea:** Never switch textures
  - Sprite sheet is many images
  - We can draw part of texture
  - One texture for everything?

- Called a **texture atlas**
  - Supported in CUGL
  - See file `loading.json`
  - Ideal for **interface design**

- Has some **disadvantages**
  - Textures cannot repeat
  - Recall texture size limits
Aside: This is How Fonts Work

- Each Font creates an atlas
  - Reason you must specify size
  - Atlas limited to 512x512
  - Multiple atlases if necessary
- TextLayout makes vertices
  - Quads made from font metrics
  - Includes kerning, alignments
  - Vertices include texture cords
- This makes text very fast
  - Generating vertices is quick
  - Actual font cached in atlas(es)
The SpriteBatch Shader

- Provides support for
  - Solid/vertex colors
  - Color gradients (linear, radial)
  - Textures/texture coords
  - Gaussian blur
  - Scissoring/masking

- Not “user-serviceable”
  - Do not try to replace this
  - Will break all the UI code

- Want a custom shader?
  - Make a new pipeline
The Shader Class

- `Shader::alloc(const string vsrc, const string fsrc)`
  - Returns `nullptr` if shader compilation fails
  - Also gives helpful error message in output

- The shaders are **strings**, not **files**
  - You could load files and read into strings
  - But this means pipeline *waits* on asset loading
  - Better to put directly in your source code

- CUGL approach: **raw strings**
  - Write shader code into a header file
  - Special include assigns contents to a variable
Using a Shader Object

• Activate it with `bind()` command
  • Can only have one shader at a time
  • This method makes it the active shader
  • Call `unbind()` to release it.
  • Like begin/end with SpriteBatch

• Assign **uniforms** to shader with **setters**
  • `s->setUniformMat4("uCamera",cam->getCombined());`
  • Support for primitives and all CUGL math objects
  • Applies to both vertex and fragment uniforms
  • But not texture; that is special
Make a Vertex Type

- Can be **any class** of your making
  - Should have **position** (Vec2, Vec3, or Vec4)
  - Can have anything else that you want
  - There are (almost) no restrictions

**Example**: `SpriteVertex2`
- Position (Vec2)
- Color (unsigned int)
- Texture coords (Vec2)
- Gradient coords (Vec2)
Create a Geometry

- Need two things to **define shape**
  - An array of vertices
  - An array of indices
- Indices refer to **array positions**
  - Used to create triangles
  - Meaning depends on command
- **Poly2** does all of this for you!
  - But it only has position data
  - Only supports triangle lists
- For more, see class **Mesh<T>**
Create a VertexBuffer Object

- `VertexBuffer::alloc(sizeof(VertexClass))`
  - `sizeof` tells it number of bytes per vertex
  - Stream size is determined when you **load** vertices

- `v->setupAttribute("var",bytes,type1,type2,loc)`
  - Maps shader variable to slot in vertex class
  - See documentation/example for how to do this

- `v->attach(shader)`
  - Tell vertex buffer to send data to the shader
  - This is how the shader gets the vertex data!
VertexBuffer vs Shader

Have a **many-one** relationship
**VertexBuffer vs Shader**

Set active VertexBuffer with bind/unbind

Have a **many-one** relationship
Loading Data Into Vertex Buffer

- \texttt{v->loadVertexData(array, size)}
  - Loads the array of vertices
  - Remembers until you load new data

- \texttt{v->loadIndexData(array, size)}
  - Loads the array of indices
  - Should be updated when the vertices are

- \texttt{v->draw(command, index\_count, index\_start)}
  - Tells how to interpret the indices (list, strip, fan)
  - Does the actual drawing at this time (not delayed)
### Aside: Static Draw vs Stream Draw

#### Static Draw
- Vertex buffer is **fixed**
  - Object altered via *uniforms*
  - **Example**: Transform matrix
- Used if **lots of vertices**
  - Uniform changes stall drawing
  - But reloading vertices is worse
- Common in **3d rendering**
  - Models are *large meshes*
  - Each model its own buffer

#### Stream Draw
- Vertex buffer **changes often**
  - Always updating position
  - Always updating geometry
- Used if **low complexity**
  - Few vertices per object (quads)
  - Can’t give each sprite a buffer
- Common in **2d rendering**
  - Data is very *heterogeneous*
  - How SpriteBatch works
Last Step: Textures

- Textures are **not** set by a shader method
  - Data is way too big for normal uniforms
  - All data is stored in a `Texture` object

- This object has its own `bind/unbind`
  - Call `bind` to make it the **active texture**
  - Call `unbind` to remove it/have no texture

- Possible to have **more than one texture**
  - Each shader texture variable has a slot (0-10)
  - Can call `bind(slot)` to put it in a slot
Putting It All Together

shader->bind();
vbuffer->bind(); // Binds shader if necessary
texture->bind(); // Make active texture in slot 0
vbuffer->draw(mesh.command,mesh.indices.size(),0);
...
// More drawing commands
texture->unbind(); // If need to change texture
...
// More drawing commands
vbuffer->unbind(); // If need to change buffer
shader->unbind(); // If need to change shader
Putting It All Together

shader->bind();
vbuffer->bind(); // Binds shader if necessary
texture->bind(); // Make active texture in slot 0
vbuffer->draw(mesh.command, mesh.indices.size(), 0);
... // More drawing commands
texture->unbind(); // If need to change texture
... // More drawing commands
vbuffer->unbind(); // If need to change buffer
shader->unbind(); // If need to change shader
void CustomNode::draw(const std::shared_ptr<SpriteBatch>& batch, const Affine2& transform, Color4 tint) {

    // Stop the previous graphics pipeline
    batch->end();

    // Adjust pipeline camera by the node transform
    Mat4 camera = _scene->getCombined()*transform;

    // Custom drawing code
    ...
    ...

    // Restart the sprite batch
    batch->begin(_scene->getCombined());
}
Two Final Classes

**UniformBuffer**
- Used if many uniforms
  - Setting each uniform slow
  - Put uniforms in byte array
  - Set pointer to byte array
- Permits uniform streaming
  - Dual of VertexBuffer
- Used by SpriteBatch
  - Holds gradients, scissors
  - See code for usage

**RenderTarget**
- Used to render offscreen
  - Draw to a special buffer
  - Turn buffer into a texture
  - Apply texture to shapes
- Great for special effects
  - Render screen to texture
  - Apply 2nd shader to texture
- Used in Scene2Texture
  - See documentation
Summary

- CUCL uses **OpenGLES 3** for rendering
  - Uses shaders to produces triangles on screen
  - **SpriteBatch** makes all of this very easy

- Custom shaders require a **separate pipeline**
  - Need a **Shader** to output to screen
  - Need a **Mesh** to define the geometry
  - Need a **VertexBuffer** to pass Mesh to Shader
  - (Optional) Need a **Texture** to fill in triangles

- Want more? Take **CS 5625**
Advanced Technique
Triangles Have Hard Edges
Sometimes Want Softer Edges
Sometimes Want Softer Edges

OpenGLES does NOT support multisampling
Extrude The Triangle Boundary
Extrude The Triangle Boundary
Use Alpha to Fade Out Extrusion

Alpha = 255 (opaque)

Alpha = 0 (transparent)
Use Alpha to Fade Out Extrusion

Alpha = 255 (opaque)

Alpha = 0 (transparent)

See Pipeline Demo