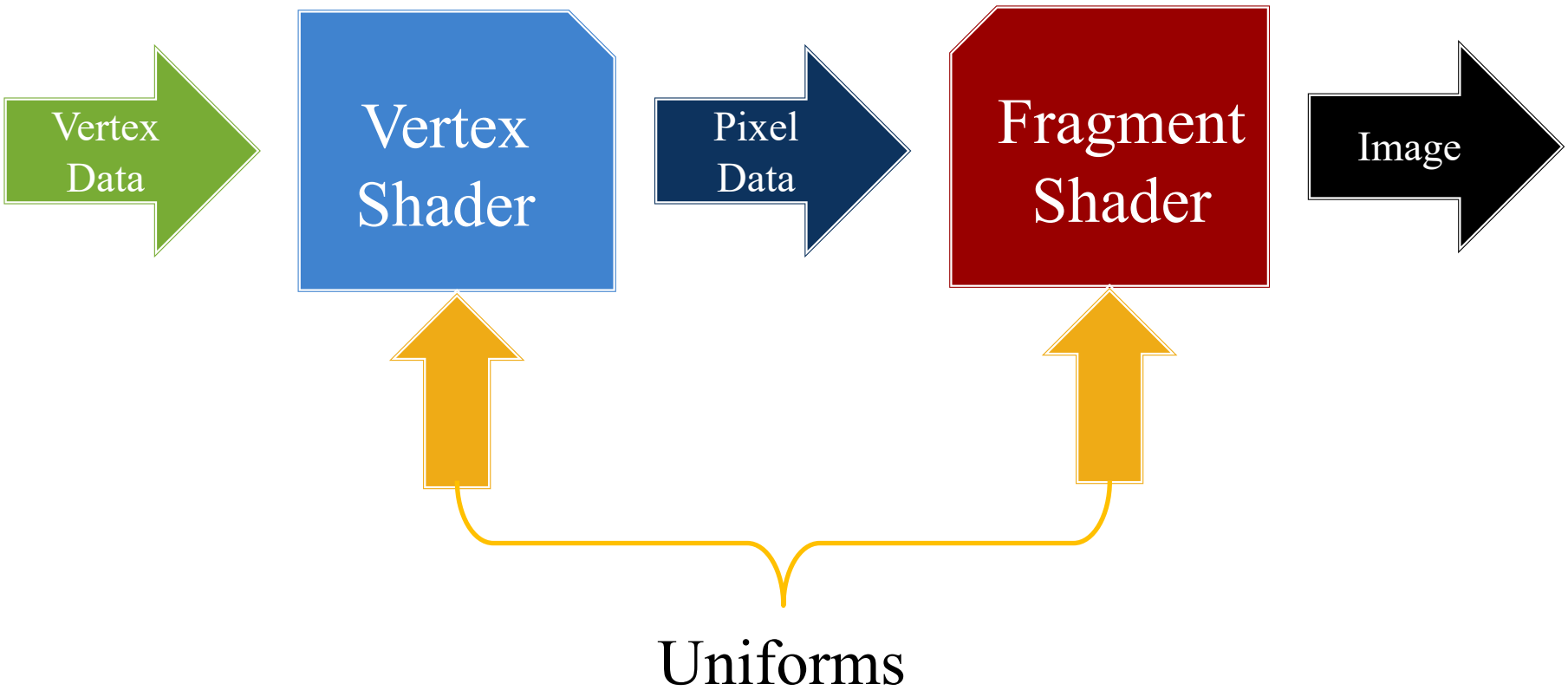
the gamedesigninitiative at cornell university

Lecture 12

Custom Pipelines

Recall: Graphics Shaders



Recall: Shader Basics

Vertex

- Should contain
  - An `in` for each **attribute**
  - An `out` for each **interpolant**
  - A `uniform` (?) for **camera**
  - A `main()` function
- Anything else is optional

Fragment

- Should contain
  - An `in` for each **interpolant**
  - An `out` for the **pixel color**
  - A `uniform` (?) for **texture**
  - A `main()` function
- Anything else is optional

Linking requires interpolants match

A Color-Texture Shader (GLSL 3.0)

Vertex Shader

```
// Attributes
in vec4 aPosition;
in vec4 aColor;
in vec4 aCoord;
// Interpolants
out vec4 outColor;
out vec4 outCoord;

uniform mat4 uCamera;

void main(void) {
   gl_Position = uCamera*aPosition;
   outColor = aColor;
   outCoord = aCoord;
}
```

Fragment Shader

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

// Interpolants
in vec4 outColor;
in vec4 outCoord;

uniform sampler2D uTexture;

void main(void) {
   frag_color = texture(uTexture,
                        outCoord);

   frag_color *= outColor;
}
```

Some Nomenclature Issues

- We are using **shader** to mean **vertex** + **fragment**
  - Some people use shader to refer just one of these
  - Combining them together is often called a **pipeline**
- But only graphics programmers talk like this
  - Hobbyists typically use **shader** to mean **pipeline**
- In CUGL we use the following terminology
  - `ShaderSource` is single GLSL program
  - `GraphicsShader` is a pipeline combination
  - This thinking is motivated by Vulkan

The ShaderSource Class

- Meant to be on the stack: no alloc
  - initOpenGL(const string source, ShaderStage stage)
  - initVulkan(const vector<byte>& bytes, ShaderStage stage)
- In OpenGL, the source is a **string**, not a **file**
  - You could load files and read into strings
  - But this means pipeline *waits* on asset loading
- CUGL approach: **raw strings**
  - Write shader code into a header file
  - Special include assigns contents to a variable

The ShaderSource Class

- Meant to be on the stack: no alloc
  - initOpenGL(const string source, ShaderStage stage)
  - initVulkan(const vector<byte>& bytes, [illegible]ge)
- In O[illegible]ource is a **string**,
  - Yo[illegible]es and read into strings
  - But this means pipeline *waits* on asset loading
- CUGL approach: **raw strings**
  - Write shader code into a header file
  - Special include assigns contents to a variable

Not yet supported

Vertex or Fragment

Ready to go at start-up

C++ Raw Strings

- Raw strings are multiline strings
  - Work like """…""" in Python
  - Great for inlining interpreted code
- Delimit as R"(…)"
  - Parentheses are **not** part of string
  - Often put string in a header file
- Header usage

```
const std::string rawString =
#include "rawstring.h"
;
```

Assigns contents of file to string

The GraphicsShader Class

- Call alloc with ShaderSource (vertex & fragment)
- But this does not complete the shader set-up!
  - You need to **declare** all variables in the shaders
  - That creates data structures to use the shader
- There are three declaration types
  - AttributeDef: Declare an **attribute**
  - UniformDef: Declare a simple **uniform/push constant**
  - ResourceDef: Declare a **texture** or **uniform buffer**
- When done, compile the shader (this step can **fail**)

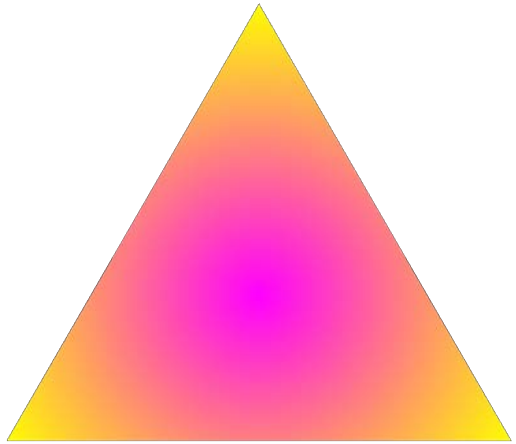
The GraphicsShader Class

- Call alloc with ShaderSource (vertex & fragment)
- But this does not complete the shader set-up!
  - You need to **declare** all variables in the shaders
  - That creates data structures to use the shader
- There are three declaration types
  - Attrib
  - Unifor tant
  - Resou r
- When done, compile the shader (this step can **fail**)

See the Pipeline Demo

Recall: 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**: `SpriteVertex`
  - Position (`Vec2`)
  - Color (`unsigned int`)
  - Texture coords (`Vec2`)
  - Gradient coords (`Vec2`)



Declaring Attributes in GraphicsShader

```
AttributeDef attrib;
attrib.location = 0;
attrib.group = 0;
attrib.type = GLSLType::VEC2;
attrib.offset = offsetof(SpriteVertex,position);
shader->declareAttribute("aPosition", attrib);

attrib.location = 1;
attrib.type = GLSLType::UCOLOR;
attrib.offset = offsetof(SpriteVertex,color);
shader->declareAttribute("aColor", attrib);
```

Declaring Attributes in GraphicsShader

```
AttributeDef attrib;
attrib.location = 0;
attrib.group = 0;
attrib.type = GLSLType::VEC2;
attrib.offset = offsetof(SpriteVertex,position);
shader->declareAttribute("aPosition", attrib);

attrib.location = 1;
attrib.type = GLSLType::UCOLOR4;
attrib.offset = offsetof(SpriteVertex,color);
shader->declareAttribute("aColor", attrib);
```

Shader Type

Position in C++ class

Name in shader

Declaring Attributes in GraphicsShader

```
AttributeDef attrib;
attrib.location = 0;
attrib.group = 0;
attrib.type = GLSLType::VEC2;
attrib.offset = offsetof(SpriteVertex,position);
shader->declareAttribute("aPosition", attrib);

attrib.location = 1;
attrib.type = GLSLType::UCOLOR;
attrib.offset = offsetof(SpriteVertex,color);
shader->declareAttribute("aColor", attrib);
```

Can reuse declaration

Declaring Attributes in GraphicsShader

```
AttributeDef attrib;
attrib.location = 0;
attrib.group = 0;
attrib.type = GLSLT
attrib.offset = offsetof(SpriteVertex,position);
shader->declareAttribute("aPosition", attrib);

attrib.location = 1;
attrib.type = GLSLType::UCOLOR;
attrib.offset = offsetof(SpriteVertex,color);
shader->declareAttribute("aColor", attrib);
```

Location in shader

Mesh "instance"

CUGL Meshes

- The type is `Mesh<T>` (in cugl::graphics)
  - Type `T` must be a vertex type
  - Also includes the drawing geometry
- This is not a standard game mesh
  - Stores data on the **CPU side**
  - Allows you to make lots of little changes
- True meshes are `VertexBuffer` objects
  - Stores the drawing data on **GPU side**
  - Designed to be updated once a frame

The VertexBuffer Class

- VertexBuffer::alloc(size,sizeof(VertexClass),access)
  - size is the maximum number of vertices
  - sizeof tells it number of bytes per vertex
  - access indicates how often we update it
- Load data from Mesh into the VertexBuffer
  - verts->loadData(mesh.vertices.data(), _mesh.vertices.size())
  - Each **attribute group** gets separate Mesh/VertexBuffer
- There is also an associated IndexBuffer
  - **Optional:** not required to use VertexBuffer
  - But only one; applies to all VertexBuffer objects

Aside: VertexBuffer Access Rules

Static

- 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

Dynamic

- 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

Using a GraphicsShader

```
shader->setDrawMode(mesh.command);

shader->setVertices(0, vertbuff);
shader->setIndices(indxbuff);

shader->setTexture("uTexture",texture);

shader->begin();

shader->pushInt("uType", type);
shader->pushMat4("uPerspective", camera->getCombined());

shader->drawIndexed(mesh.indices.size());

shader->end();
```

Using a GraphicsShader

```
shader->setDrawMode(mesh.command);

shader->setVertices(0, vertbuff);
shader->setIndices(indxbuff);

shader->setTexture("uTexture",texture);

shader->begin();

shader->pushInt("uType", type);
shader->pushMat4("uPerspective", camera->getC

shader->drawIndexed(mesh.indices.size());

shader->end();
```

Pass data to the GPU

SpriteBatch-style drawing pass

Using a GraphicsShader

```
shader->setDrawMode(mesh.command);

shader->setVertices(0, vertbuff);
shader->setIndices(indxbuff);

shader->setTexture("uTexture",texture);

shader->begin();

shader->pushInt("uType", type);
shader->pushMat4("uPerspective", camera->getC

shader->drawIndexed(mesh.indices.size());

shader->end();
```

Only need to be set once

Must be called every frame

Using a GraphicsShader

```
shader->setDrawMode(mesh.command);

shader->setVertices(0,
shader->setIndices(ind

shader->setTexture("uTexture",texture);

shader->begin();

shader->pushInt("uType", type);
shader->pushMat4("uPerspective", camera->getCombined());

shader->drawIndexed(mesh.indices.size());

shader->end();
```

Resource Uniform

Pushed Uniforms

Resources: UniformBuffer

- Vulkan limits **push constants** to 196 bytes
  - Just enough to hold three matrices
  - Effectively camera, modelview, and normal
  - OpenGL does not enforce this, but also a good idea
- Larger uniform data should be in a UniformBuffer
  - Similar format to a VertexBuffer
  - Same access controls and data loading
  - Difference is that it applies to all vertices
- Unlike push constants, attach to shader once

Pipeline Settings

- A graphics shader has other non-uniform values
  - **Examples**: Color blending, stencil support, depth testing
  - These are called **pipeline settings**
  - They are very specific to the backend API
- CUGL is designed to be backend agnostic
  - `BlendMode`: How to blend colors together
  - `StencilMode`: Use a stencil to constrain drawing
  - `CullMode`: Whether to eliminate backwards-facing triangles
  - `FrontFace`: How the "front" of a triangle is deteremined

Color Blending

- `BlendMode` is the most important setting for 2d
  - Defines how images are composited together
  - `BlendMode::ALPHA` is often used for alpha blending
- But the CUGL default is `BlendMode::PREMULT`
  - Premultiplied means color is multiplied by alpha
  - Clear **must** be (0,0,0,0); (1,1,1,0) is impossible
  - Why? Because that is how SDL3 loads images
- If you use `BlendMode::ALPHA`, edges will look **black**
  - Because transparency fades to black
  - `BlendMode::PREMULT` adjusts for this and fixes it.

Color Blending Images in SDL3

BlendMode::ALPHA

BlendMode::PREMULT

Pipeline Demo

Recall: Texture Atlases

- CUGL support
  - Specify in asset directory
  - Subentry of texture
- **Example**

```
"progress": {
    "file" : "progress.png",
    "atlas" : {
        "back" : [0,    0, 320, 45],
        "fore"  : [24,  45, 296, 90],
        "left"   : [0,   45,  24, 90],
        "right" : [296, 45, 320, 90]
    } }
```

- But the class is Texture!

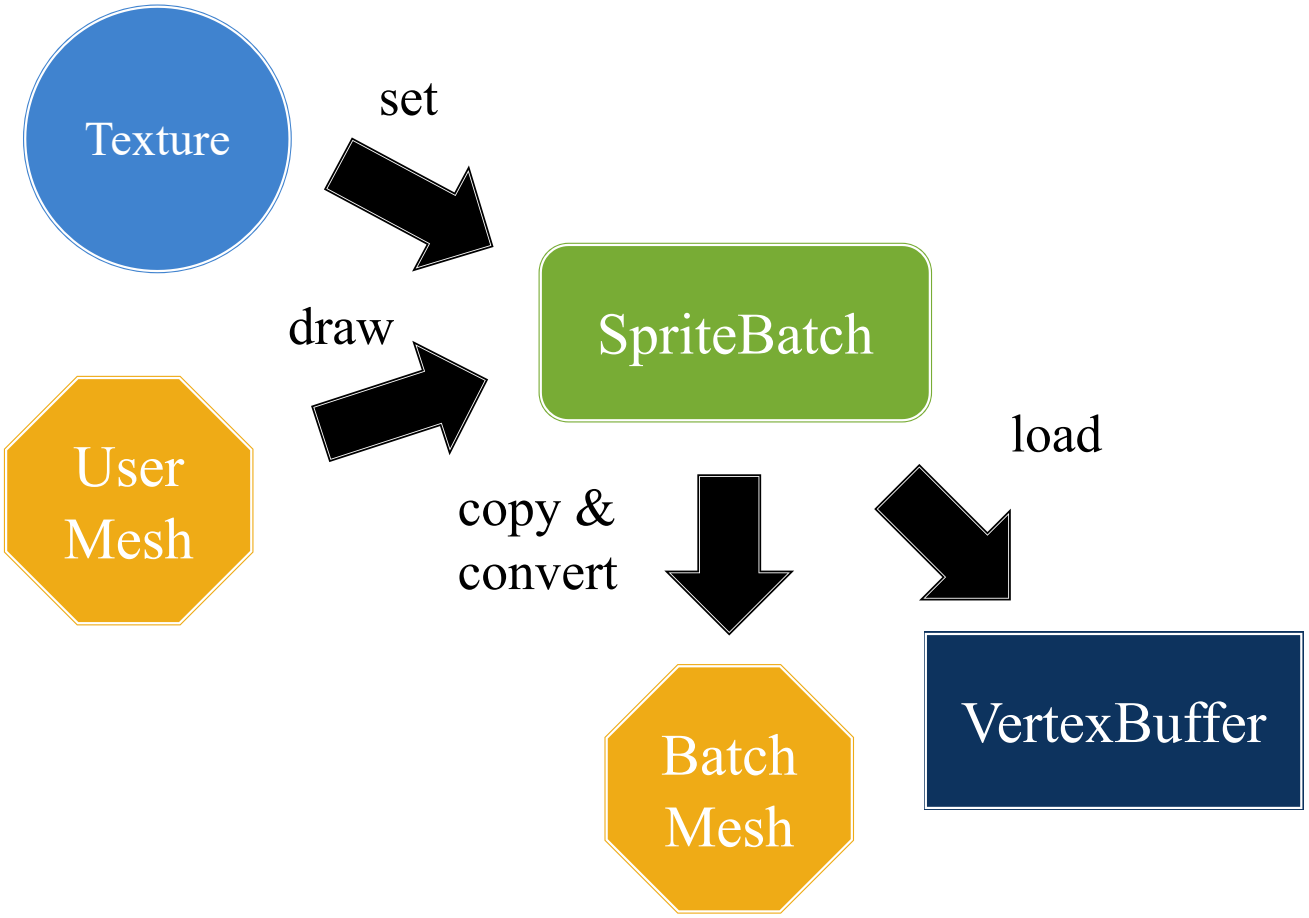
Subtextures

- `sub = texture->getSubtexture(minS,minT,maxS,maxT);`
  - Parameters are in **texture coordinates** not pixels
  - Returns a texture (`sub`) spanning that area
- **Analogy:** `TextureRegion` in LibGDX
  - That was a separate class that did same thing
  - But it forced us to "double" all methods
  - So CUGL just uses one class: `Texture`
- However, does not work the way you think
  - Passing it to a texture sends the **whole** texture
  - You have to adjust your texture coordinates

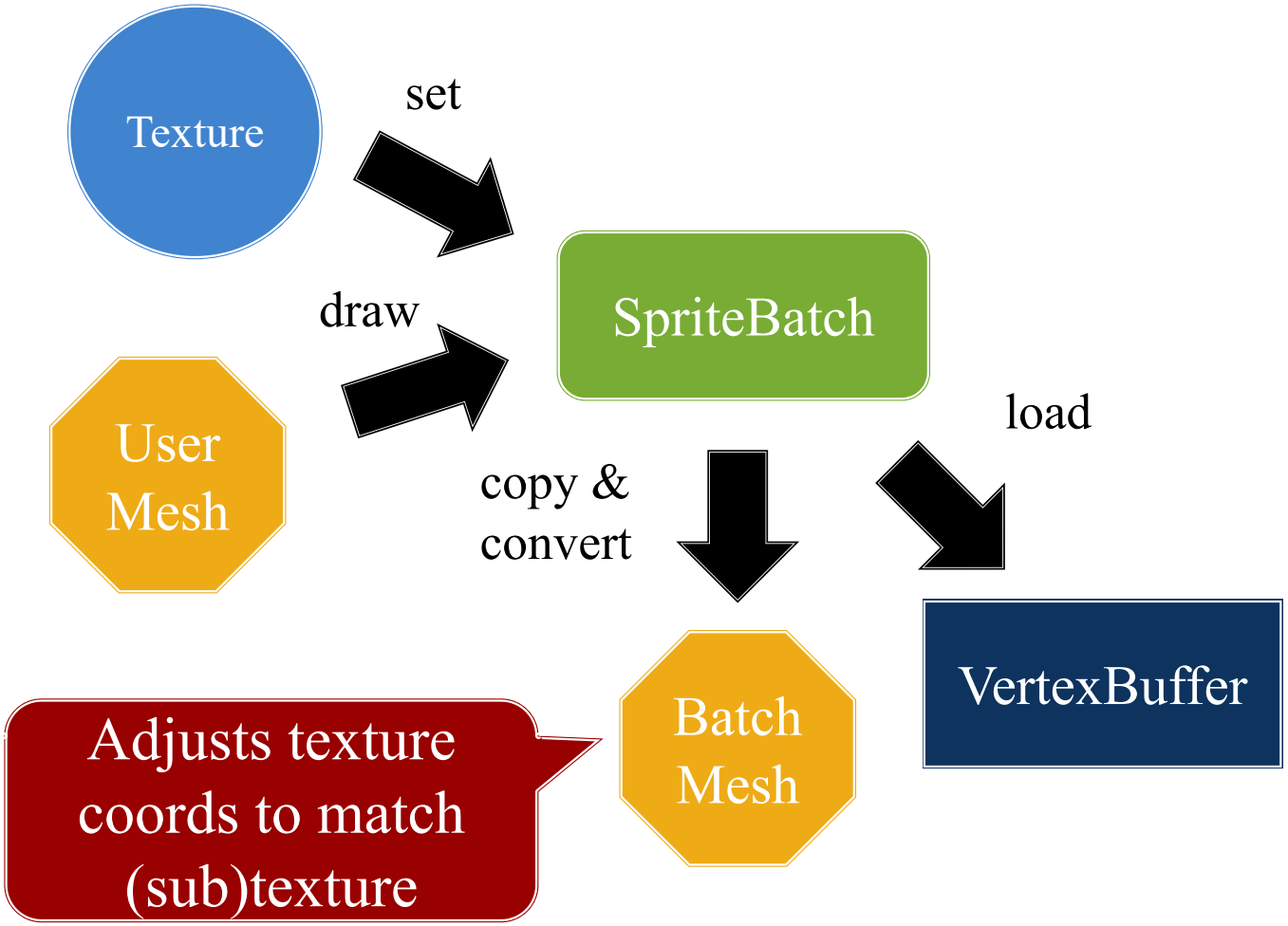
Using a Subtexture

- Use isSubtexture() to check if a subtexture
- Has getter methods for the texture range
  - getMinS(), getMaxS(), getMinT(), getMaxT()
  - By default, the values are 0,1,0,1
- Must convert mesh texture coordinates (s,t)
  - s => getMinS()+s*(getMaxS()-getMinS())
  - t => getMinT()+t*(getMaxT()-getMinT())
- This is how SpriteBatch works

How SpriteBatch Works



How SpriteBatch Works



So When Do We Do This?

- Do not touch the SpriteBatch shader
  - It is not "user serviceable"
  - You will break the UI code
- There are two classic options
  - Make a shader for a **custom scene graph node**
  - Make a shader for **post-processing an image**
- Anything more is beyond the scope of course
  - Covered in **CS 5625:** Interactive Graphics
  - Might be offered next year

Pipelines and Scene Graphs

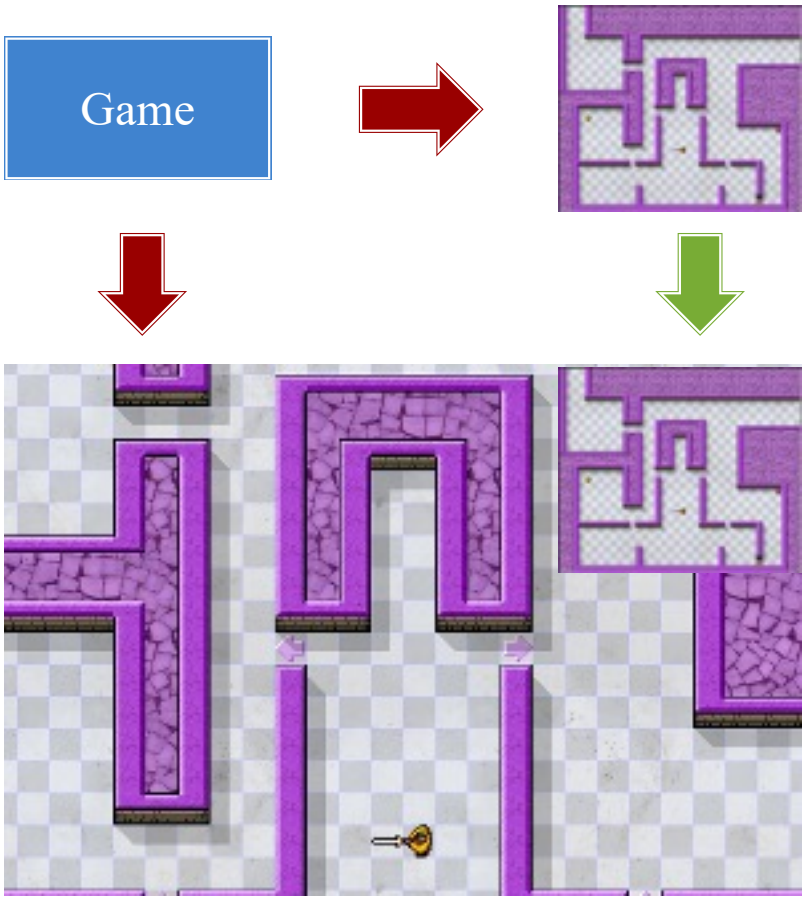
```
void CustomNode::draw(const std::shared_ptr<SpriteBatch>& batch,
                      const Affine2& transform, Color4 tint) {

    // 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();
}
```

Pipeline and Postprocessing

Pipeline and Postprocessing

Offscreen Rendering

- Can create *dynamic* textures
  - Draw an image to a texture
  - Draw that texture to screen
  - Called *offscreen rendering*
- Applications
  - Mini-maps
  - Scene Transitions
  - Light & Shadow
- Made easy in CUGL
  - Class `FrameBuffer`
  - Uses begin/end pattern

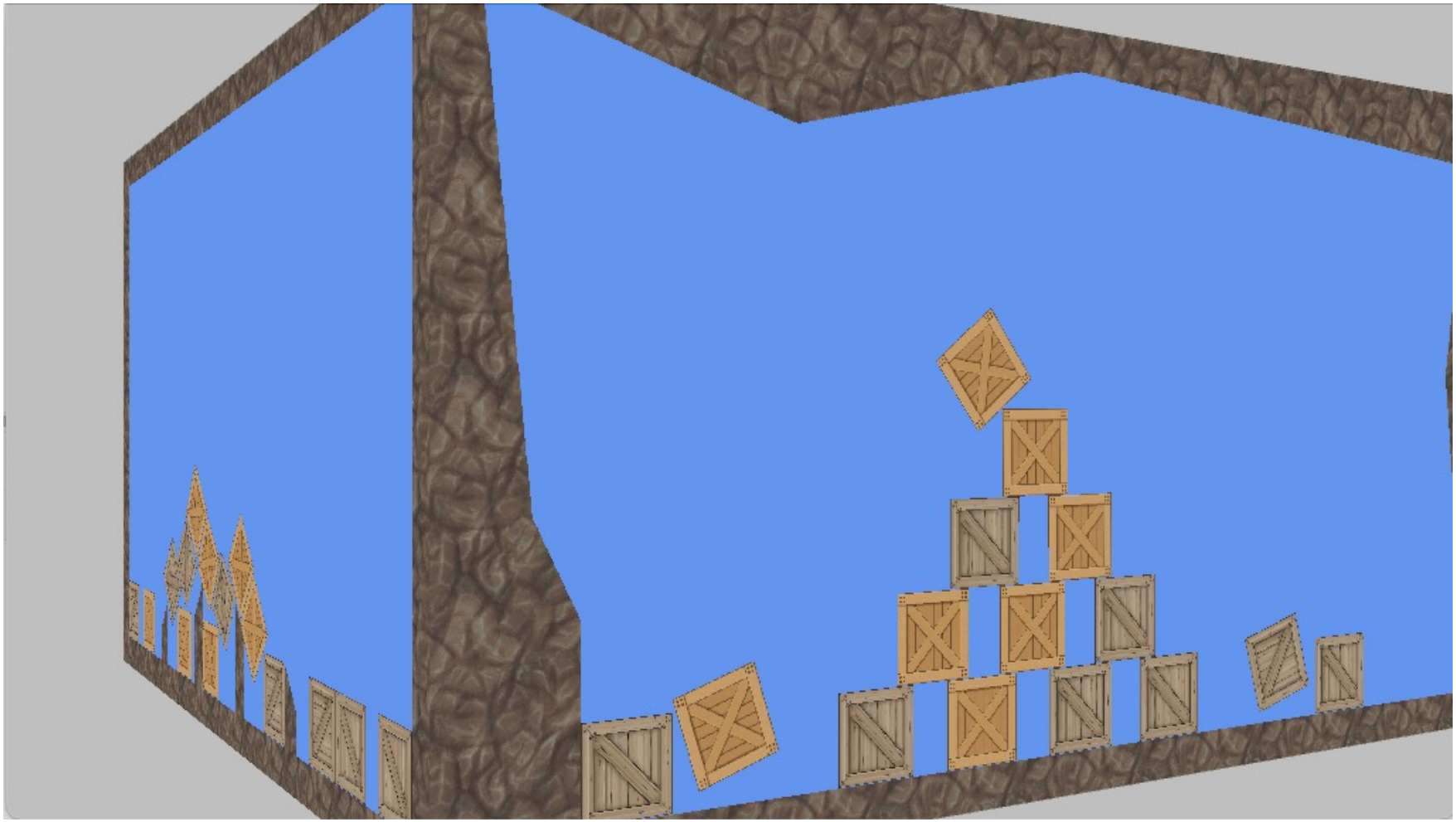


Using the FrameBuffer Class

```
// Set the resolution of the texture
auto target = FrameBuffer::alloc(width, height)

// This indicates this pass if offscreen
batch->begin(target);

...

batch->end();

auto texture = target->getTexture();

// Now use texture to draw to screen
```

GraphicsShader **and** SpriteBatch

Example: Cube Transitions

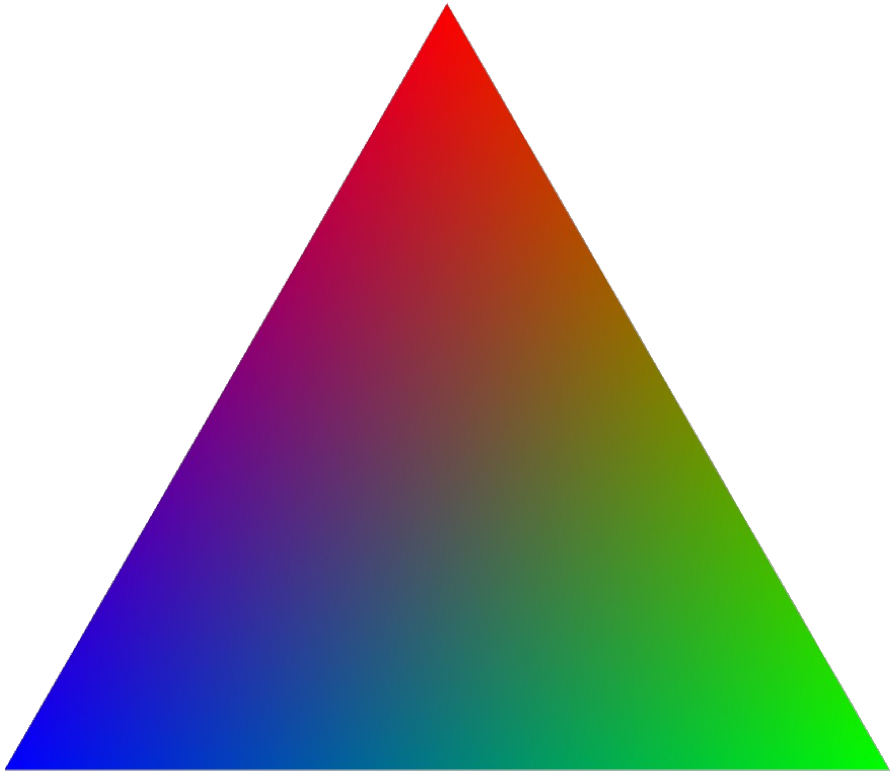


Summary

- Custom shaders require a **separate pipeline**
  - Need a `GraphicsShader` to output to screen
  - Need a `Mesh` to define the geometry
  - Need a `VertexBuffer/IndexBuffer` to pass mesh to shader
  - (Optional) Need a `Texture` to fill in triangles
- Custom pipelines enable postprocessing
  - Use `FrameBuffer` to draw to a texture
  - Create custom `GraphicsShader` to filter texture
- Want more? Take **CS 5625**

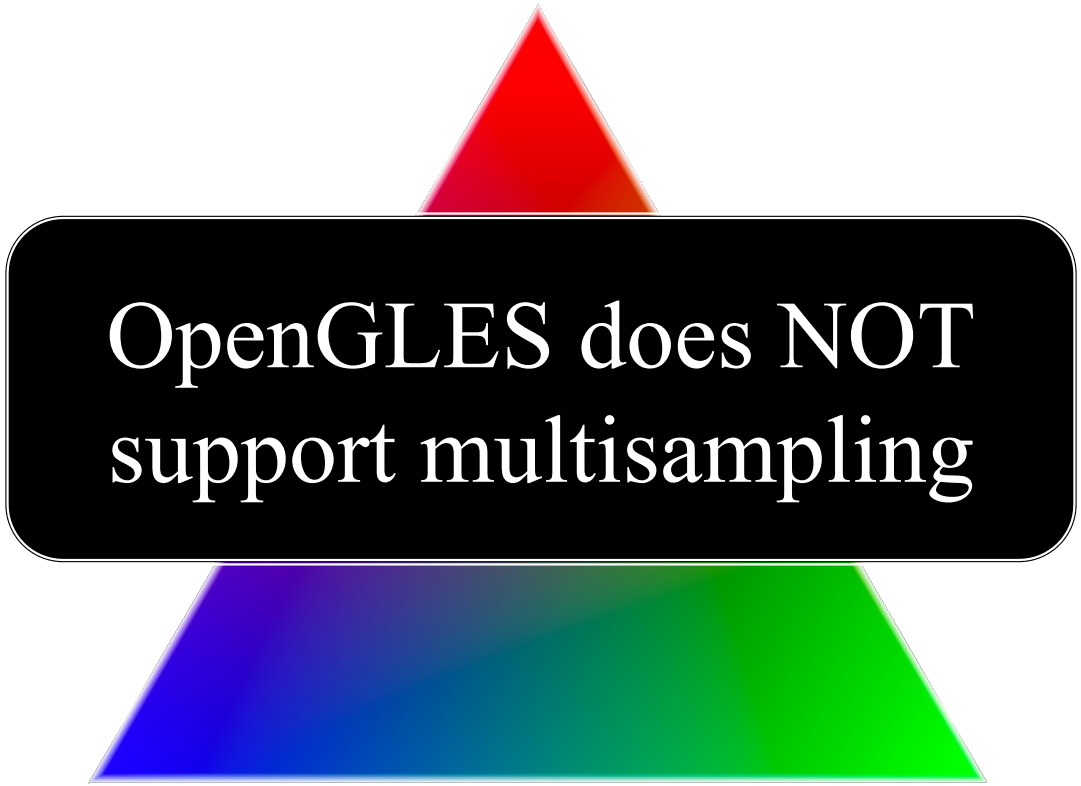
Additional Technique

Triangles Have Hard Edges

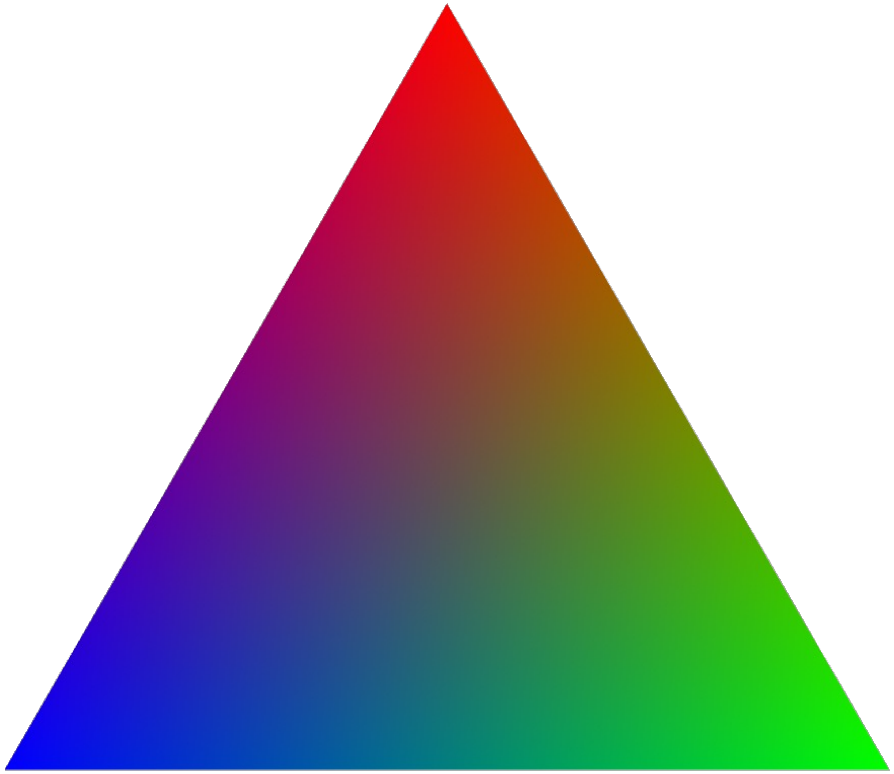


Sometimes Want Softer Edges

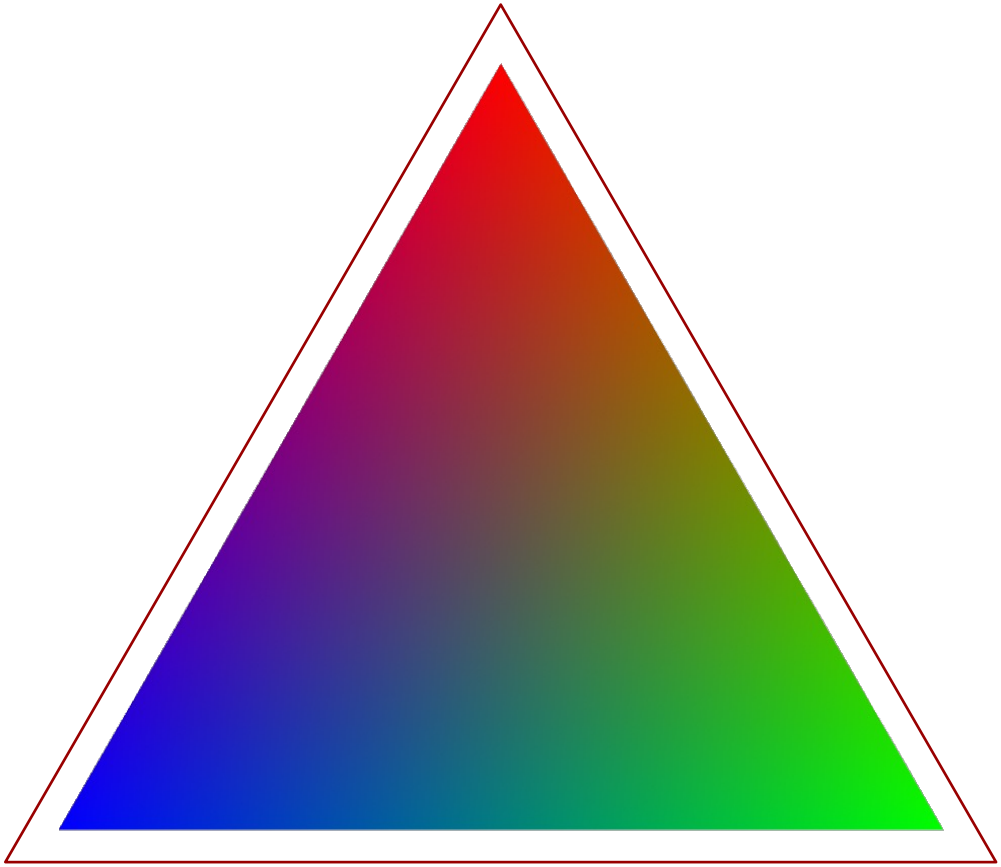
Sometimes Want Softer Edges



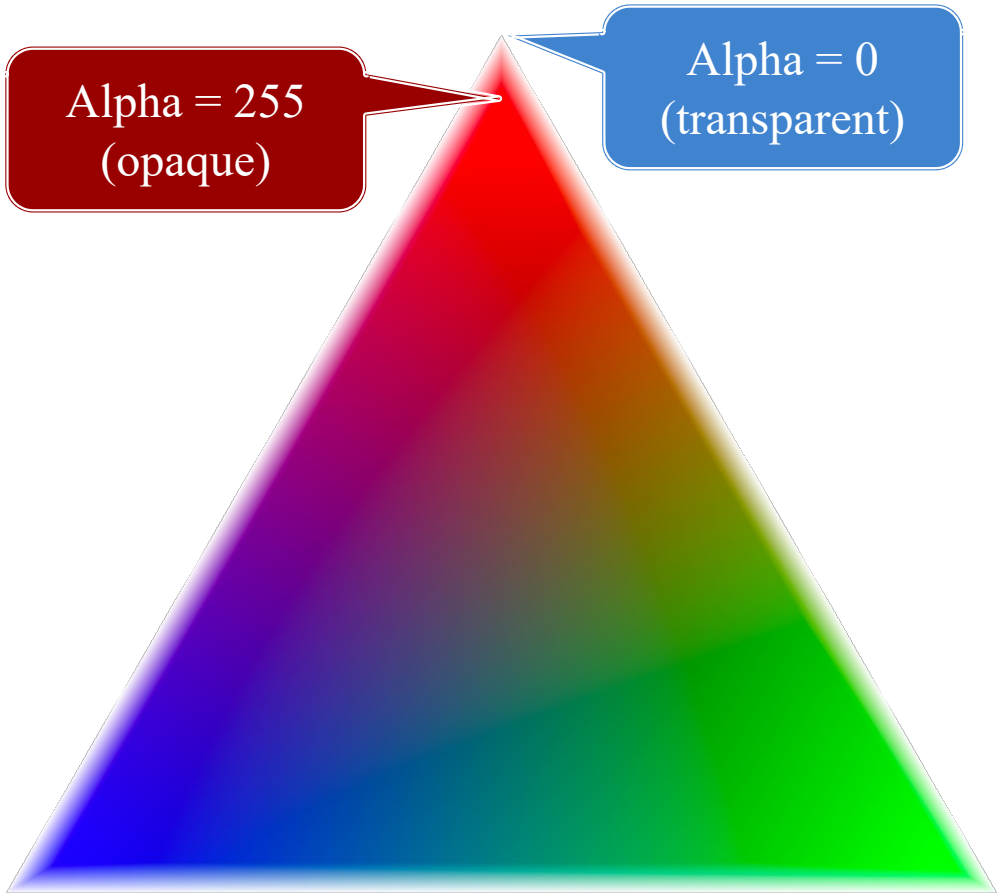
Extrude The Triangle Boundary



Extrude The Triangle Boundary



Use Alpha to Fade Out Extrusion



Use Alpha to Fade Out Extrusion