the gamedesigninitiative at cornell university

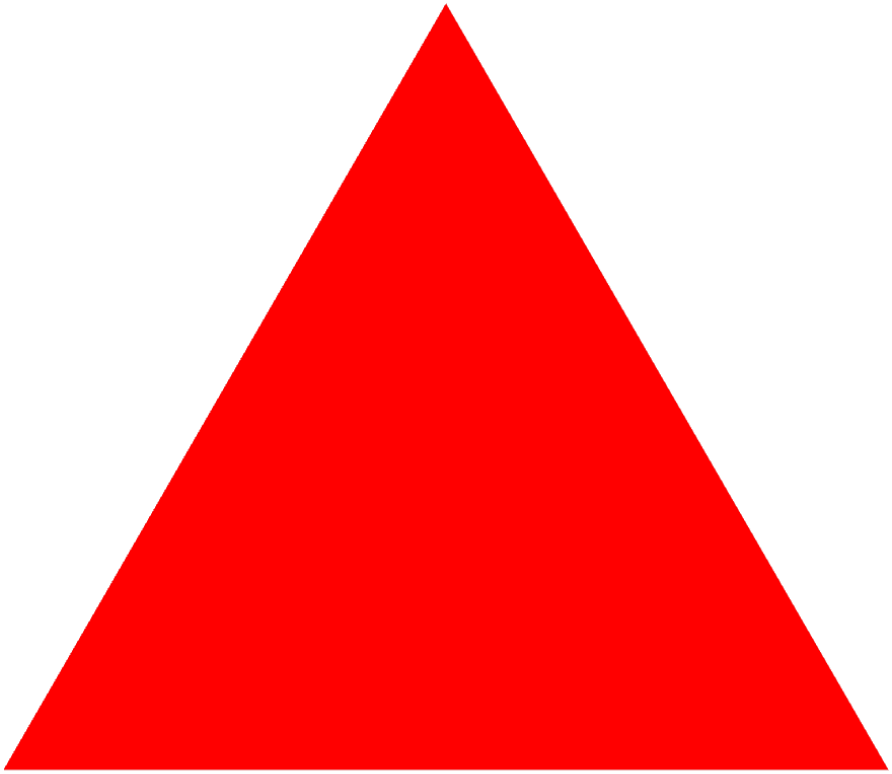
Lecture 11

The Graphics Pipeline (Overview)

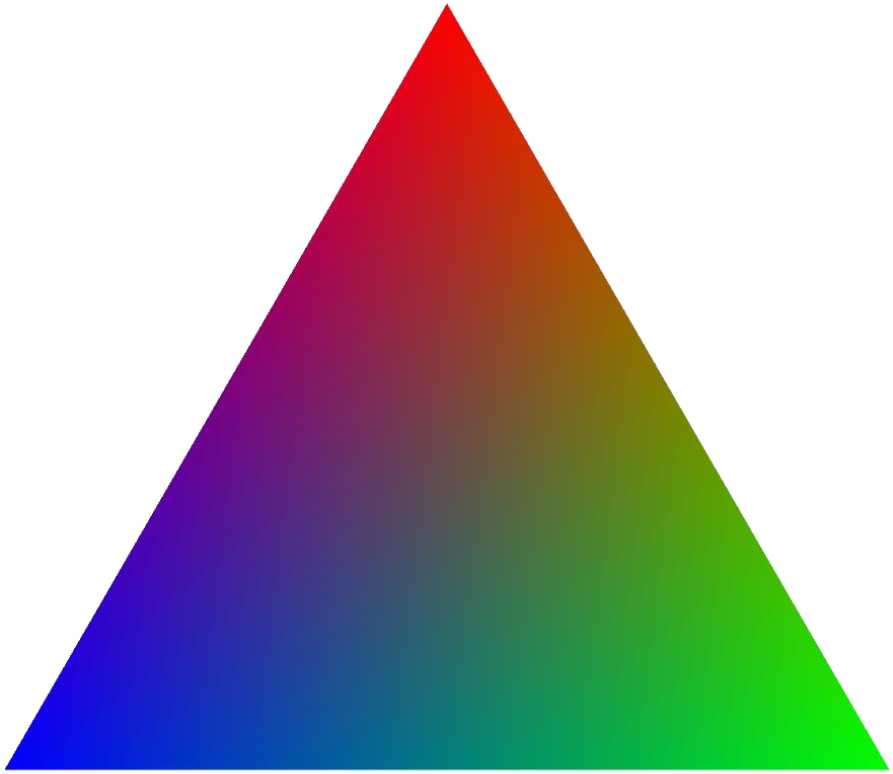
Caveat About Today's Lecture

- We want this lecture to be **API agnostic**
  - CUGL currently uses **OpenGL** for its rendering
  - **Vulkan** is very close and want to prepare you
- However, shader code causes a bit of a problem
  - CUGL **OpenGL** on Desktop uses **GLSL 4.1 core**
  - CUGL **OpenGLES** on mobile uses **GLSL 3.0 es**
  - **Vulkan** uses **GLSL 4.5 Vulkan profile**
- These are different **dialects** (Vulkan is very different)
  - We will use **GLSL 3.0 es** for our code examples
  - Will compare other dialects when it is important

Graphics Cards Draw Triangles



Triangles Can Be Colored

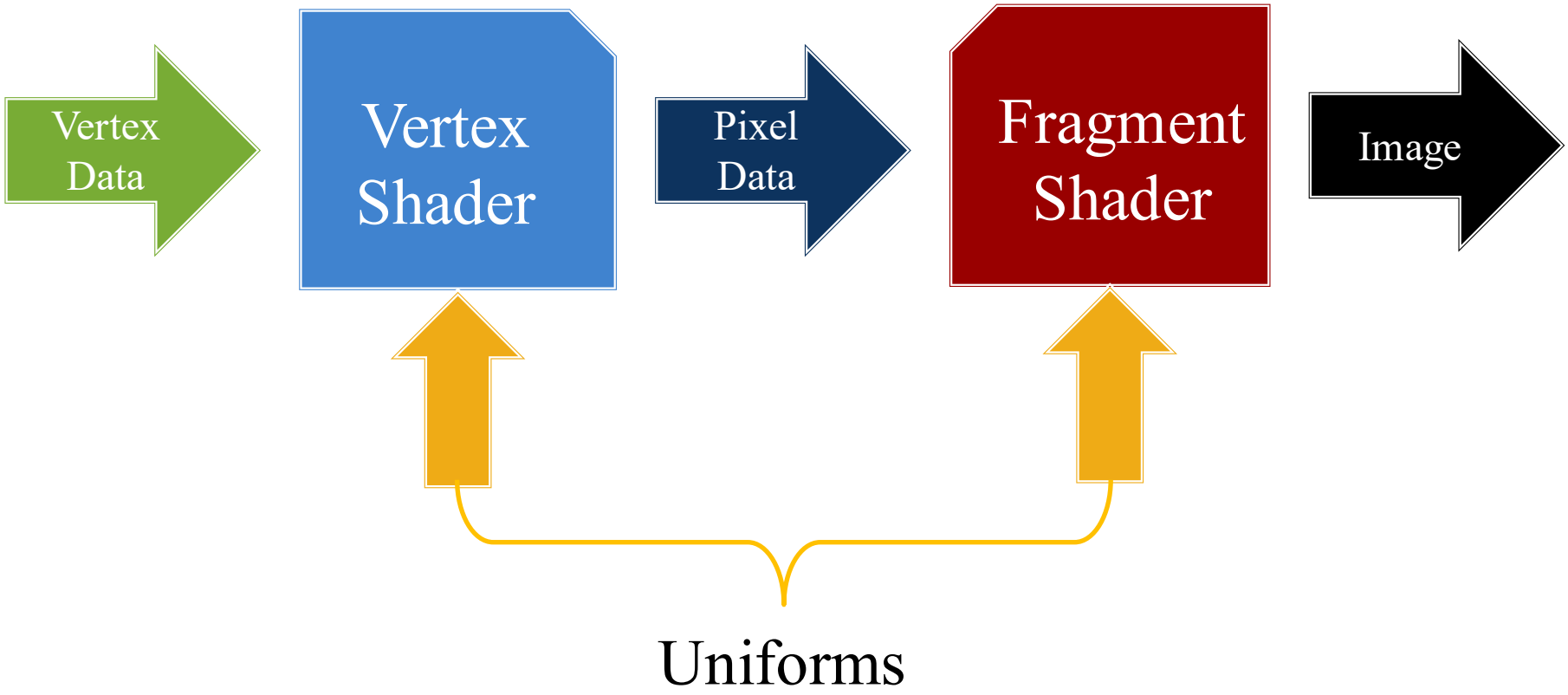


Triangles Can Be Textured

Triangles Can Be Both

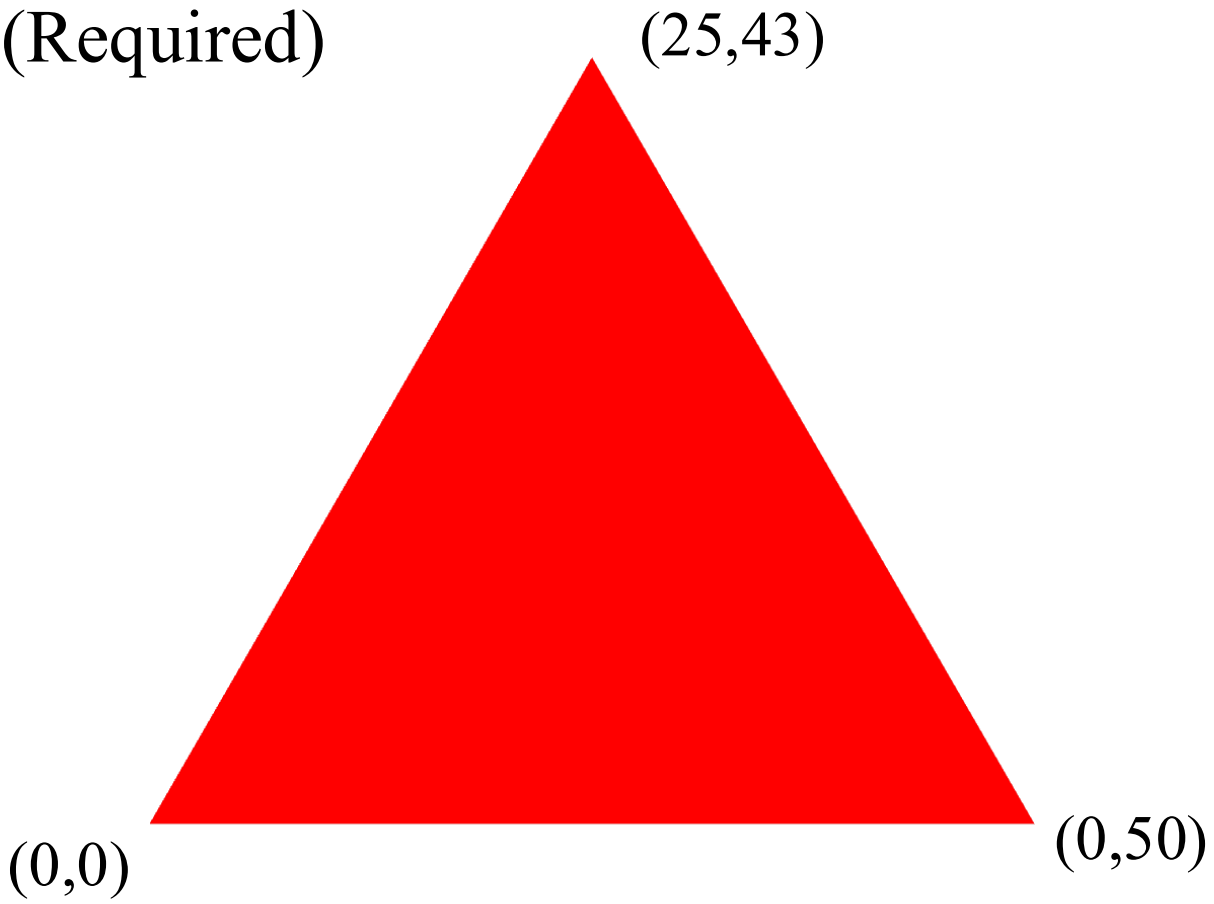
A Sprite is (Often) Two Triangles

Triangles are Drawn with Shaders



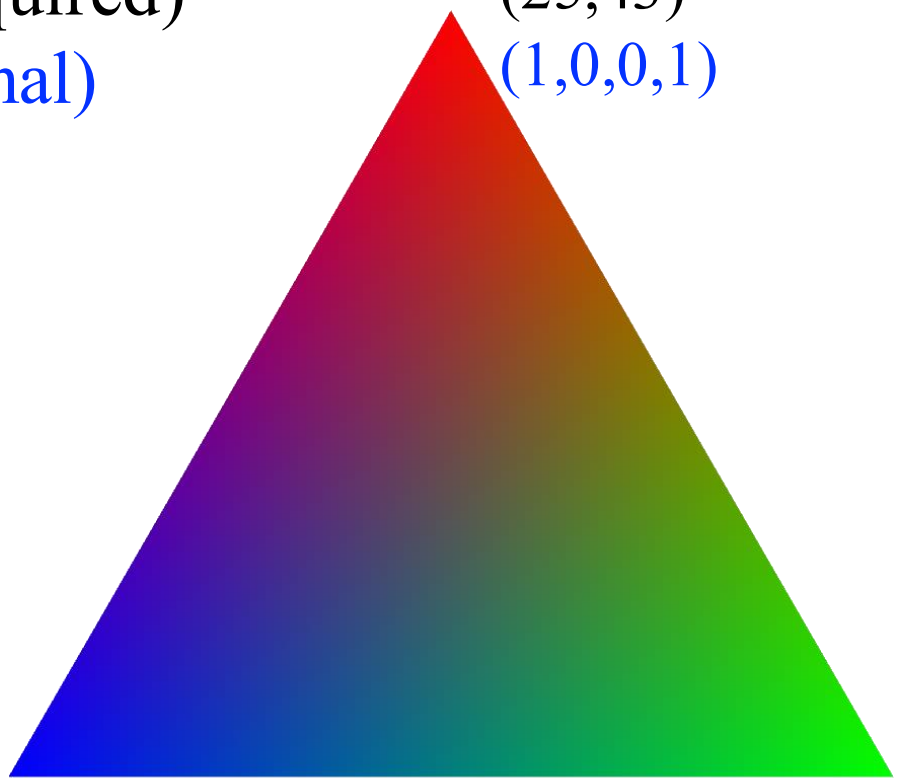
Vertex Data Defines the Triangle

Position (Required)



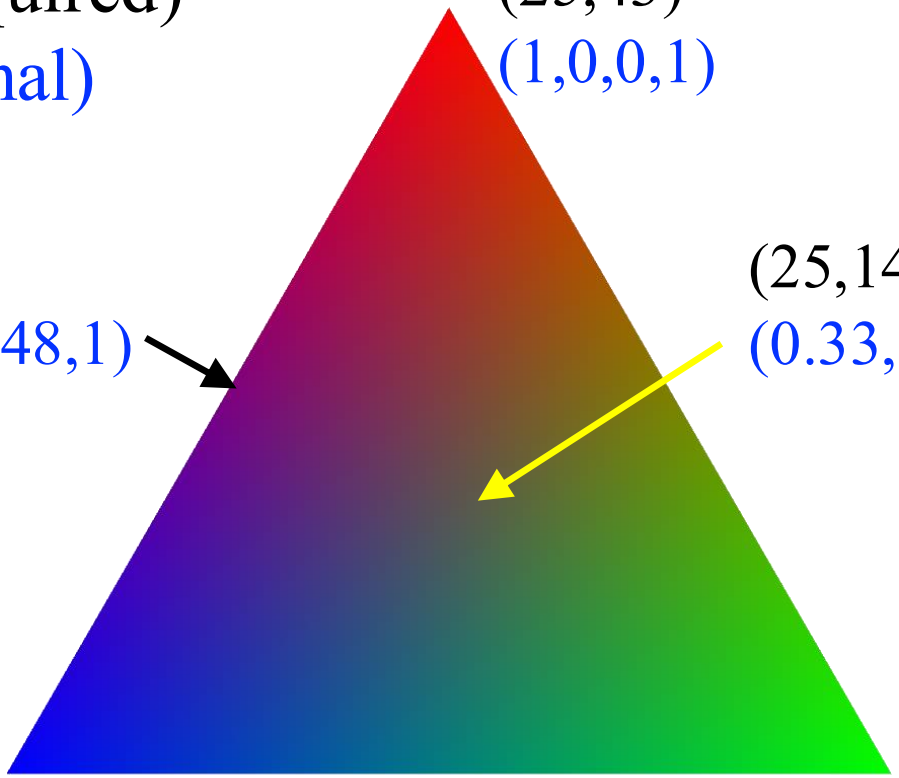
Vertex Data Defines the Triangle

Position (Required)
Color (Optional)



Vertex Shader **Interpolates** Pixels

Position (Required)
Color (Optional)



What Does The Fragment Shader Do?

- Vertex shader just produces interpolated values
  - Interpolated vector for position
  - Interpolated color for the pixel
- Fragment shader assigns the "official" color
  - May be the color interpolated by vertex shader
  - May be some variation of this color
- Often applies post-processing effects
  - **Example:** gaussian blur
  - Sometimes the more complicated of the two

A Very Simple Shader (GLSL 3.0)

Vertex Shader

```
// 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;
}
```

Input

Input

Fragment Shader

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

// Color resu
vertex shader
in vec4 outColor;

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

Input

A Very Simple Shader (GLSL 3.0)

Vertex Shader

```
// Positions
in vec4 aPosition;

// Colors
in  vec4 aColor;
out vec4 outColor;

uniform mat4 uCamera;

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

Input

Input

Output

Output

Fragment Shader

```
// The output
out vec4 frag_color;

// Color resu
vertex shader
in vec4 outColor;

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

Output

Input

A Very Simple Shader (GLSL 3.0)

Vertex Shader

```
// Positions
in vec4 aPosi
// Colors
in  vec4 aCo
out vec4 out
uniform mat4 uCamera;
// Interp      ition
and color
void main(void) {
    gl_Position =
uCamera*aPosition;
    outColor = aColor;
}
```

Input

Input

Output

Output

Fragment Shader

```
// The output
out vec4 frag_
// Color resu
vertex shader
in vec4 outColor;
// Just use color
computed
void main(v      d) {
    frag_
outcolor;
}
```

Output

Input

Pass-through

A Very Simple Shader (GLSL 3.0)

Vertex Shader

```
// 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;
}
```

Input

Input

Output

Output

Fragment Shader

```
// The output
out vec4 frag_color;

// Color resu
vertex shader
in vec4 outColor;

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

Output

Input

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 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
- Chang
  - Hav
  - Sen
  - Send m
- 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?

Aside: Vulkan and Uniforms

- Vulkan breaks uniforms into two categories
  - **Push Constants:** Small values that change often
  - **Uniforms:** Data buffers that change less often
  - (e.g. a Vulkan `Uniform` is an OpenGL `UniformBuffer`)
- CUGL adopts this convention too but…
  - We refer to **push constants** as **uniforms**
  - We refer to Vulkan **uniforms** as **resources**
  - Did this because OpenGL is still primary backend
  - May change naming when Vulkan more common

Images Have Texture Coordinates

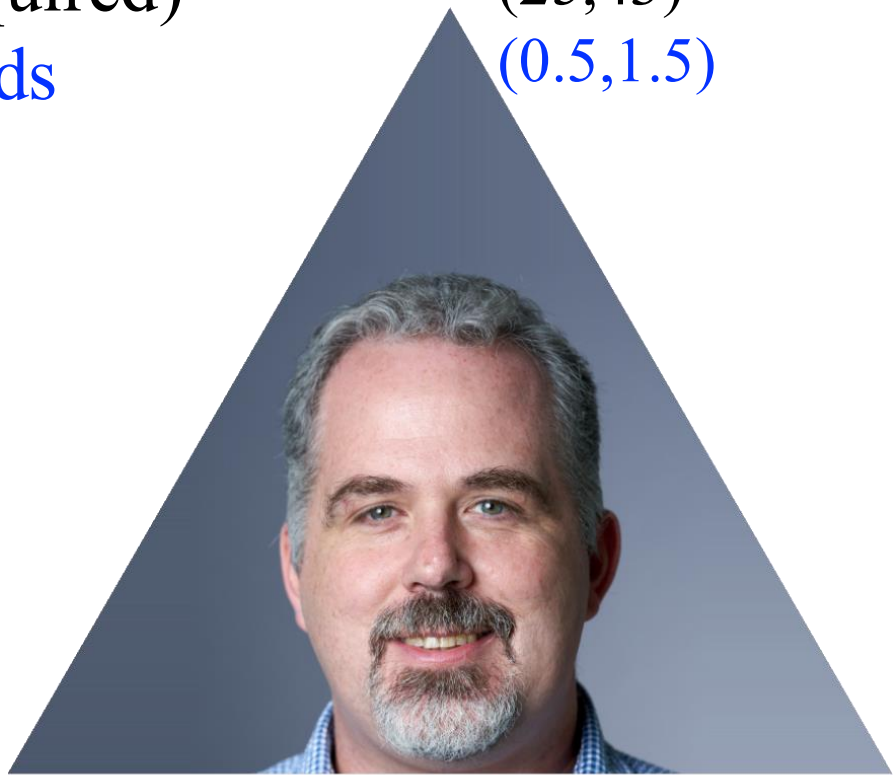
(0,1) (1,1)

(0,0) (1,0)

Vertex Data Can Include Texture Data

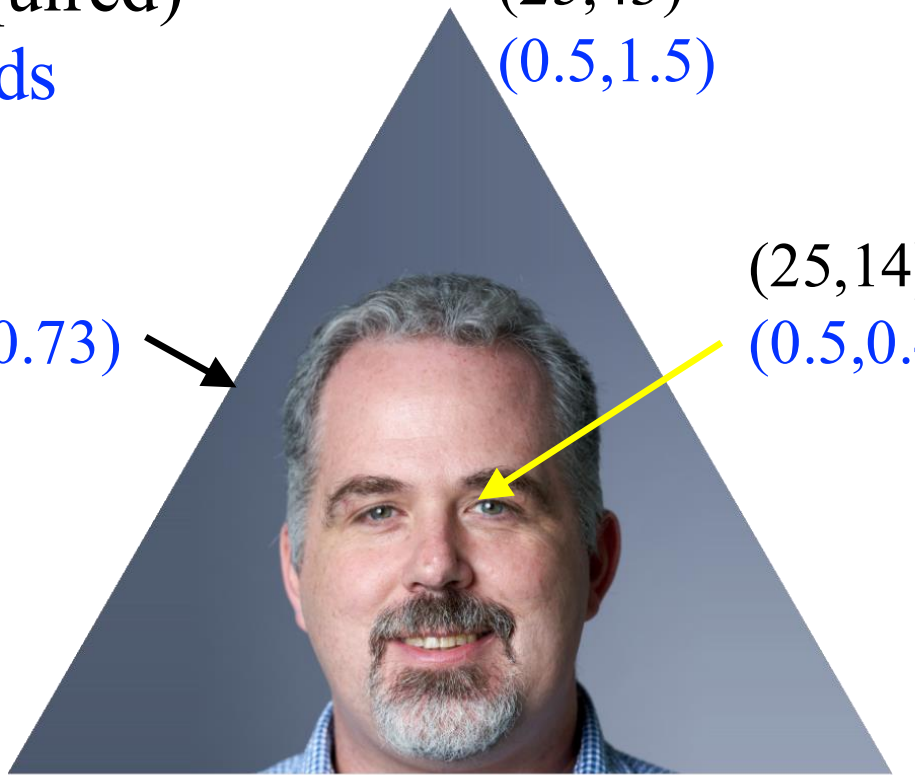
Position (Required)

Texture Coords (Optional)



Vertex Shader **Interpolates** Pixels

Position (Required)
Texture Coords (Optional)



A Texture Shader

Vertex Shader

```
// Positions
in vec4 aPosition;

// Texture Coords
in  vec2 aCoord;
out vec2 outCoord;

uniform mat4 uCamera;

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

Fragment Shader

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

// Texture coord from
vertex shader
in vec2 outCoord;

uniform sampler2D
uTexture;

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

A Texture Shader

Vertex Shader

```
// Positions
in vec4 aPosition;

// Texture Coords
in  vec2 aCoord;
out vec2 outCoord;

uniform mat4 uCamera;

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

Fragment Shader

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

// Texture coord
vertex shader
in vec2 outCoord

uniform sampler2
uTexture;

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

texture + coord = color

A Texture Shader

Vertex Shader

```
// Positions
in vec4 aPosition;

// Texture Coords
in  vec2 aCoord;
out vec2 outCoord;

uniform mat4 uCamera;

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

Fragment Shader

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

// Texture coord from
vertex shader
in vec2 outCoord;

uniform sampler2D
uTexture;

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

A Texture Shader

Vertex Shader

```
// Positions
in vec4 aPosition;

// Texture Coords
in  vec2
out vec2

uniform m

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

Fragment Shader

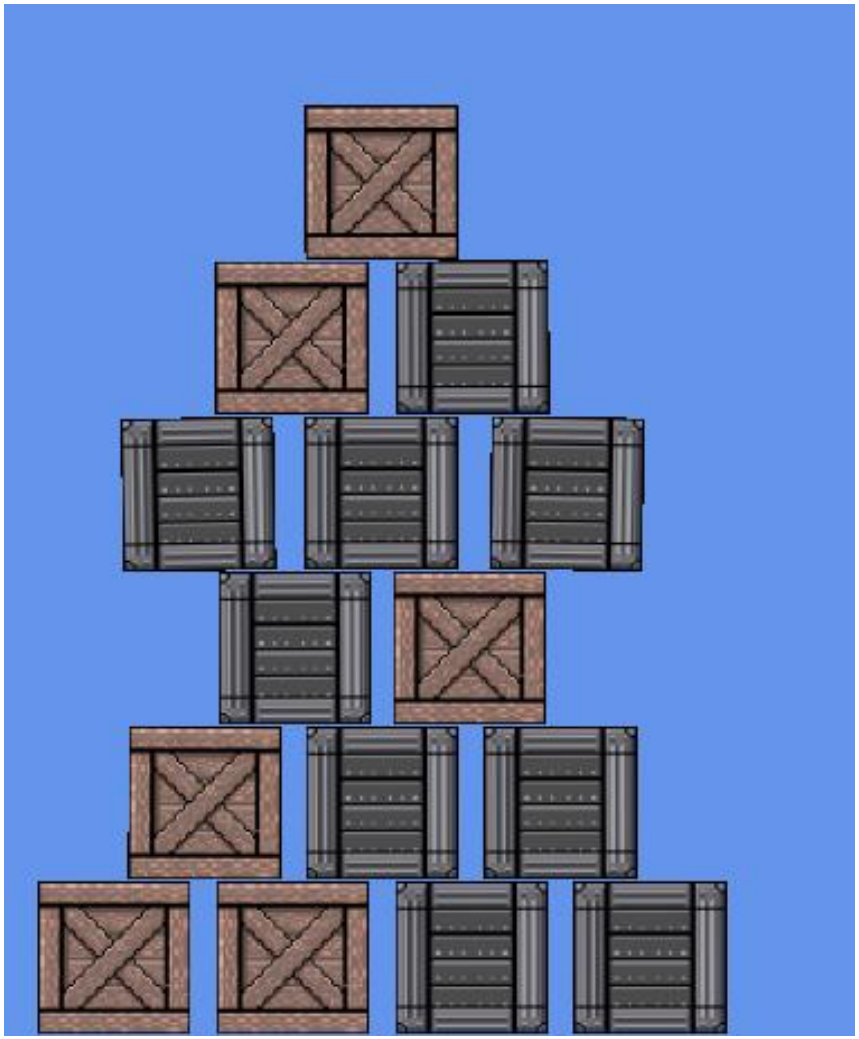
```
// The output color
out vec4 frag_color;

// Texture coord from


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

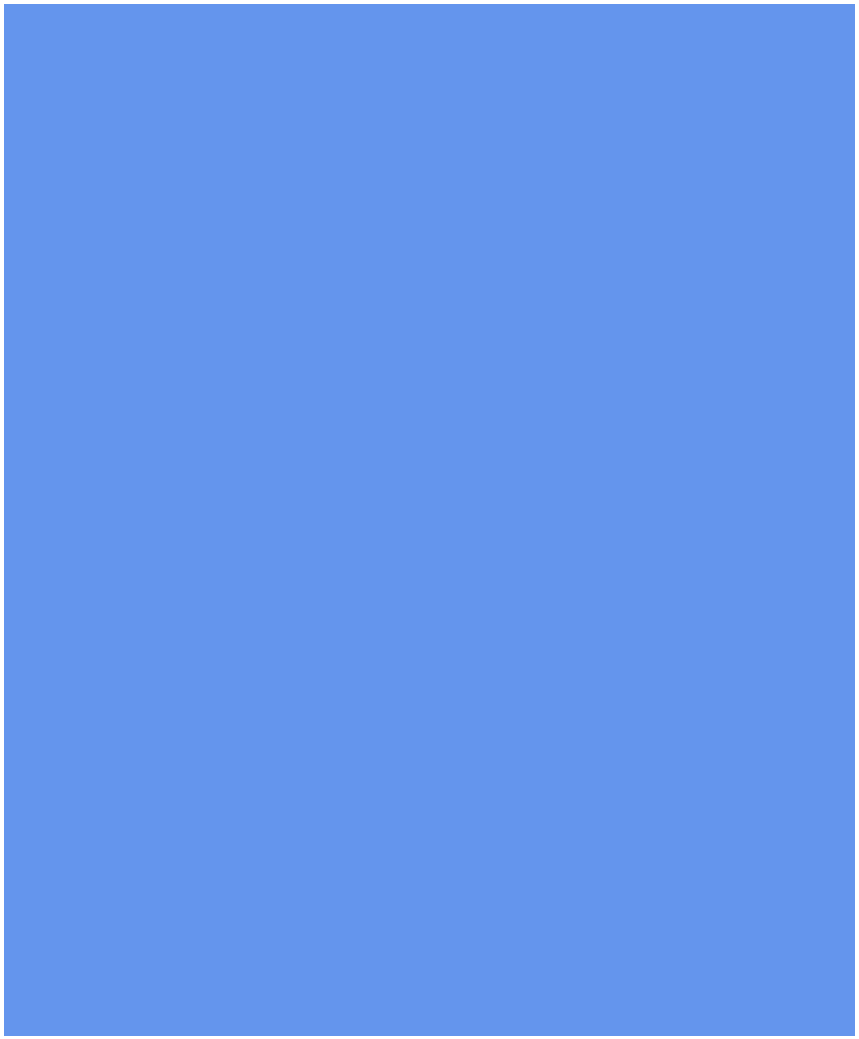
Changing the texture *stalls* the stream

How Does a `SpriteBatch` Work?



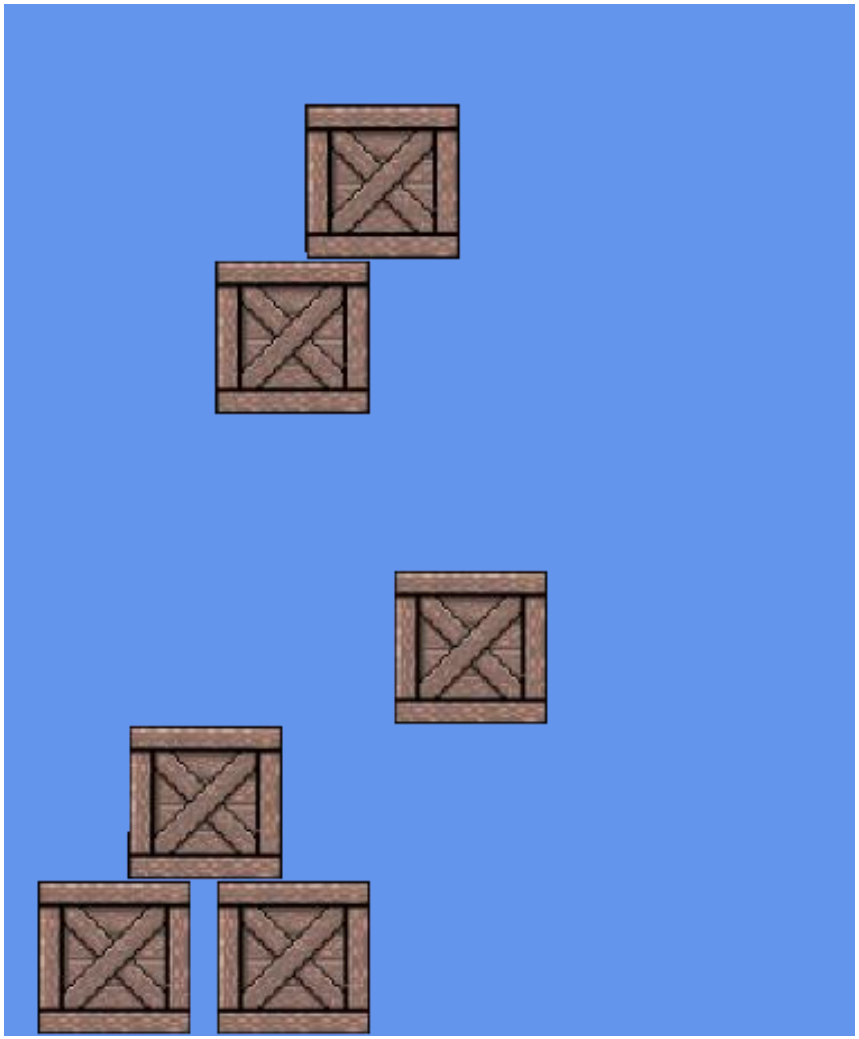
- `SpriteBatch` has a **shader**
  - Methods create vertices
  - Vertices have color, texture
  - Sends vertices to shader
- Groups data by **uniforms**
  - Adds all vertices to a set
  - Breaks set into *batches*
  - Uniforms fixed each batch
- Each texture is a **new batch**
  - How often do you switch?

How Does a `SpriteBatch` Work?



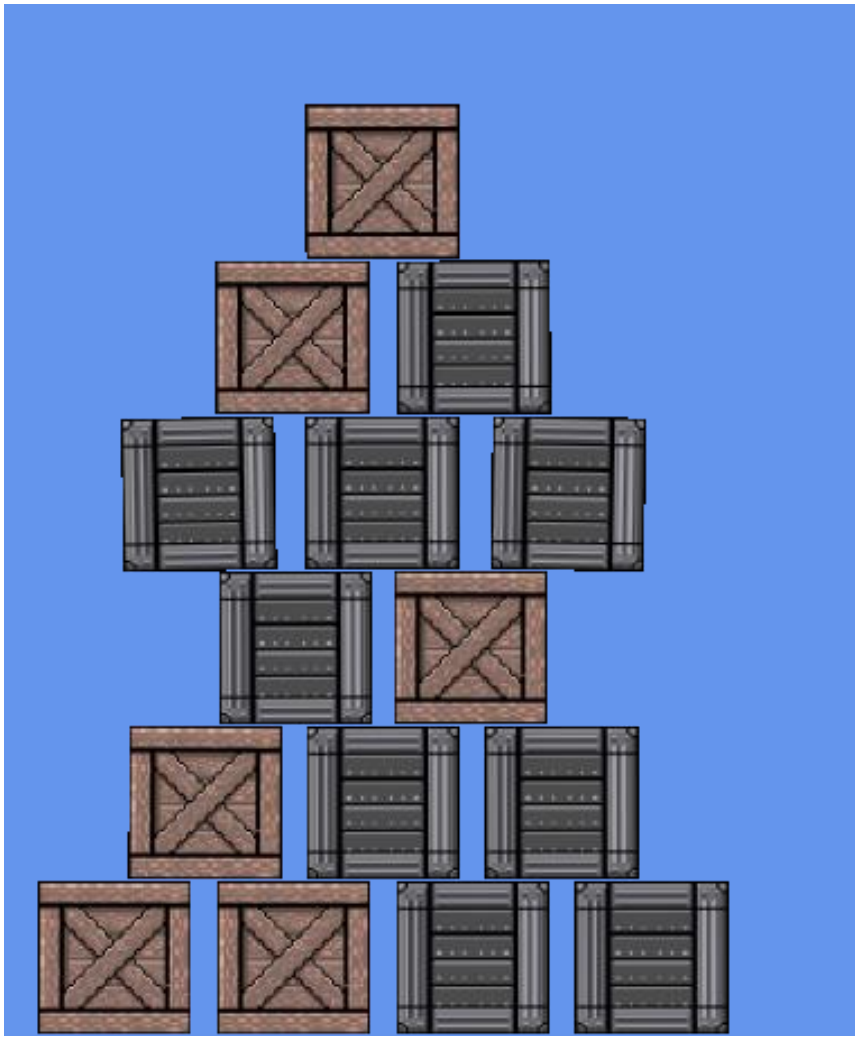
- `SpriteBatch` has a **shader**
  - Methods create vertices
  - Vertices have color, texture
  - Sends vertices to shader
- Groups data by **uniforms**
  - Adds all vertices to a set
  - Breaks set into *batches*
  - Uniforms fixed each batch
- Each texture is a **new batch**
  - How often do you switch?

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?



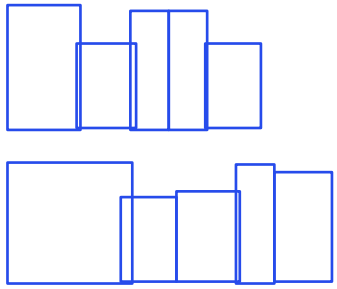
- `SpriteBatch` has a **shader**
  - Methods create vertices
  - Vertices have color, texture
  - Sends vertices to shader
- Groups data by **uniforms**
  - Adds all vertices to a set
  - Breaks set into *batches*
  - Uniforms fixed each batch
- Each texture is a **new batch**
  - How often do you switch?

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

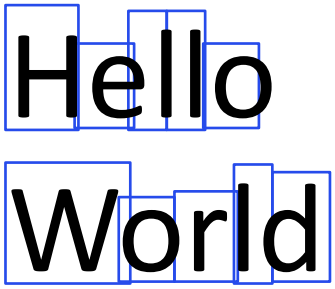
Aside: This is How Fonts Work

- 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)



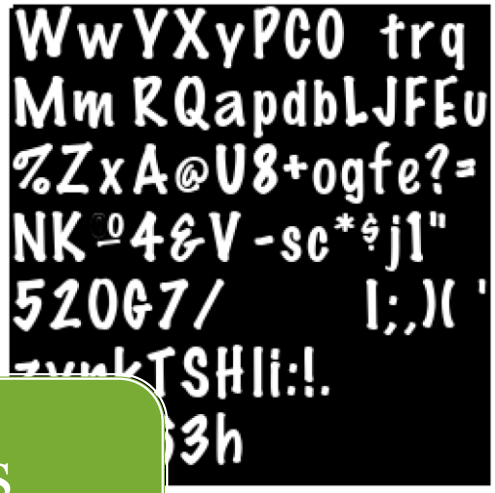
Aside: This is How Fonts Work

- 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)

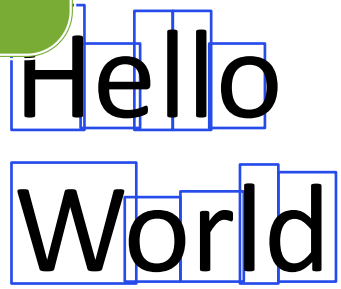


Aside: This is How Fonts Work

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



Which glyphs go in the atlas?



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

Optimizing Performance: **Atlases**

- **Idea**: Never switch textures
  - Sprite sheet is many images
  - We can draw part of texture
  - One tex[illegible]
- Called a
  - Suppo
  - See fil
  - Ideal f
- Has some **disadvantages**
  - Textures cannot repeat
  - Recall texture size limits

Better: Vulkan supports Texture **array** uniforms

A Vulkan Texture Shader

Vertex Shader

```
// Attributes
layout(location=0) in vec4
aPosition;
layout(location=1) in vec2
aCoord;
layout(location=2) in uint
aIndex;

layout(location=0) out vec2
outCoord;
layout(location=1) flat out
unit outIndex;

// Push Constant
layout(push_constant) uniform
camPC {
    mat4 uCamera;
} camera;
void main(void) {
```

Fragment Shader

```
// Attributes
layout(location=0) out vec4
frag_color;
layout(location=0) in vec2
outCoord;
layout(location=0) flat in uint
outIndex;

// Texture array
layout(set=0, binding=0)
uniform
    sampler2D uTextures[];

void main(void) {
    nonuniformEXT uint idx =
outIndex;
    frag_color =
texture(uTextures[idx],
outCoord);
```

A Vulkan Texture Shader

Vertex Shader

```
// Attributes
layout(location=0) in vec4
aPosition;
layout(location=1) in vec2
aCoord;
layout(location=2) in uint
aIndex;

layout(location=0) out vec2
outCoord;
layout(location=1) out
unit outIndex;

// Push Constant
layout(push_constant) uniform
camPC {
    mat4 uCamera;
} camera;
void main(void) {
```

Push Constant

Fragment Shader

```
// Attributes
layout(location=0) out vec4
frag_color;
layout(location=0)
outCoord;
layout(location=1)
outIndex;

// Texture array
layout(set=0, binding=0)
uniform
    sampler2D uTextures[];

void main(void) {
    nonuniformEXT uint idx =
outIndex;
    frag_color =
texture(uTextures[idx],
outCoord);
```

"True" Uniform

A Vulkan Texture Shader

Vertex Shader

```
// Attributes
layout(location=0) in v
aPosition;
layout(location=1) in vec2
aCoord;
layout(location=2) in uint
aIndex;

layout(location
outCoord;
layout(location
unit outIndex;

// Push Constant
layout(push_constant) uniform
camPC {
    mat4 uCamera;
} camera;
void main(void) {
```

Array Index

Do Not Interpolate

Fragment Shader

```
// Attributes
layout(location=0) out vec4
frag_color;
layout(location=0)
outCoord;
layout(location=0
outIndex;

// Texture array
layout(set=0, binding=0)
uniform
    sampler2D uTextures[];

void main(void) {
    nonuniformEXT uint idx =
outIndex;
    frag_color =
texture(uTextures[idx],
outCoord);
```

Array of Textures

The SpriteBatch Shader

```
out vec4 frag_color;

in vec2 outPosition;
in vec4 outColor;
in vec2 outTexCoord;
in vec2 outGradCoord;

uniform sampler2D uTexture;
uniform int  uType;
uniform vec2 uBlur;
layout (std140) uniform uContext
{
    mat3 scMatrix;      // 48
    vec2 scExtent;      //  8
    vec2 scScale;       //  8
    mat3 gdMatrix;      // 48
    vec4 gdInner;       // 16
    vec4 gdOuter;       // 16
    vec2 gdExtent;      //  8
    float gdRadius;     //  4
    float gdFeathr;     //  4
};

float boxgradient(vec2 pt, vec2 ext, float radius, float feather) {
    vec2 ext2 = ext - vec2(radius,radius);
    vec2 dst = abs(pt) - ext2;
    float m = min(max(dst.x,dst.y),0.0) + length(max(dst,0.0)) - radius;
    return clamp((m + feather*0.5) / feather, 0.0, 1.0);
}

float scissormask(vec2 pt) {
    vec2 sc = (abs((scMatrix * vec3(pt,1.0)).xy) - scExtent);
    sc = vec2(0.5,0.5) - sc * scScale;
    return clamp(sc.x,0.0,1.0) * clamp(sc.y,0.0,1.0);
}

vec4 blursample(vec2 coord) {
    float factor[5] = float[]( 1.0,  4.0, 6.0, 4.0, 1.0 );
    float steps[5]  = float[]( -1.0, -0.5, 0.0, 0.5, 1.0 );

    vec4 result = vec4(0.0);
    for(int ii = 0; ii < 5; ii++) {
        vec4 row = vec4(0.0);
        for(int jj = 0; jj < 5; jj++) {
            vec2 offs = vec2(uBlur.x*steps[ii],uBlur.y*steps[jj]);
            row += texture(uTexture, coord + offs)*factor[jj];
        }
        result += row*factor[ii];
    }
    return result/vec4(256);
}

void main(void) {
    vec4 result;
    float fType = float(uType);

    if (mod(fType, 4.0) >= 2.0) {
        // Apply a gradient color
        mat3  cmatrix = gdMatrix;
        vec2  cextent = gdExtent;
        float cfeathr = gdFeathr;
        vec2 pt = (cmatrix * vec3(outGradCoord,1.0)).xy;
        float d = boxgradient(pt,cextent,gdRadius,cfeathr);
        result = mix(gdInner,gdOuter,d)*outColor;
    } else {
        // Use a solid color
        result = outColor;
    }
    if (mod(fType, 2.0) == 1.0) {
        // Include texture (tinted by color and/or gradient)
        if (uType >= 8) {
            result *= blursample(outTexCoord);
        } else {
            result *= texture(uTexture, outTexCoord);
        }
    }
    if (mod(fType, 8.0) >= 4.0) {
        // Apply scissor mask
        result.w *= scissormask(outPosition);
    }
    frag_color = result;
}
```

- 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 SpriteBatch Shader

```
out vec4 frag_color;

in vec2 outPosition;
in vec4 outColor;
in vec2 outTexCoord;
in vec2 outGradCoord;

uniform sampler2D uTexture;
uniform int  uType;
uniform vec2 uBlur;
layout (std140) uniform uContext
{
    mat3 scMatrix;      // 48
    vec2 scExtent;      //  8
    vec2 scScale;       //  8
    mat3 gdMatrix;      // 48
    vec4 gdInner;       // 16
    vec4 gdOuter;       // 16
    vec2 gdExtent;      //  8
    float gdRadius;     //  4
    float gdFeathr;     //  4
};

float boxgradient(vec2 pt, vec2 ext, float radius, float feather) {
    vec2 ext2 = ext - vec2(radius,radius);
    vec2 dst = abs(pt) - ext2;
    float m = min(max(dst.x,dst.y),0.0) + length(max(dst,0.0)) - radius;
    return clamp((m + feather*0.5) / feather, 0.0, 1.0);
}

float scissormask(vec2 pt) {
    vec2 sc = (abs((scMatrix * vec3(pt,1.0)).xy) - scExtent);
    sc = vec2(0.5,0.5) - sc * scScale;
    return clamp(sc.x,0.0,1.0) * clamp(sc.y,0.0,1.0);
}

vec4 blursample(vec2 coord) {
    float factor[5] = float[]( 1.0,  4.0, 6.0, 4.0, 1.0 );
    float steps[5]  = float[]( -1.0, -0.5, 0.0, 0.5, 1.0 );

    vec4 result = vec4(0.0);
    for(int ii = 0; ii < 5; ii++) {
        vec4 row = vec4(0.0);
        for(int jj = 0; jj < 5; jj++) {
            vec2 offs = vec2(uBlur.x*steps[ii],uBlur.y*steps[jj]);
            row += texture(uTexture, coord + offs)*factor[jj];
        }
        result += row*factor[ii];
    }
    return result/vec4(256);
}

void main(void) {
    vec4 result;
    float fType = float(uType);

    if (mod(fType, 4.0) >= 2.0) {
        // Apply a gradient color
        mat3  cmatrix = gdMatrix;
        vec2  cextent = gdExtent;
        float cfeathr = gdFeathr;
        vec2 pt = (cmatrix * vec3(outGradCoord,1.0)).xy;
        float d = boxgradient(pt,cextent,gdRadius,cfeathr);
        result = mix(gdInner,gdOuter,d)*outColor;
    } else {
        // Use a solid color
        result = outColor;
    }
    if (mod(fType, 2.0) == 1.0) {
        // Include texture (tinted by color and/or gradient)
        if (uType >= 8) {
            result *= blursample(outTexCoord);
        } else {
            result *= texture(uTexture, outTexCoord);
        }
    }
    if (mod(fType, 8.0) >= 4.0) {
        // Apply scissor mask
        result.w *= scissormask(outPosition);
    }
    frag_color = result;
}
```

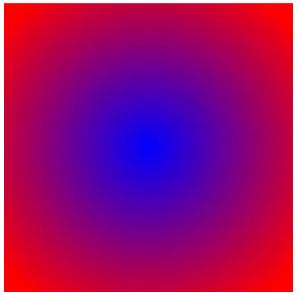
- Provides support for
  - Solid/vertex colors
  - Color gradients (linear, radial)
  - Textures/texture coords
  - Gaussian blur
  - Scissoring/masking
- Not "**user-serviceable**"
  - Do not try to replace this
  - Will break all the UI code
- More on that next time

Gradients

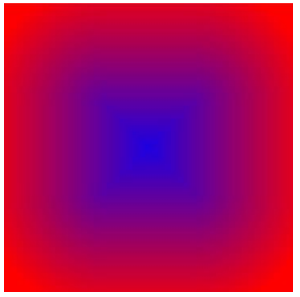
- `Gradient` in `cugl::graphics`
  - Only supports two *stops*
  - More colors = more shapes
- Has its own coordinates
  - Defined on unique square
  - Coords define the "stretch"
  - Often same as texture cords
- Primarily nice in UI effects
  - Can be defined in JSON
  - But no Figma support

Linear

Radial

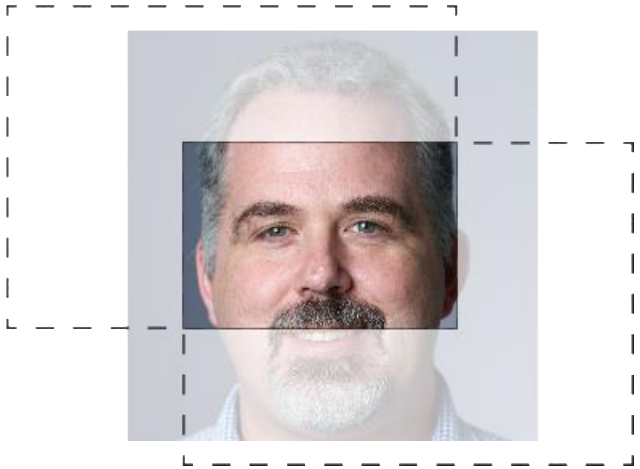


Box



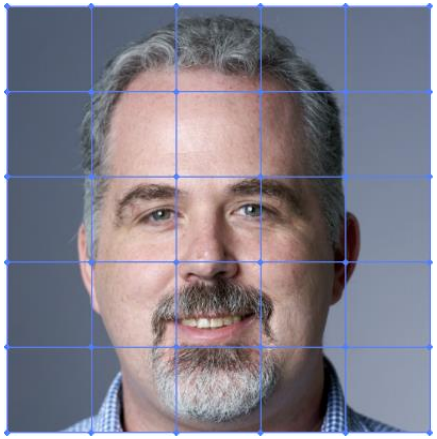
Scissors

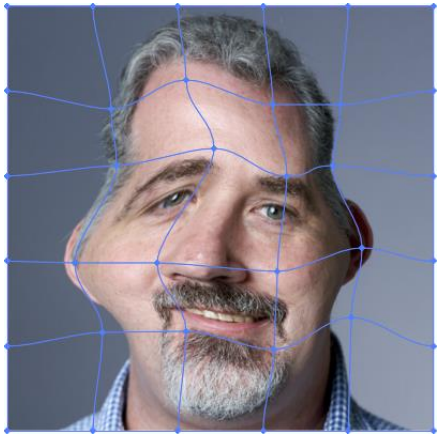
- Mask part of the screen
  - Defined as a rectangle
  - Drops pixels outside rect
- Scissors can be…
  - Rotated, Transformed
  - Intersected
  - But not really *both*
- Used by `ScrollPane`
  - Makes internal "window"
  - Can scroll the contents



What Goes to The Shader?

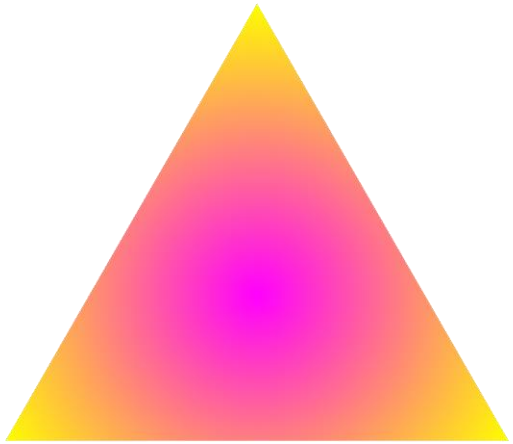
- Templated class `Mesh<T>`
  - Type is a **vertex** class
  - Mesh adds **geometry** info
- CUGL meshes are special
  - Meshes are **CPU side**
  - They API independent
  - Backend syncs with API
- Vertex must match shader
  - Check each vertex shader `in`
  - Must have attribute for it





The Vertex Class

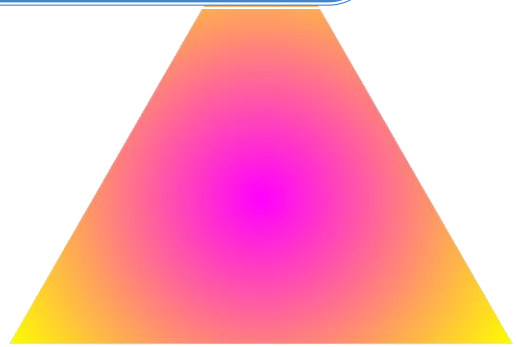
- Can be **any class** of your making
  - Should have **position** (`Vec2`, `Vec3`, or `Vec4`)
  - Can have anything else that you want
  - There are (almost) no restrictions
- **Example**: `SpriteVertex`
  - Position (`Vec2`)
  - Color (`Color4`)
  - Texture coords (`Vec2`)
  - Gradient coords (`Vec2`)



The Vertex Class

- Can be **any class** of your making
  - Should have **position** (`Vec2`, `Vec3`, or `Vec4`)
  - Can have anything else that you want
  - The
- **Exan**
  - Position (`Vec2`)
  - Color (`Color4`)
  - Texture coords (`Vec2`)
  - Gradient coords (`Vec2`)

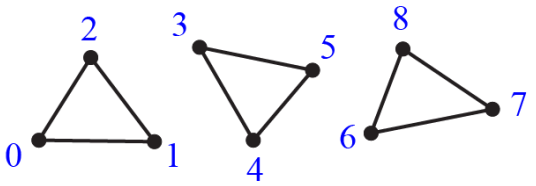
See code demos for other examples



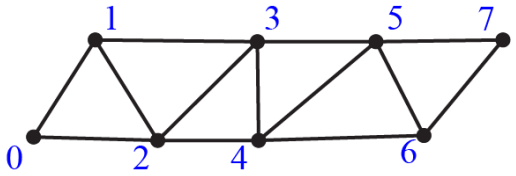
The Mesh Geometry

- 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 most of this for you!
  - Only supports triangle **lists**
  - Also only has positional data
  - But can *initialize* a Mesh

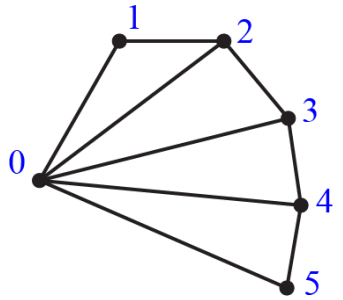
Triangle List



Triangle Strip

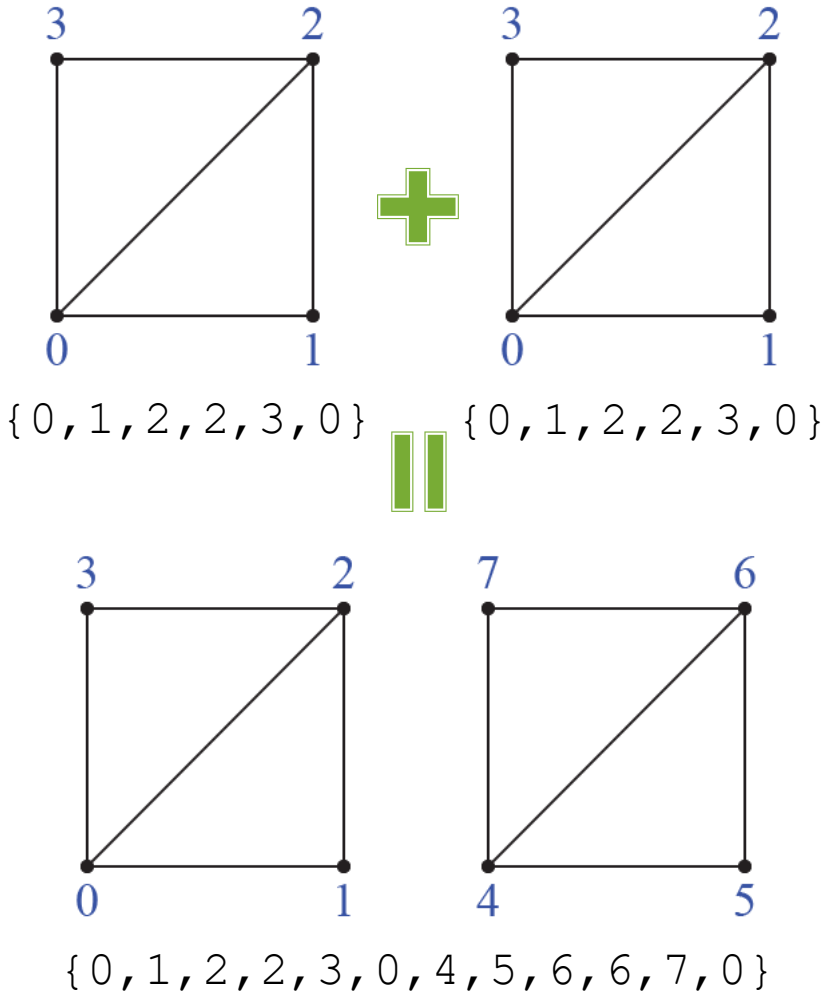


Triangle Fan



Why Triangle Lists?

- Lists are the least compact
  - Lists need 3n indices
  - Strip uses n+2 indices
  - Fan also uses n+2 indices
- But lists are **compositional**
  - Lists can be concatenated
  - Not true for fan/strips
- Needs fewer commands
  - How sprite batch works
  - Just one `Mesh<SpriteVertex>`



Standard Mesh Creation

- Use CUGL tools to **create a geometry**
  - Geometry defines position and triangles
  - End result is (typically) a Poly2 object
  - Just like the geometry lab
- Pass `Poly2` to the **`Mesh<T>` constructor**
  - Your vertex must have a position attribute
  - All other values are set to the default
- Manually **adjust other attributes**
  - Usually just texture and/or color
  - Choices depend on your shader

How Do We Talk to The Shader?

Next Time!

Summary

- CUGL has different rendering **backends**
  - `SpriteBatch` (usually) makes all of this easy
  - But it is possible to create your own pipelines
- All data sent to graphics card is a **mesh**
  - An array of vertices
  - A geometry on those vertices
  - Like `Poly2` but with more attributes
- **Shaders** render a mesh to the screen
  - Specify data at each vertex
  - Intermediate pixels are interpolated