

Advances in OpenGL ES

Session 505

Dan Omachi

GPU Software

These are confidential sessions—please refrain from streaming, blogging, or taking pictures

Introduction

- OpenGL ES offers the most direct access to graphics hardware on iOS
 - Allows lots of flexibility and power
 - Flexibility can be challenging to master
- Utilizing the API to its fullest can set your app apart
 - Make the difference between shipping something good or great

Agenda

What you will learn

- New in iOS 7
 - Instancing
 - Vertex texture sampling
 - sRGB texture formats
- Tuning and performance
 - Understand the GPU pipeline
 - Insight into feedback from GPU tools

A Note About Power Efficiency

Rendering, but not...

Rendering Requires Power

- All GPUs used by iOS are power efficient
 - Still require considerable power to render

Rendering Requires Power

- All GPUs used by iOS are power efficient
 - Still require considerable power to render
- Manage frame rate
 - Use `CADisplayLink` to sync to display
 - Target 30 fps

Rendering Requires Power

- All GPUs used by iOS are power efficient
 - Still require considerable power to render
- Manage frame rate
 - Use `CADisplayLink` to sync to display
 - Target 30 fps
- Often unnecessary to render at all
 - Do not re-render a frame if it looks the same as the previous frame
 - No animation or movement

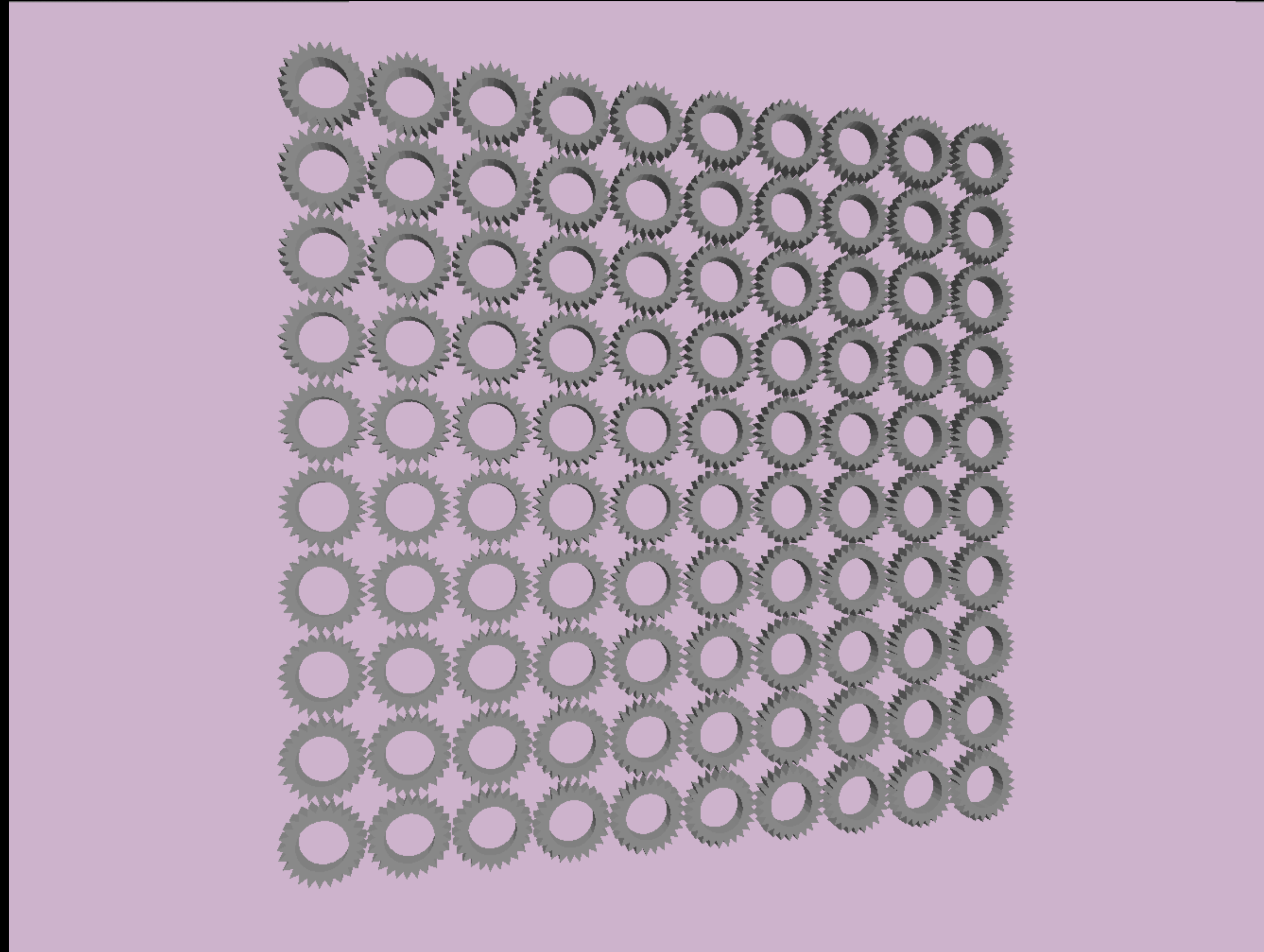
Rendering Requires Power

- All GPUs used by iOS are power efficient
 - Still require considerable power to render
- Manage frame rate
 - Use `CADisplayLink` to sync to display
 - Target 30 fps
- Often unnecessary to render at all
 - Do not re-render a frame if it looks the same as the previous frame
 - No animation or movement
- Particularly important with the multilayered iOS 7 UI
 - UI can skip compositing if nothing has changed in a layer

Instantiating

From one, come many...

Drawing Many Models



Drawing Many Models

Without instancing

```
for(x = 0; x < 10; x++)
{
    for(y = 0; y < 10; y++)
    {
        // Set uniform to position the gear
        glUniform4fv(mygearPosition[x][y]);

        // Draw the gear
        glDrawArrays(GL_TRIANGLES, 0, numGearVertices);
    }
}
```

Drawing Many Models

Without instancing

```
for(x = 0; x < 10; x++)
```

```
{
```

```
    for(y = 0; y < 10; y++)
```

```
    {
```

```
        // Set uniform to position the gear
```

```
        glUniform4fv(mygearPosition[x][y]);
```

```
        // Draw the gear
```

```
        glDrawArrays(GL_TRIANGLES, 0, numGearVertices);
```

```
    }
```

```
}
```

Drawing Many Models

Without instancing

```
for(x = 0; x < 10; x++)  
{  
    for(y = 0; y < 10; y++)  
    {  
        // Set uniform to position the gear  
        glUniform4fv(mygearPosition[x][y]);  
  
        // Draw the gear  
        glDrawArrays(GL_TRIANGLES, 0, numGearVertices);  
    }  
}
```

Drawing Many Models

Without instancing

```
for(x = 0; x < 10; x++)  
{  
    for(y = 0; y < 10; y++)  
    {  
        // Set uniform to position the gear  
        glUniform4fv(mygearPosition[x][y]);  
  
        // Draw the gear  
        glDrawArrays(GL_TRIANGLES, 0, numGearVertices);  
    }  
}
```

Drawing Many Models

Without instancing

```
for(x = 0; x < 10; x++)
{
    for(y = 0; y < 10; y++)
    {
        // Set uniform to position the gear
        glUniform4fv(mygearPosition[x][y]);

        // Draw the gear
        glDrawArrays(GL_TRIANGLES, 0, numGearVertices);
    }
}
```

What Instancing Does



- Draw the same model many times
 - Different parameter for each instance
 - Single draw call
- Each instance can have different:
 - Position
 - Matrices
 - Texture coordinates

Instancing

Two forms

- Instanced arrays
 - `GL_APPLE_instanced_arrays`
 - Instance parameters in a vertex array
- Shader instance ID
 - `GL_APPLE_draw_instanced`
 - ID variable for instance drawn in vertex shader

Instancing

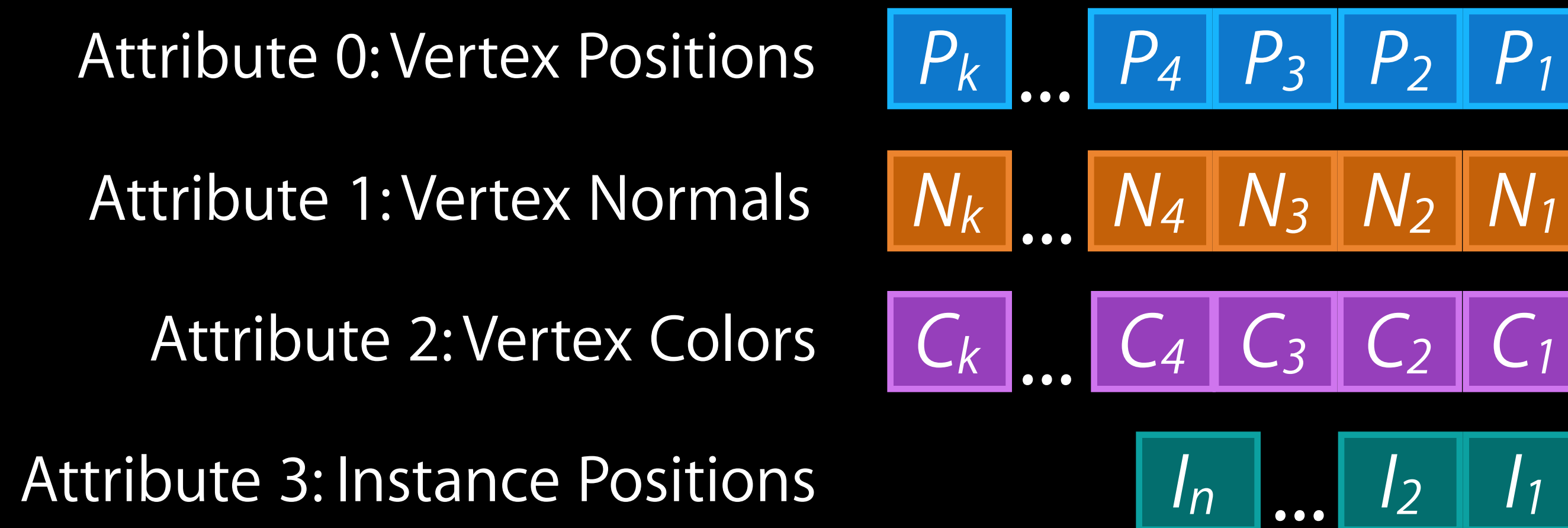
Method 1: Instanced arrays

- `glVertexAttribDivisorAPPLE` indicates
 - Attribute array supplying instance data
 - Number of instances to draw before advancing to the next element in this array
- Draw with `glDrawArraysInstancedAPPLE` or `glDrawElementsInstancedAPPLE`
 - Same as `glDrawArrays` and `glDrawElements`, but extra parameter indicates number of instances to draw

Instancing

Method 1: Instanced arrays

Vertex Arrays

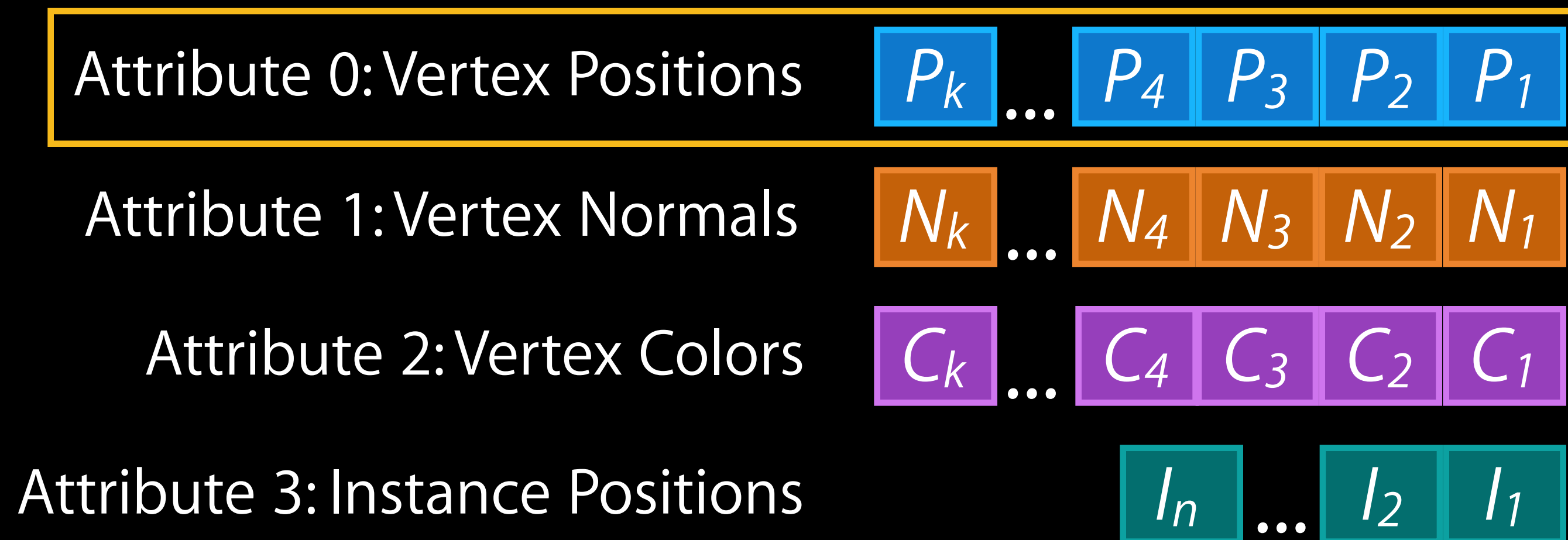


Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(0, ... , positionOffset);
```

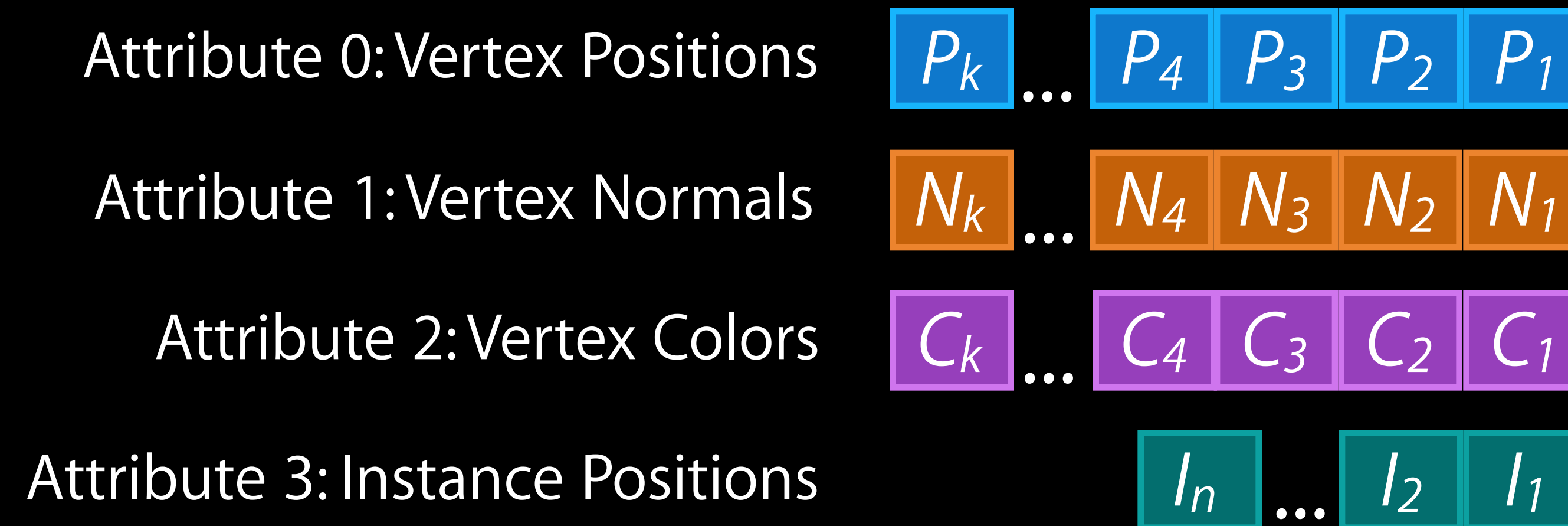
Vertex Arrays



Instancing

Method 1: Instanced arrays

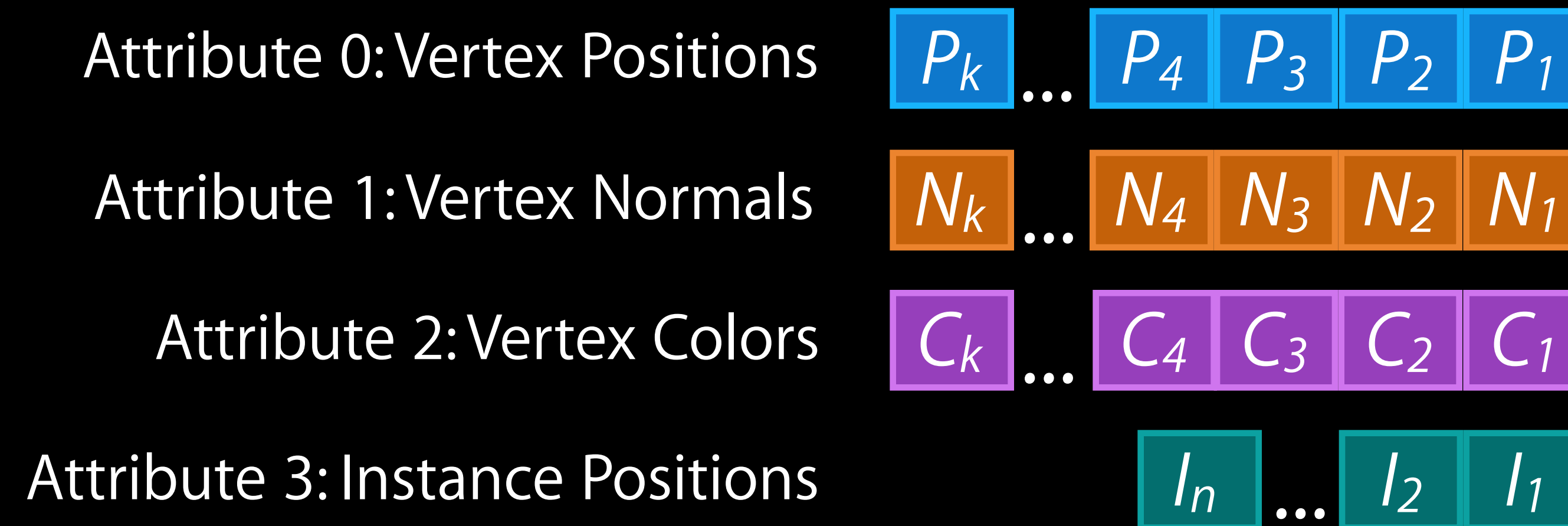
Vertex Arrays



Instancing

Method 1: Instanced arrays

Vertex Arrays

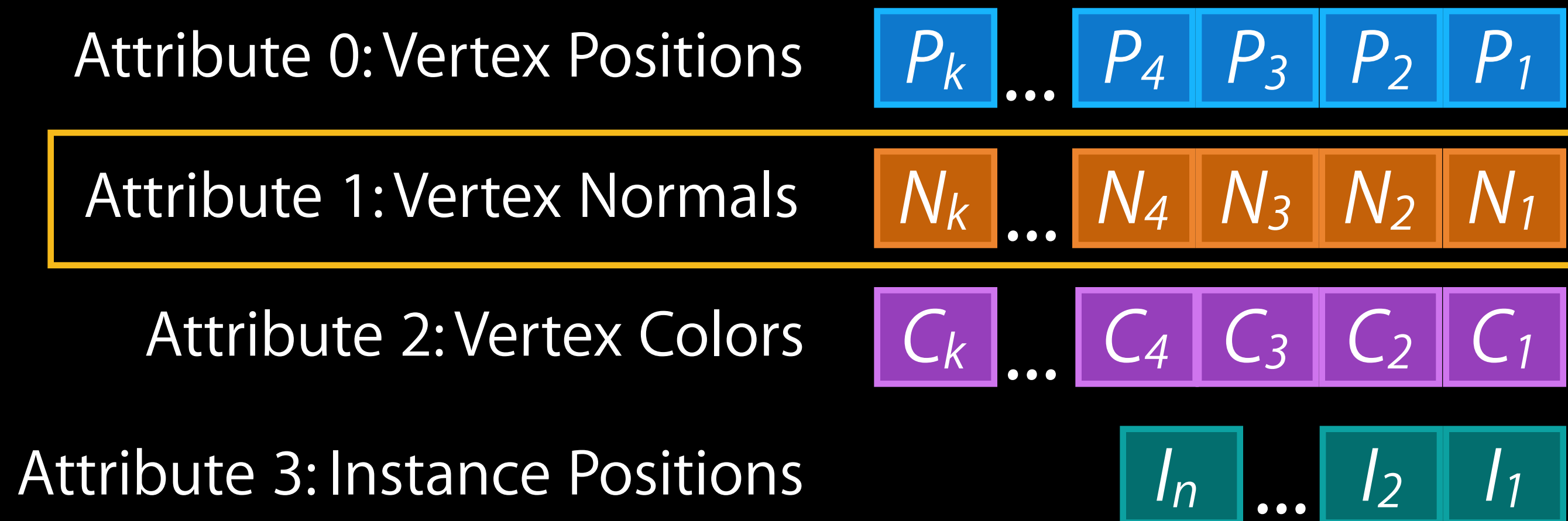


Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(1, ... , normalOffset);
```

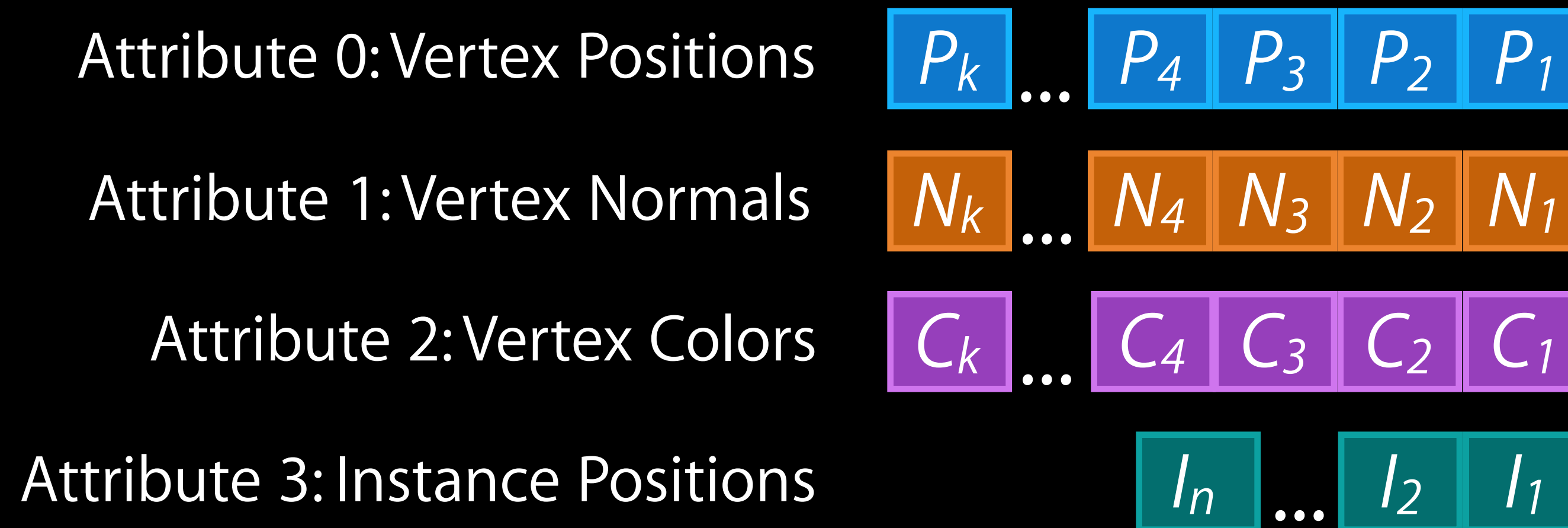
Vertex Arrays



Instancing

Method 1: Instanced arrays

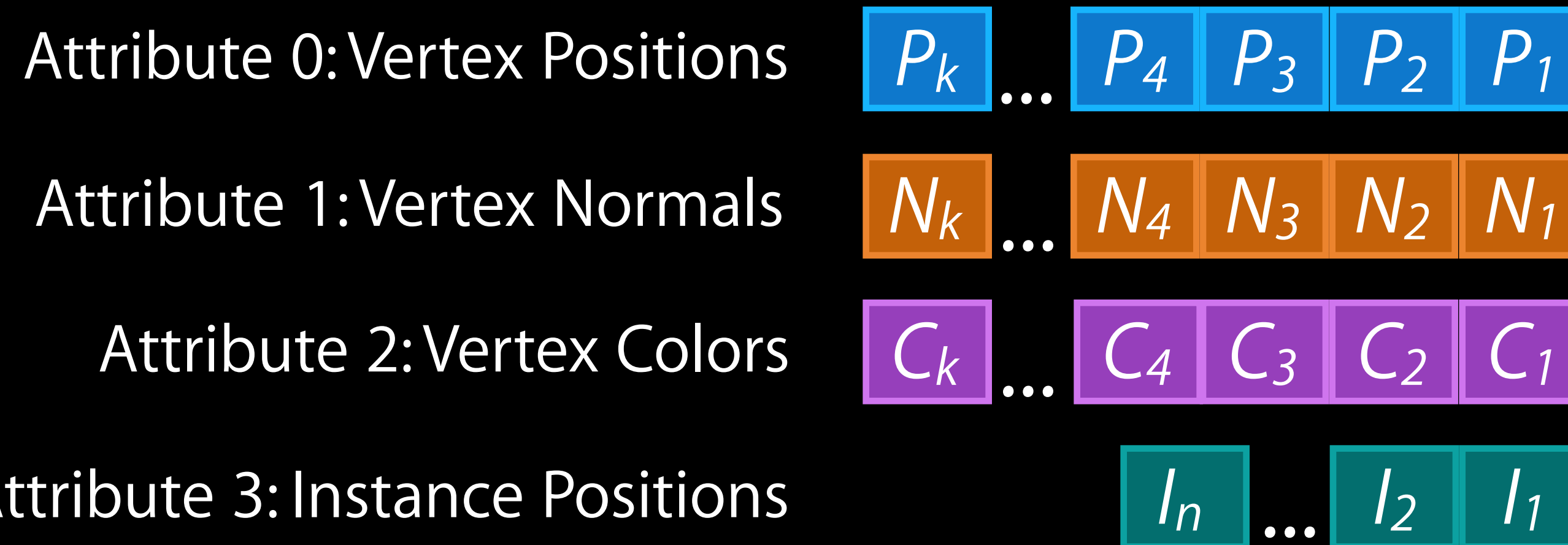
Vertex Arrays



Instancing

Method 1: Instanced arrays

Vertex Arrays



Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(2, ... , colorOffset);
```

Vertex Arrays

Attribute 0: Vertex Positions P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals N_k ... N_4 N_3 N_2 N_1

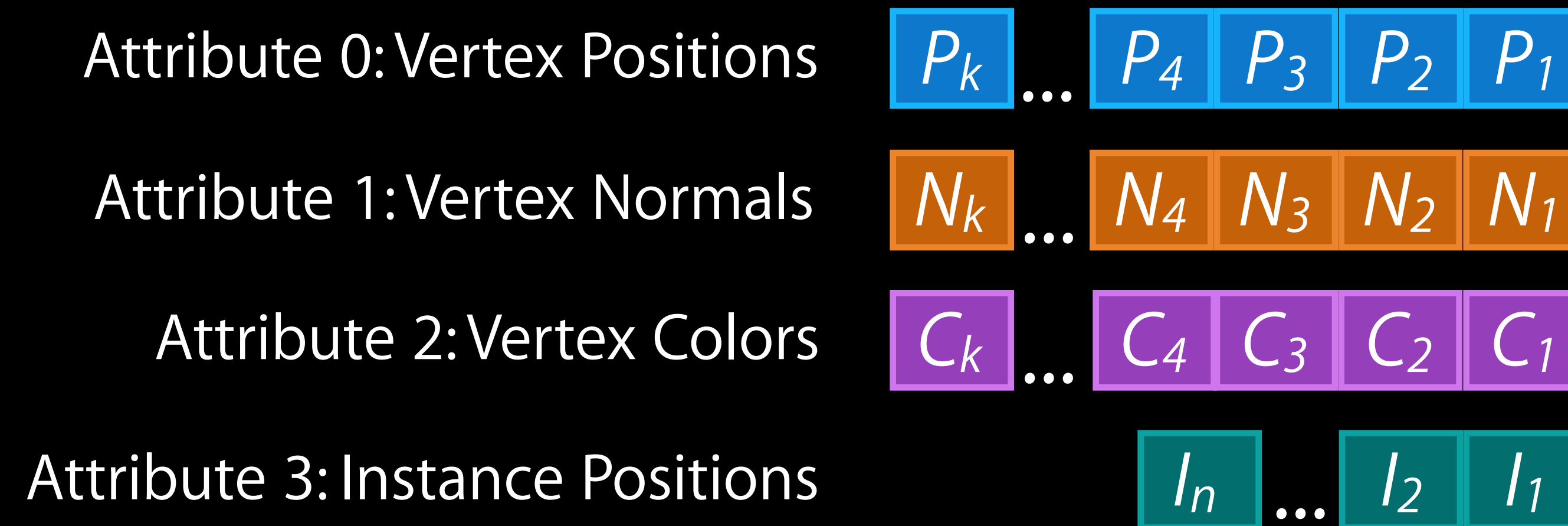
Attribute 2: Vertex Colors C_k ... C_4 C_3 C_2 C_1

Attribute 3: Instance Positions I_n ... I_2 I_1

Instancing

Method 1: Instanced arrays

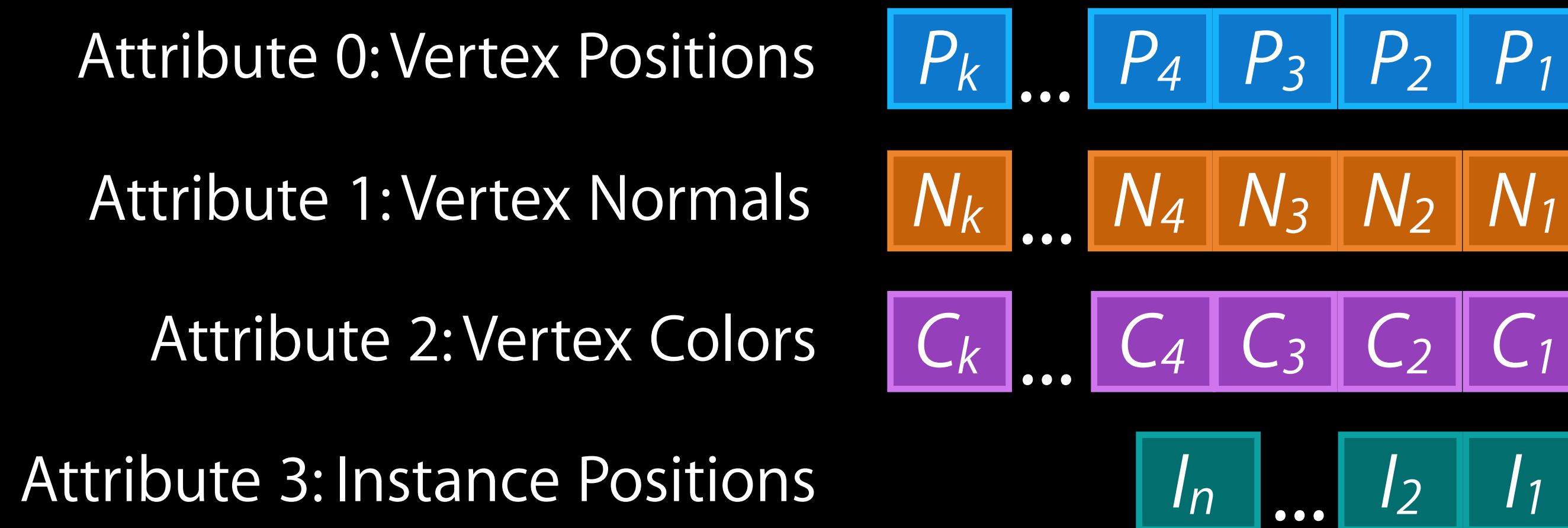
Vertex Arrays



Instancing

Method 1: Instanced arrays

Vertex Arrays



Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

Vertex Arrays

Attribute 0: Vertex Positions P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors C_k ... C_4 C_3 C_2 C_1

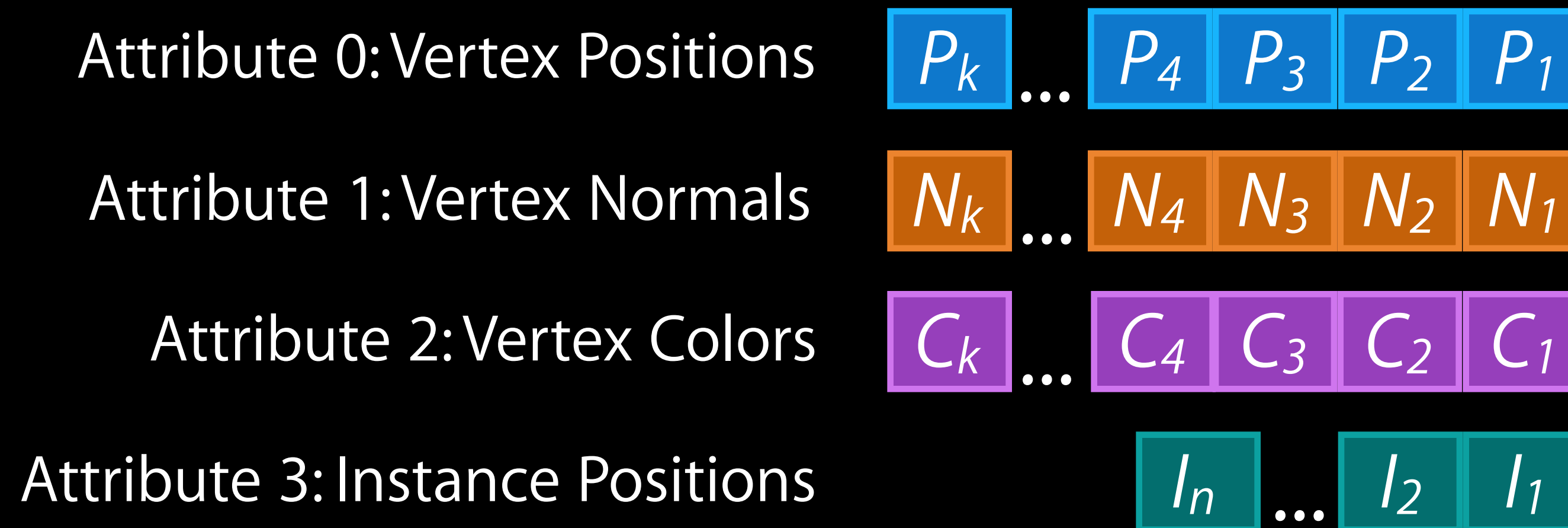
Attribute 3: Instance Positions I_n ... I_2 I_1

Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

Vertex Arrays

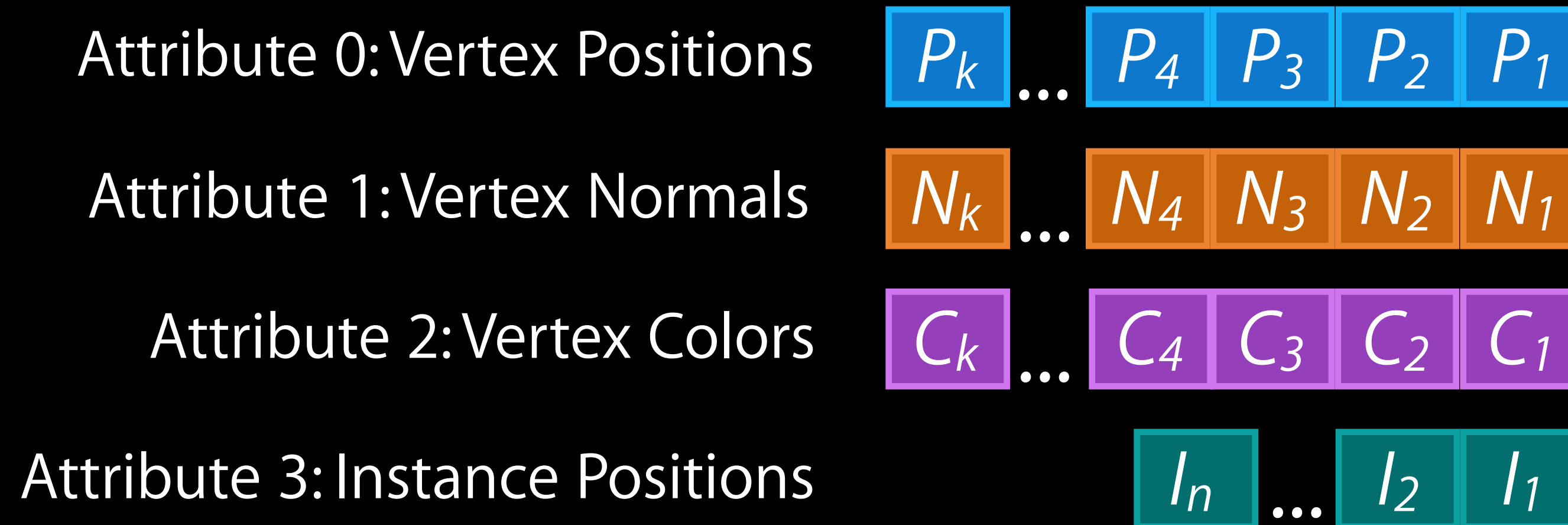


Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

Vertex Arrays



Instancing

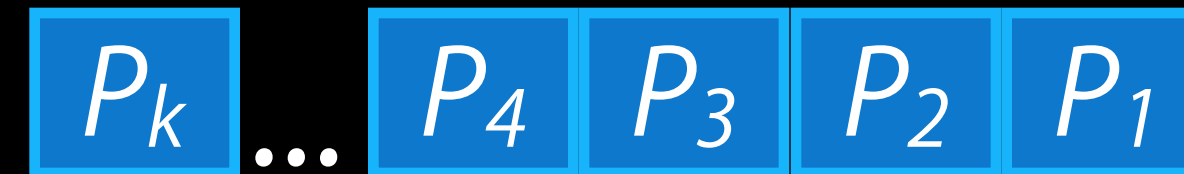
Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
glVertexAttribDivisorAPPLE(3, 1);
```

Vertex Arrays

Attribute 0: Vertex Positions



Attribute 1: Vertex Normals



Attribute 2: Vertex Colors



Attribute 3: Instance Positions



Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
glVertexAttribDivisorAPPLE(3, 1);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Attribute 3: Instance Positions

I_n ... I_2 I_1

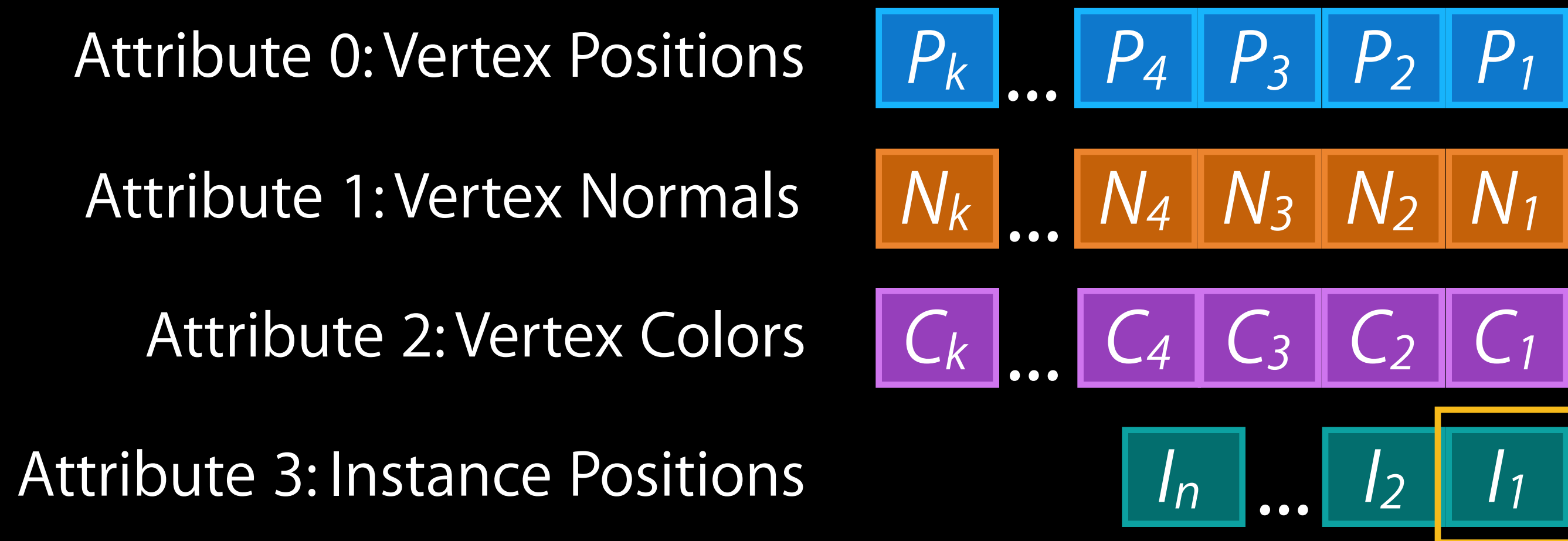
Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
glVertexAttribDivisorAPPLE(3, 1);
```

Vertex Arrays



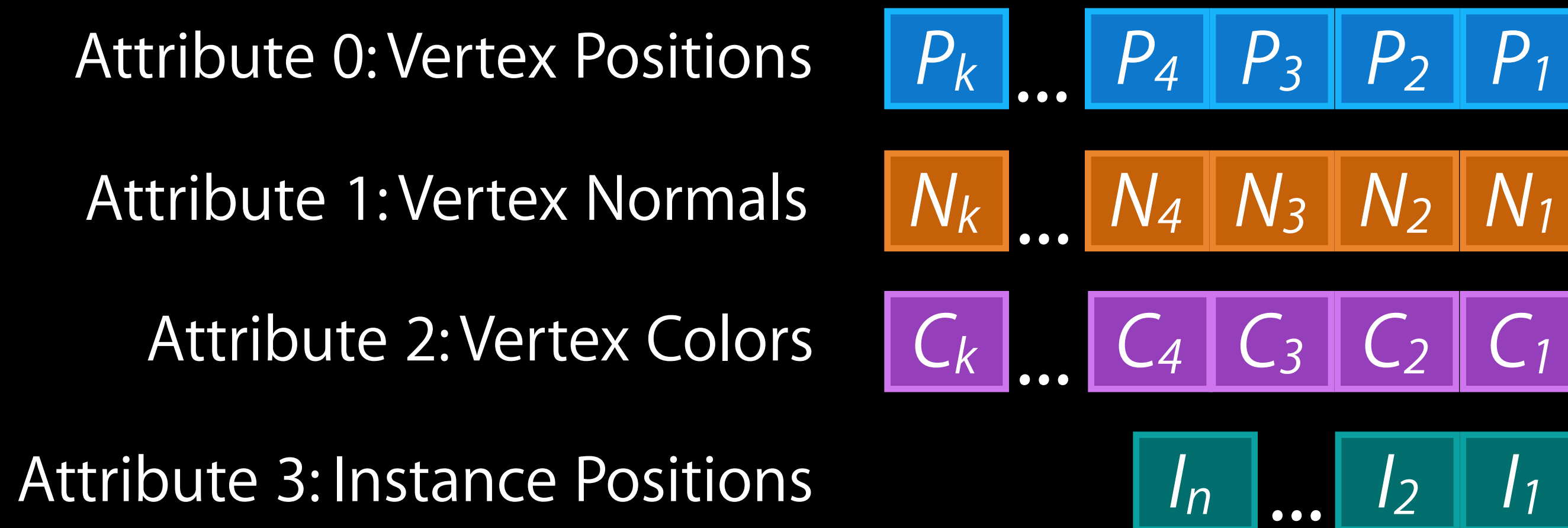
Instancing

Method 1: Instanced arrays

```
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
glVertexAttribDivisorAPPLE(3, 1);
```

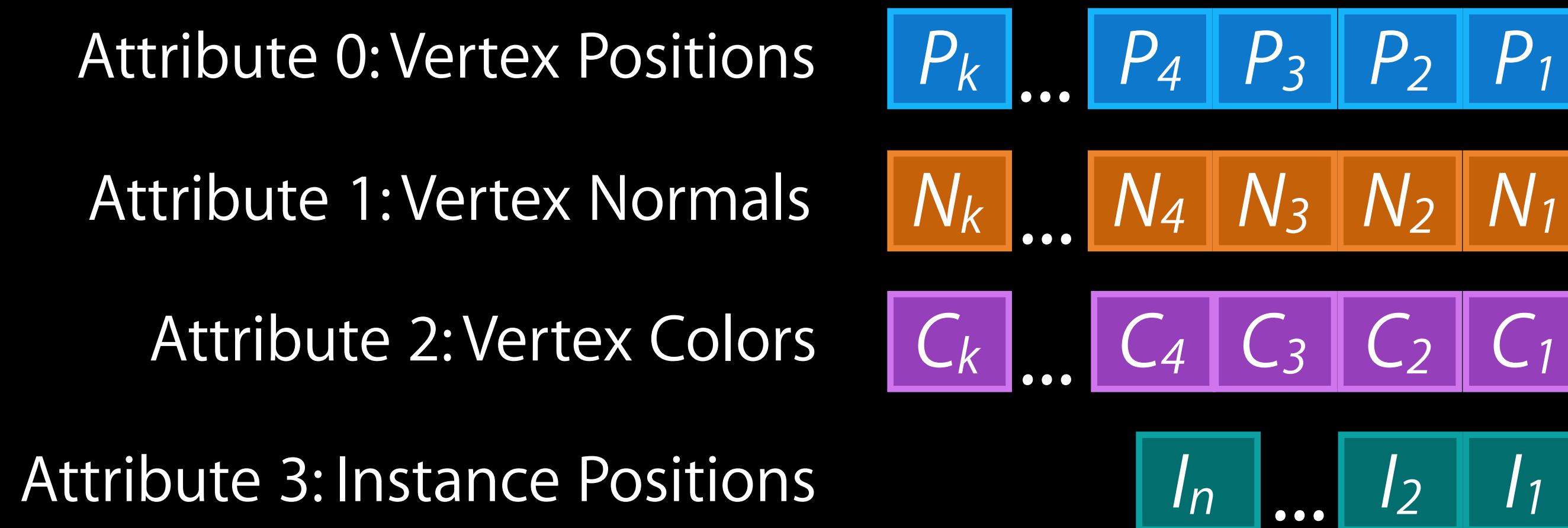
Vertex Arrays



Instancing

Method 1: Instanced arrays

Vertex Arrays

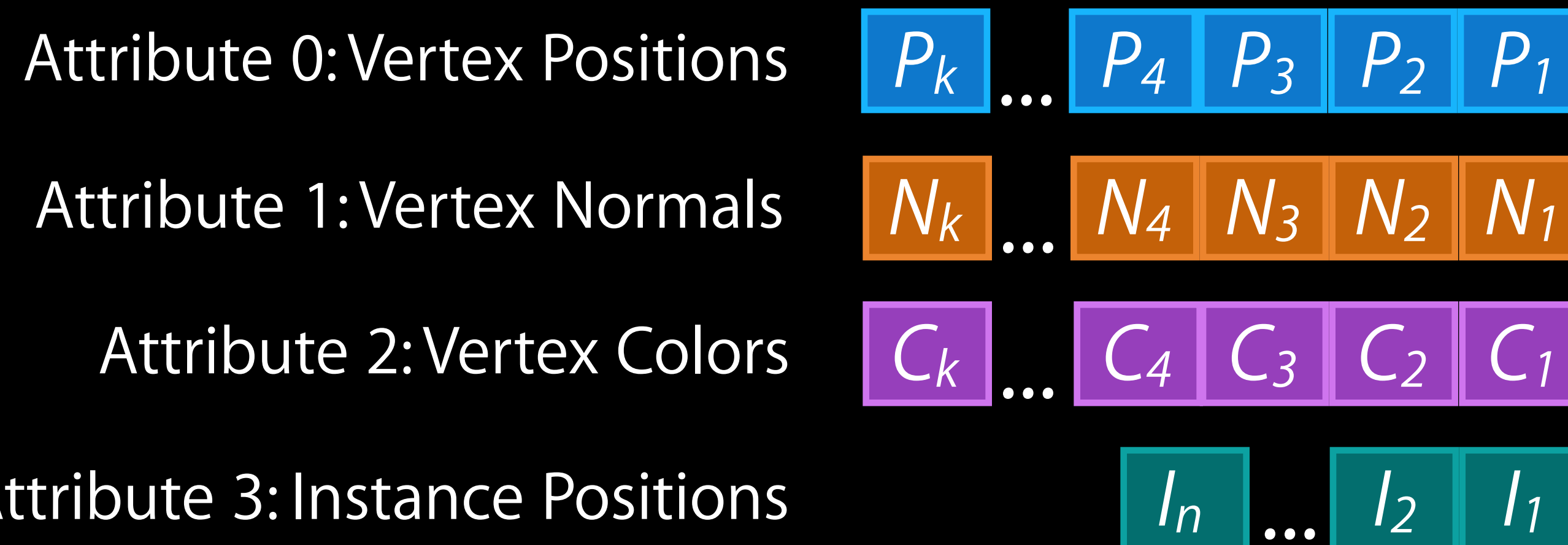


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

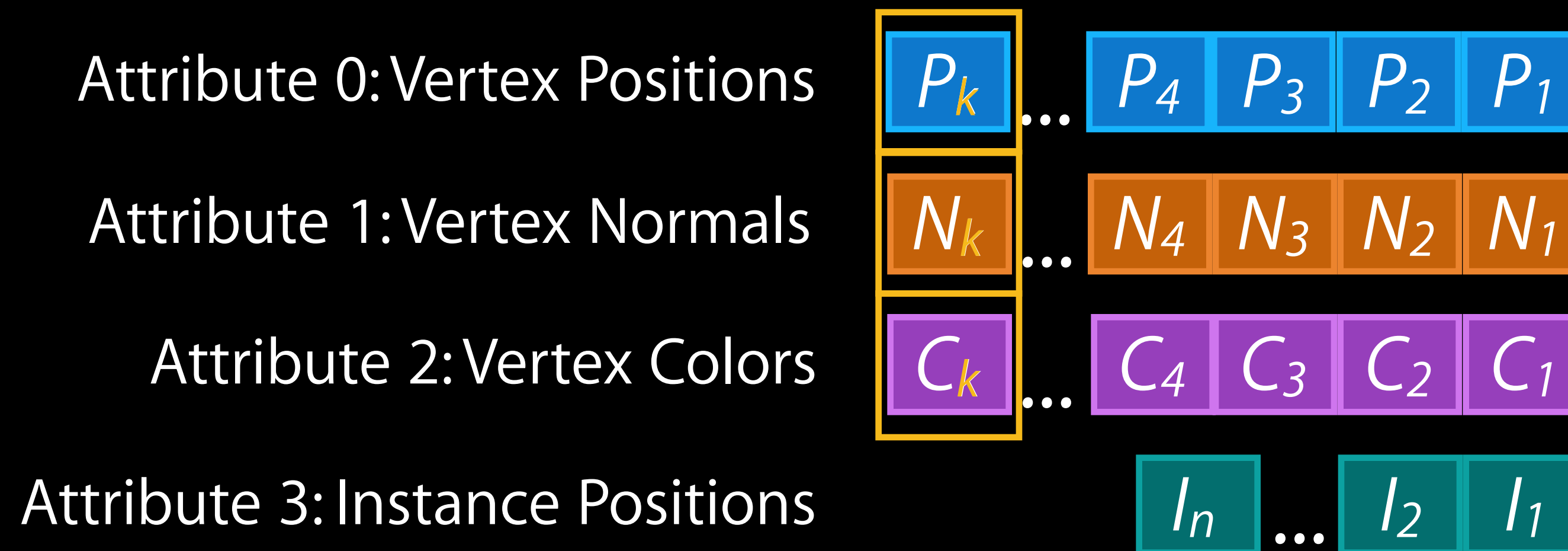


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

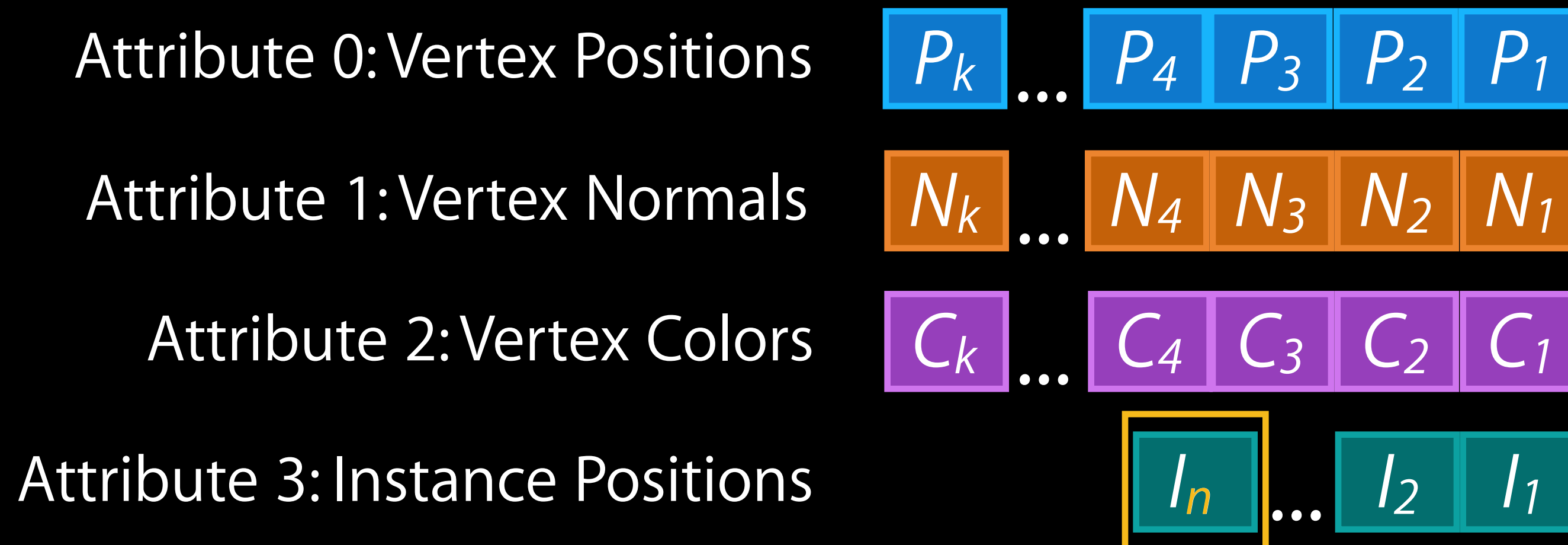


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

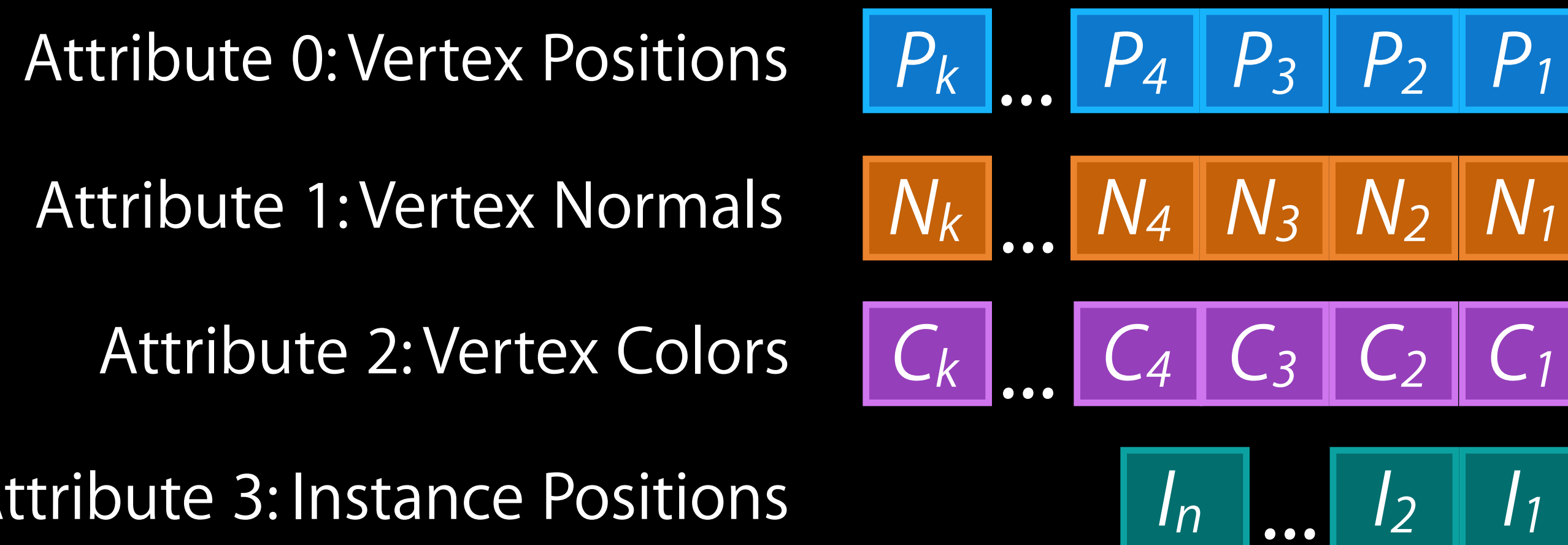


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

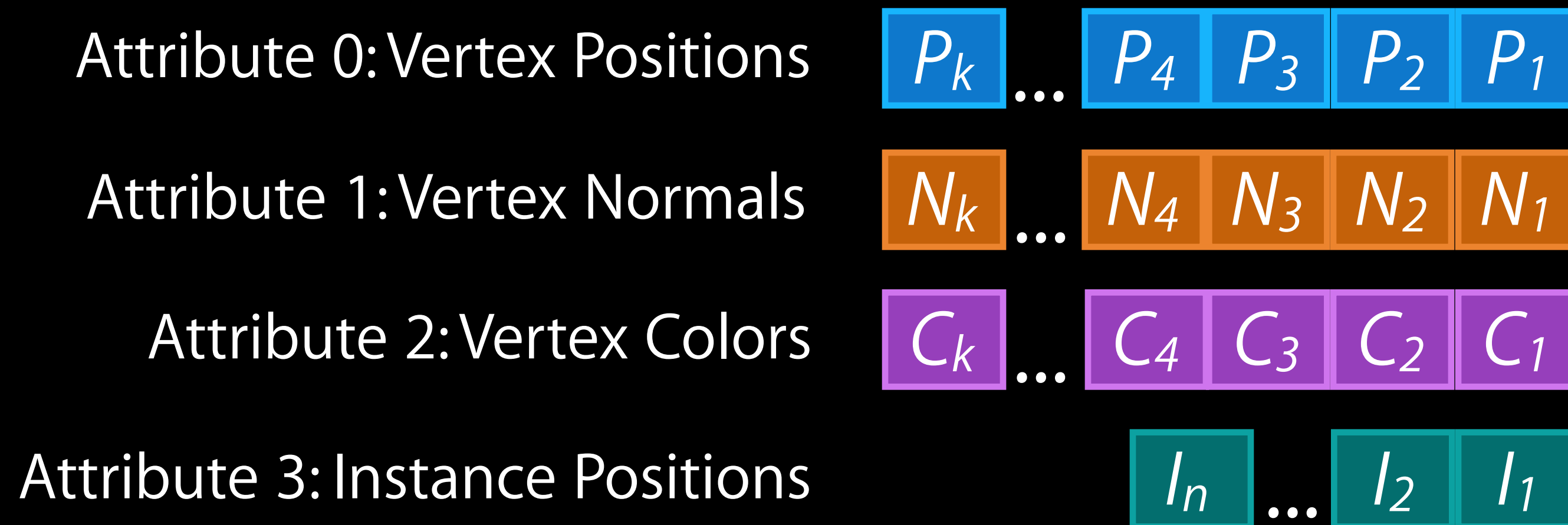


Instancing

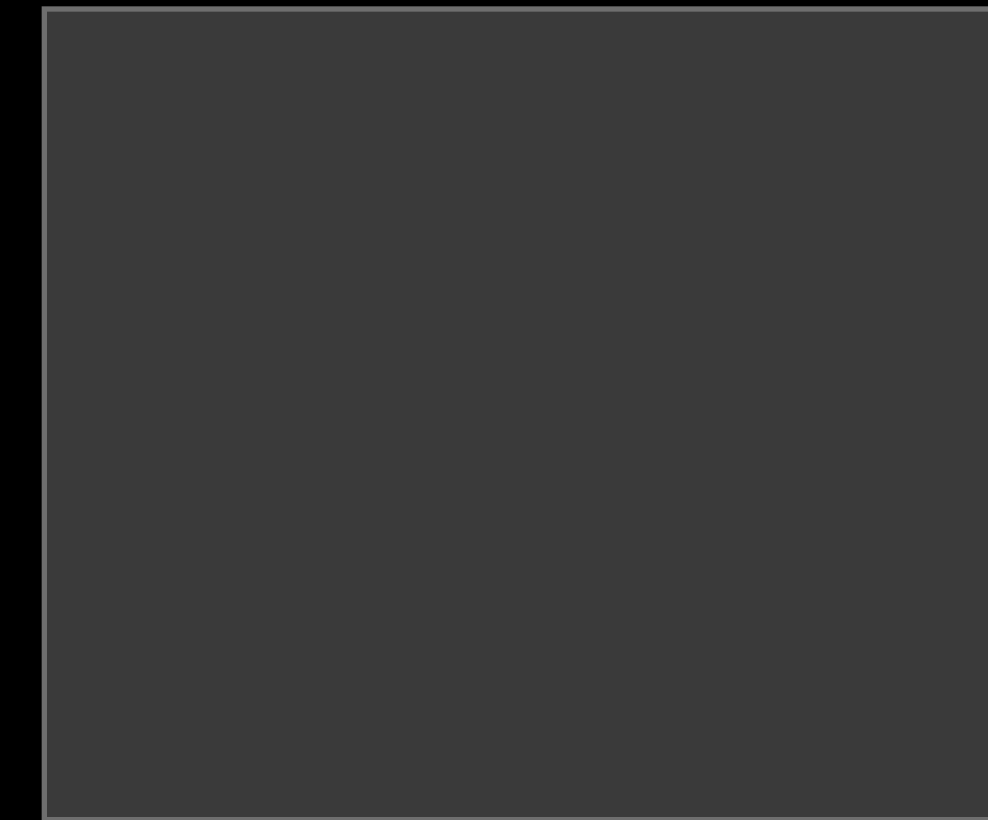
Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader



Instancing

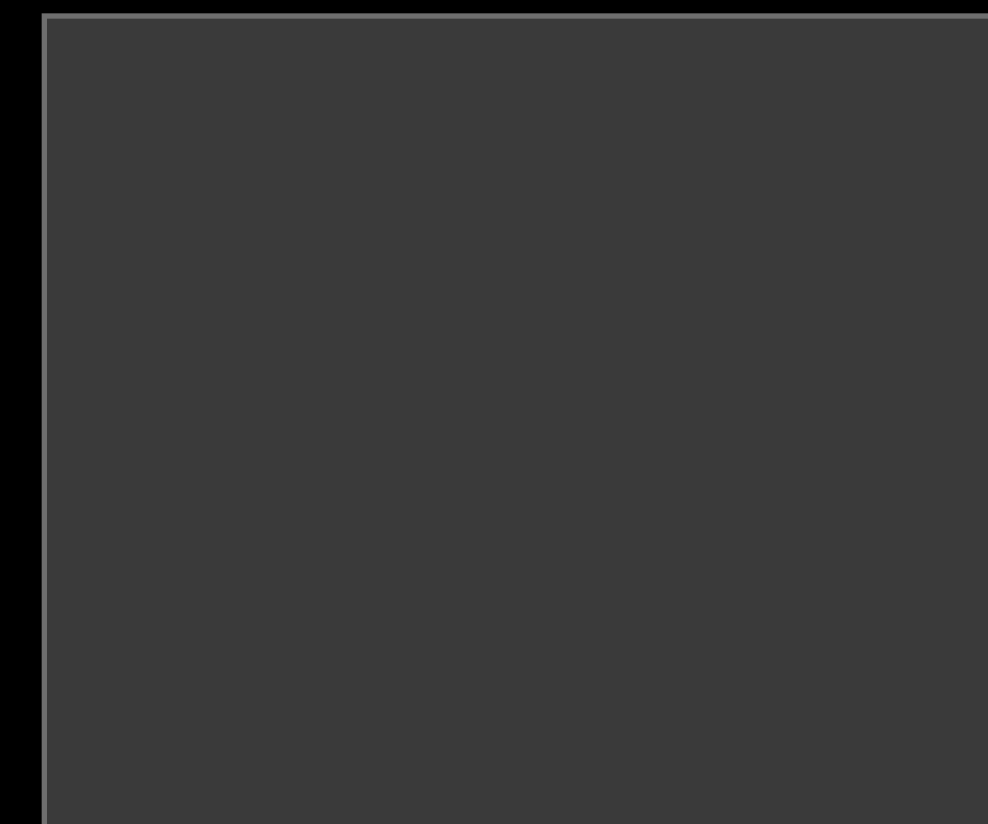
Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader

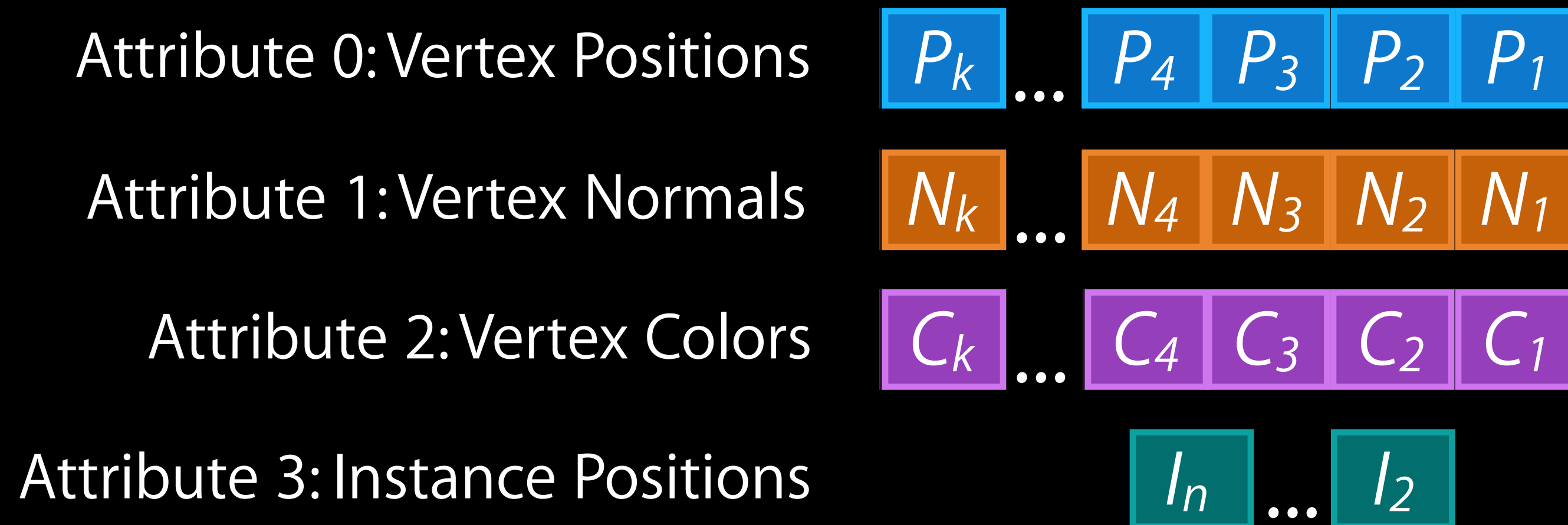


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader

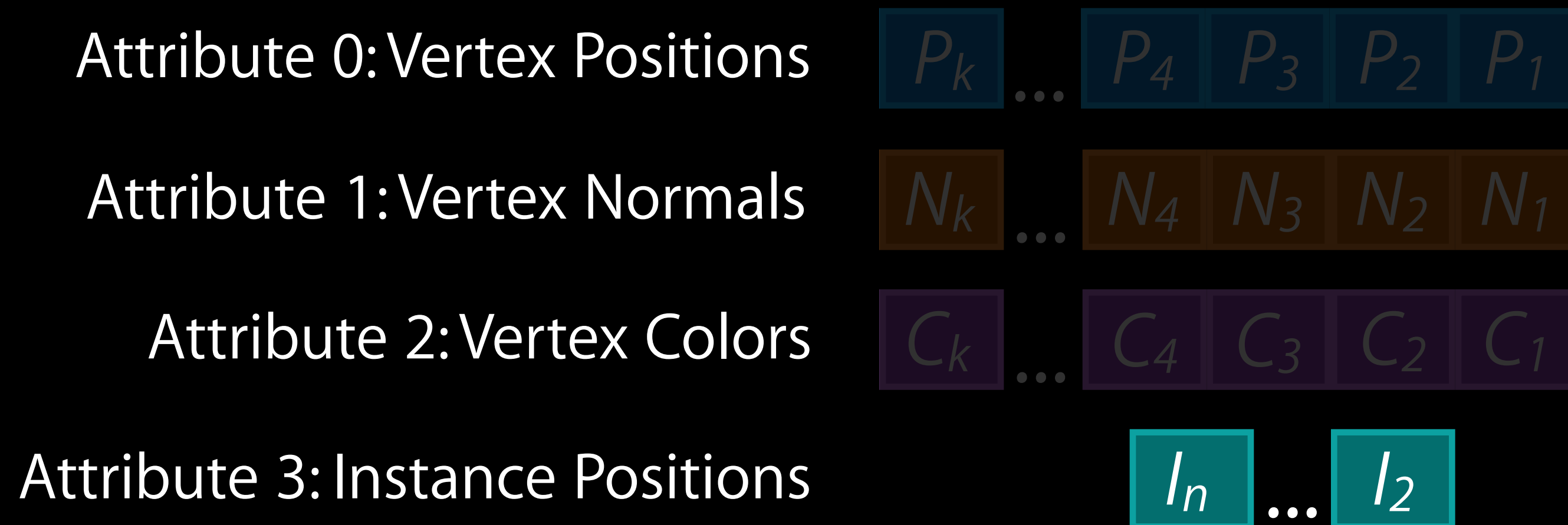


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader

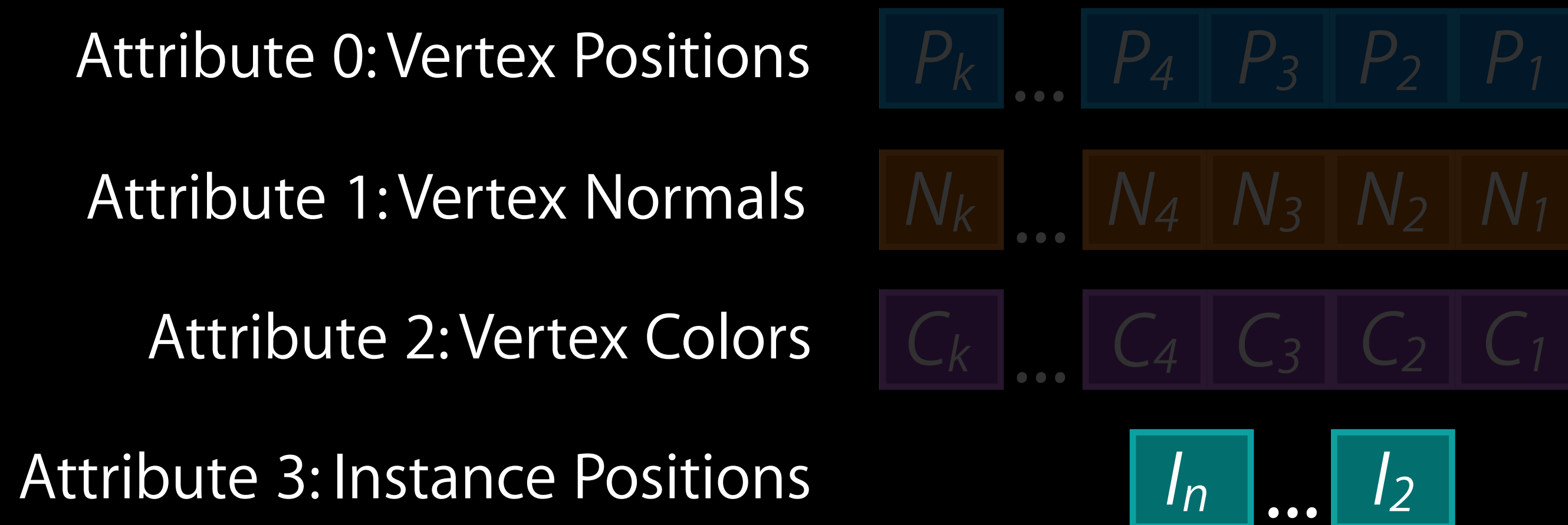


Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



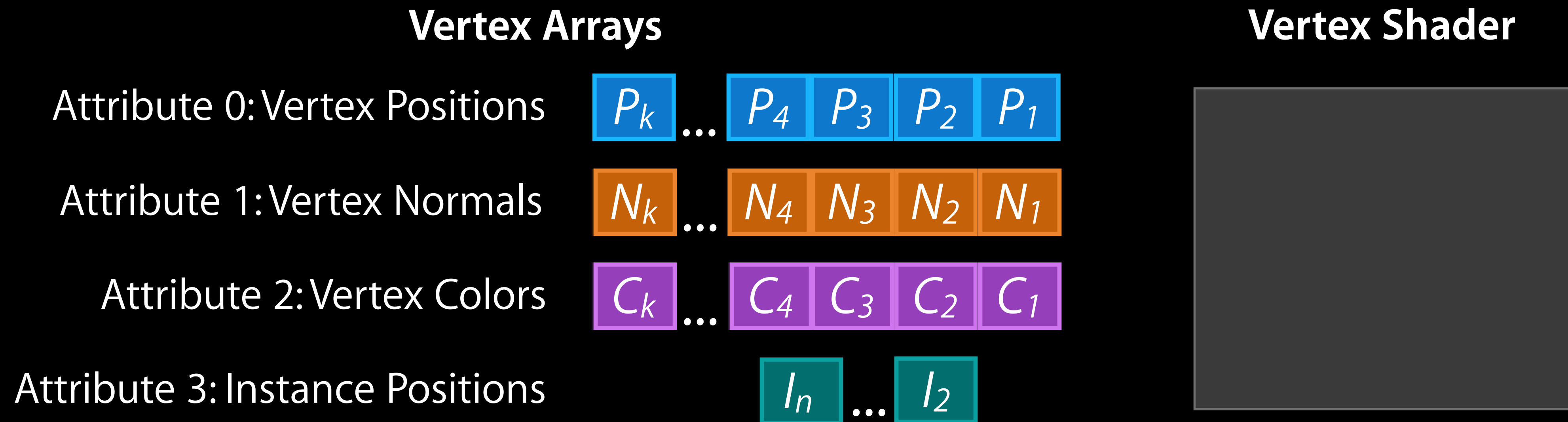
Vertex Shader



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

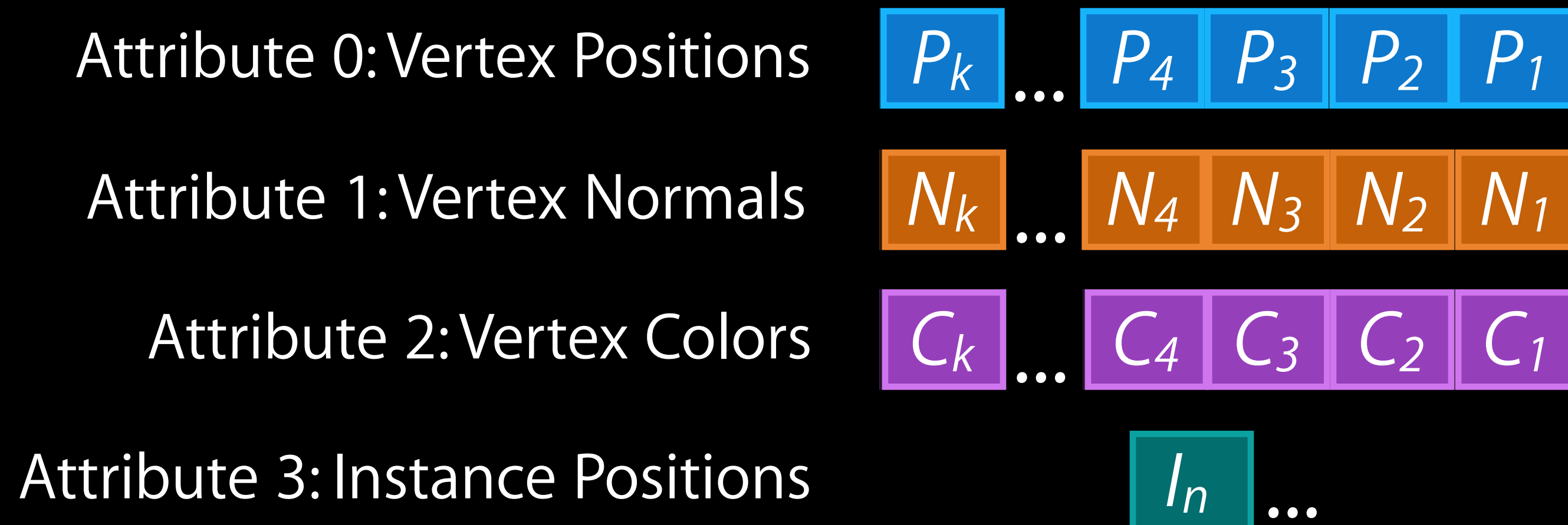


Instancing

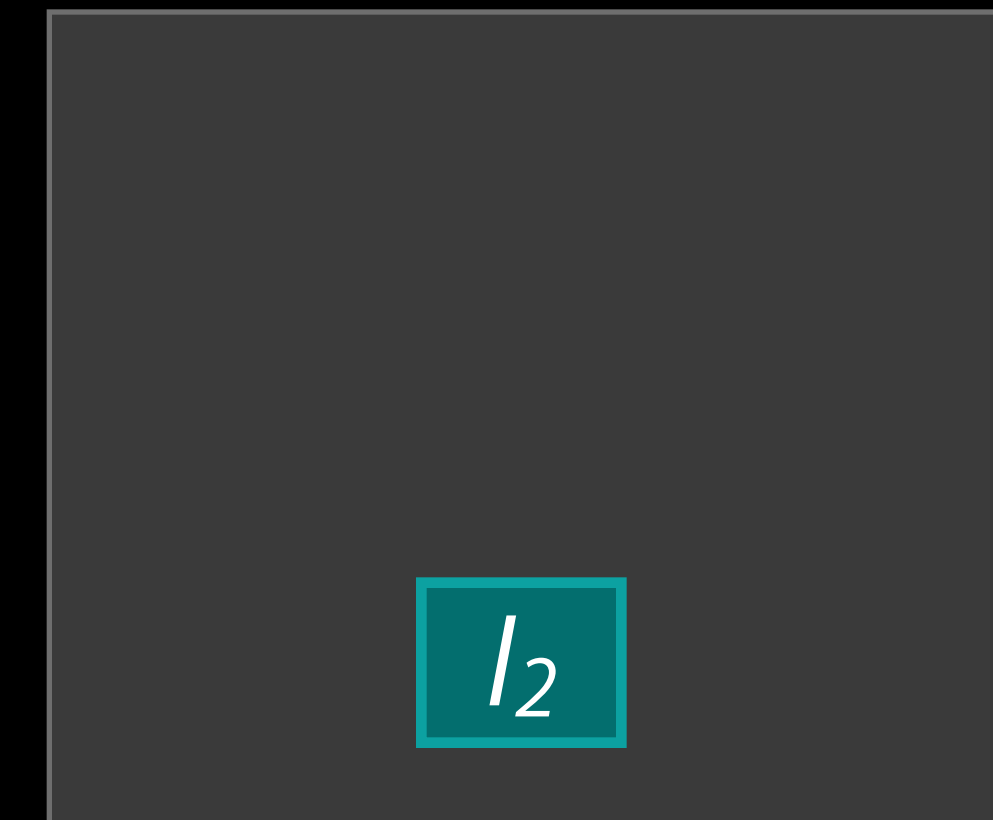
Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader



Instancing

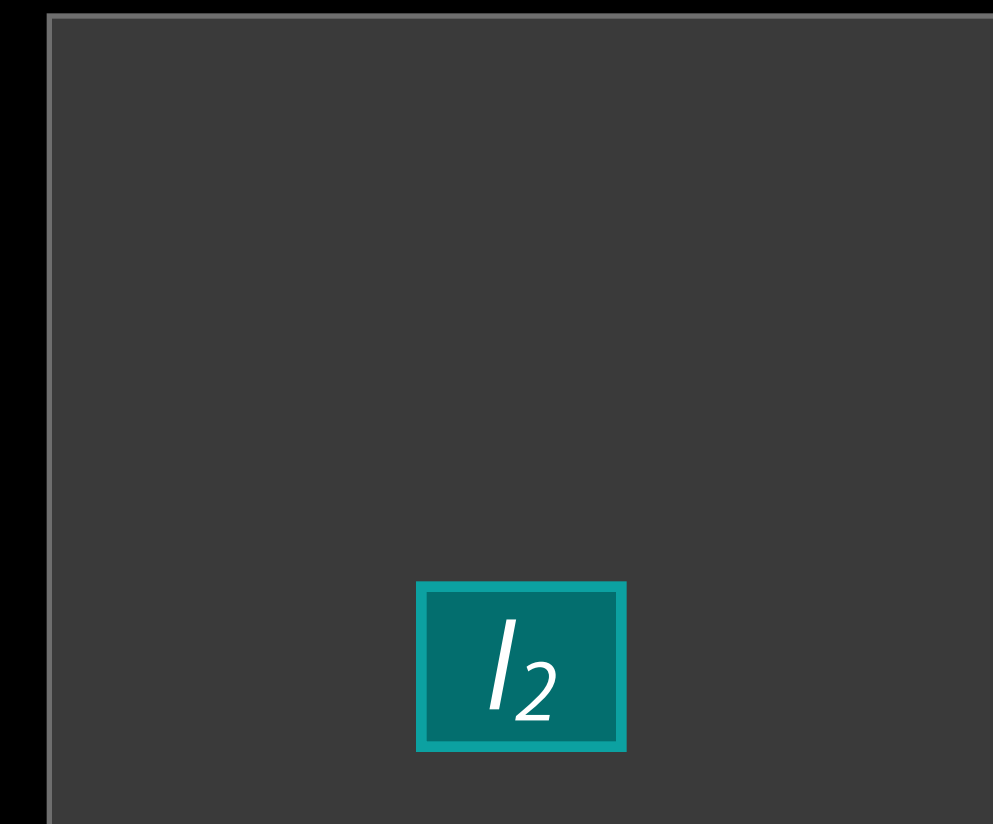
Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays



Vertex Shader



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

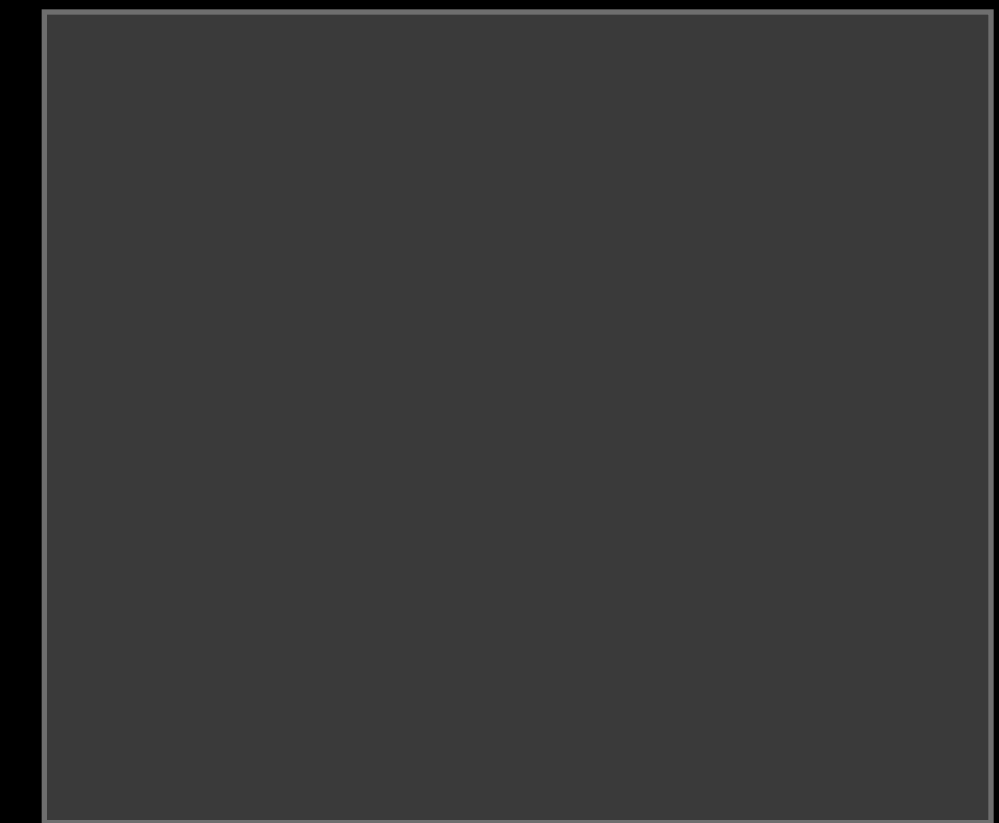
Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Attribute 3: Instance Positions

I_n ...

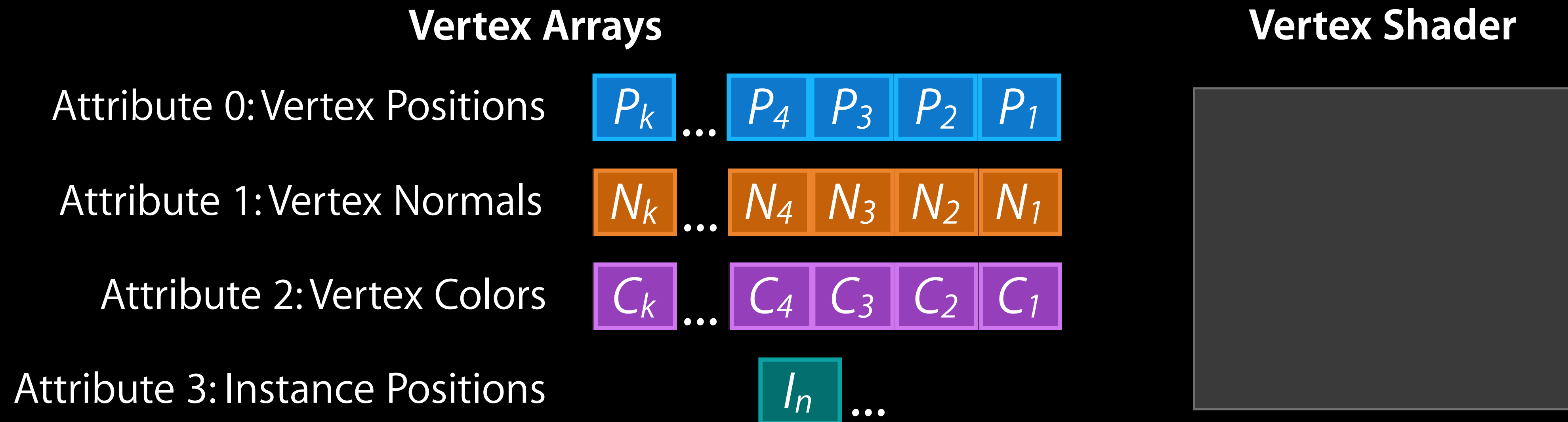
Vertex Shader



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Attribute 3: Instance Positions

Vertex Shader



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

Attribute 1: Vertex Normals

Attribute 2: Vertex Colors

Attribute 3: Instance Positions

Vertex Shader



Instancing

Method 1: Instanced arrays

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

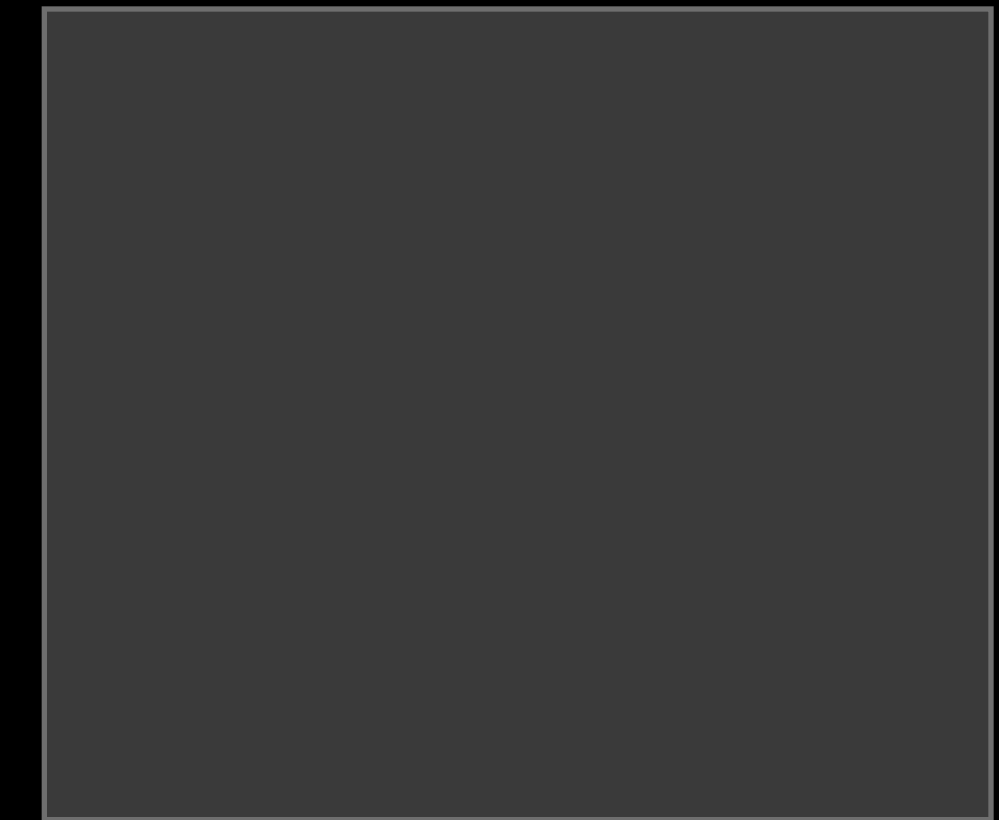
Attribute 0: Vertex Positions

Attribute 1: Vertex Normals

Attribute 2: Vertex Colors

Attribute 3: Instance Positions

Vertex Shader



Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually  
  
...  
  
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);  
  
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);  
  
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```

Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually  
  
...
```

```
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);
```

```
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```

Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually
```

```
...
```

```
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);
```

```
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```


Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually
```

```
...
```

```
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);
```

```
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);
```

```
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```

Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually  
  
...  
  
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);  
  
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);  
  
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```

Instanced Arrays

API setup



```
// Setup per vertex position, normal, and color arrays on  
// indices 0, 1, and 2 for gear model same as usually  
  
...  
  
// Setup vertex array index 3 to contain per object positions  
glVertexAttribPointer(3, ..., instancePositionOffset);  
  
// Indicate that attribute 3 should increment 1 element each instance  
glVertexAttribDivisorAPPLE(3, 1);  
  
// Draw gear model 100 times  
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, NUM_GEAR_VERTS, 100);
```

Instance Arrays

Vertex shader

```
// Per vertex and normal position
attribute vec4 pos;
attribute vec3 normal;
// Per instance position (maps to attribute index 3)
attribute vec4 instancePos;

...

void main()
{
    // Add vertex position to instance position
    vec4 tempPos = instancePos + pos;
    // Transform position to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Instance Arrays

Vertex shader

```
// Per vertex and normal position
attribute vec4 pos;
attribute vec3 normal;
// Per instance position (maps to attribute index 3)
attribute vec4 instancePos;

...

void main()
{
    // Add vertex position to instance position
    vec4 tempPos = instancePos + pos;
    // Transform position to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Instance Arrays

Vertex shader

```
// Per vertex and normal position
```

```
attribute vec4 pos;
```

```
attribute vec3 normal;
```

```
// Per instance position (maps to attribute index 3)
```

```
attribute vec4 instancePos;
```

```
...
```

```
void main()
```

```
{
```

```
// Add vertex position to instance position
```

```
vec4 tempPos = instancePos + pos;
```

```
// Transform position to clip space and output to gl_Position
```

```
...
```

```
gl_Position = tempPos;
```

```
}
```

Instance Arrays

Vertex shader

```
// Per vertex and normal position
attribute vec4 pos;
attribute vec3 normal;
// Per instance position (maps to attribute index 3)
attribute vec4 instancePos;

...

void main()
{
    // Add vertex position to instance position
    vec4 tempPos = instancePos + pos;
    // Transform position to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Instance Arrays

Vertex shader

```
// Per vertex and normal position
attribute vec4 pos;
attribute vec3 normal;
// Per instance position (maps to attribute index 3)
attribute vec4 instancePos;

...

void main()
{
    // Add vertex position to instance position
    vec4 tempPos = instancePos + pos;
    // Transform position to clip space and output to gl_Position
    ...
    gl_Position = tempPos;
}
```


Instance Arrays

Vertex shader

```
// Per vertex and normal position
attribute vec4 pos;
attribute vec3 normal;
// Per instance position (maps to attribute index 3)
attribute vec4 instancePos;

...

void main()
{
    // Add vertex position to instance position
    vec4 tempPos = instancePos + pos;
    // Transform position to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Instancing

Method 2: Instance ID

- Built-in `gl_InstanceIDAPPLE` variable in vertex shader
- ID incremented for each instance
- Can use ID to calculate unique info for instance
 - Calculate texture or position
 - Location of instance parameter in uniform array or texture
- Also uses `glDrawArraysInstancedAPPLE` or `glDrawElementsInstancedAPPLE`

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Vertex Shader

```
gl_InstanceID
```

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Vertex Shader

gl_InstanceID

0

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Vertex Shader

```
gl_InstanceID
```

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Vertex Shader

```
gl_InstanceID  
1
```

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

P_k ... P_4 P_3 P_2 P_1

Attribute 1: Vertex Normals

N_k ... N_4 N_3 N_2 N_1

Attribute 2: Vertex Colors

C_k ... C_4 C_3 C_2 C_1

Vertex Shader

```
gl_InstanceID
```

Instancing

Method 2: Instance ID

```
glDrawArraysInstancedAPPLE(GL_TRIANGLES, 0, k, n);
```

Vertex Arrays

Attribute 0: Vertex Positions

Attribute 1: Vertex Normals

Attribute 2: Vertex Colors

Vertex Shader

```
gl_InstanceID  
n
```


Shader Instance ID

Vertex shader



```
uniform float gearSize;
attribute vec2 vertexPosition;

...

void main() {
    float instanceID = float(gl_InstanceIDAPPLE);
    instancePosition.x = mod(instanceID, 10) * gearSize;
    instancePosition.y = (floor(instanceID)/ 10) * gearSize;
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;

    // Transform tempPos to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;  
attribute vec2 vertexPosition;
```

```
...
```

```
void main() {
```

```
    float instanceID = float(gl_InstanceIDAPPLE);
```

```
    instancePosition.x = mod(instanceID, 10) * gearSize;
```

```
    instancePosition.y = (floor(instanceID) / 10) * gearSize;
```

```
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;
```

```
    // Transform tempPos to clip space and output to gl_Position
```

```
    ...
```

```
    gl_Position = tempPos;
```

```
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;
attribute vec2 vertexPosition;

...

void main() {
    float instanceID = float(gl_InstanceIDAPPLE);
    instancePosition.x = mod(instanceID, 10) * gearSize;
    instancePosition.y = (floor(instanceID) / 10) * gearSize;
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;

    // Transform tempPos to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;
attribute vec2 vertexPosition;
...

void main() {
    float instanceID = float(gl_InstanceIDAPPLE);
    instancePosition.x = mod(instanceID, 10) * gearSize;
    instancePosition.y = (floor(instanceID) / 10) * gearSize;
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;

    // Transform tempPos to clip space and output to gl_Position
    ...

    gl_Position = tempPos;
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;
attribute vec2 vertexPosition;

...

void main() {
    float instanceID = float(gl_InstanceIDAPPLE);
    instancePosition.x = mod(instanceID, 10) * gearSize;
    instancePosition.y = (floor(instanceID)/ 10) * gearSize;
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;

    // Transform tempPos to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;  
attribute vec2 vertexPosition;
```

```
...
```

```
void main() {  
    float instanceID = float(gl_InstanceIDAPPLE);  
    instancePosition.x = mod(instanceID, 10) * gearSize;  
    instancePosition.y = (floor(instanceID) / 10) * gearSize;  
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;
```

```
// Transform tempPos to clip space and output to gl_Position
```

```
...
```

```
gl_Position = tempPos;
```

```
}
```

Shader Instance ID

Vertex shader



```
uniform float gearSize;
attribute vec2 vertexPosition;

...

void main() {
    float instanceID = float(gl_InstanceIDAPPLE);
    instancePosition.x = mod(instanceID, 10) * gearSize;
    instancePosition.y = (floor(instanceID)/ 10) * gearSize;
    vec4 tempPos = vertexPosition.xy + instancePosition.xy;

    // Transform tempPos to clip space and output to gl_Position

    ...

    gl_Position = tempPos;
}
```

Vertex Texture Sampling

A texture in the vertex stage?

Vertex Texture Sampling



- Texture access in the vertex shader
- Many uses
 - Displacement mapping
 - Alternative to uniforms
- Generic data store with random access

A Simple Vertex Texture Sampling Example

Height mapping



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;
uniform sampler2D heightMap;
...
main()
{
    vec4 tempPos = texture2D(heightMap, xzPos);
    tempPos.xz = xzPos;

    // Transform tempPos to from model space to
    // clip space with modelviewProjection matrix
    ...

    // output to gl_Position
    gl_Position = tempPos;
}
```



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;  
uniform sampler2D heightMap;  
  
...  
  
main()  
{  
    vec4 tempPos = texture2D(heightMap, xzPos);  
    tempPos.xz = xzPos;  
  
    // Transform tempPos to from model space to  
    // clip space with modelviewProjection matrix  
  
    ...  
  
    // output to gl_Position  
    gl_Position = tempPos;  
}
```



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;  
uniform sampler2D heightMap;  
...  
main()  
{  
    vec4 tempPos = texture2D(heightMap, xzPos);  
    tempPos.xz = xzPos;  
  
    // Transform tempPos to from model space to  
    // clip space with modelviewProjection matrix  
  
    ...  
  
    // output to gl_Position  
    gl_Position = tempPos;  
}
```



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;
uniform sampler2D heightMap;
...
main()
{
    vec4 tempPos = texture2D(heightMap, xzPos);
    tempPos.xz = xzPos;

    // Transform tempPos to from model space to
    // clip space with modelviewProjection matrix
    ...

    // output to gl_Position
    gl_Position = tempPos;
}
```



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;
uniform sampler2D heightMap;
...
main()
{
    vec4 tempPos = texture2D(heightMap, xzPos);
    tempPos.xz = xzPos;

    // Transform tempPos to from model space to
    // clip space with modelviewProjection matrix
    ...

    // output to gl_Position
    gl_Position = tempPos;
}
```

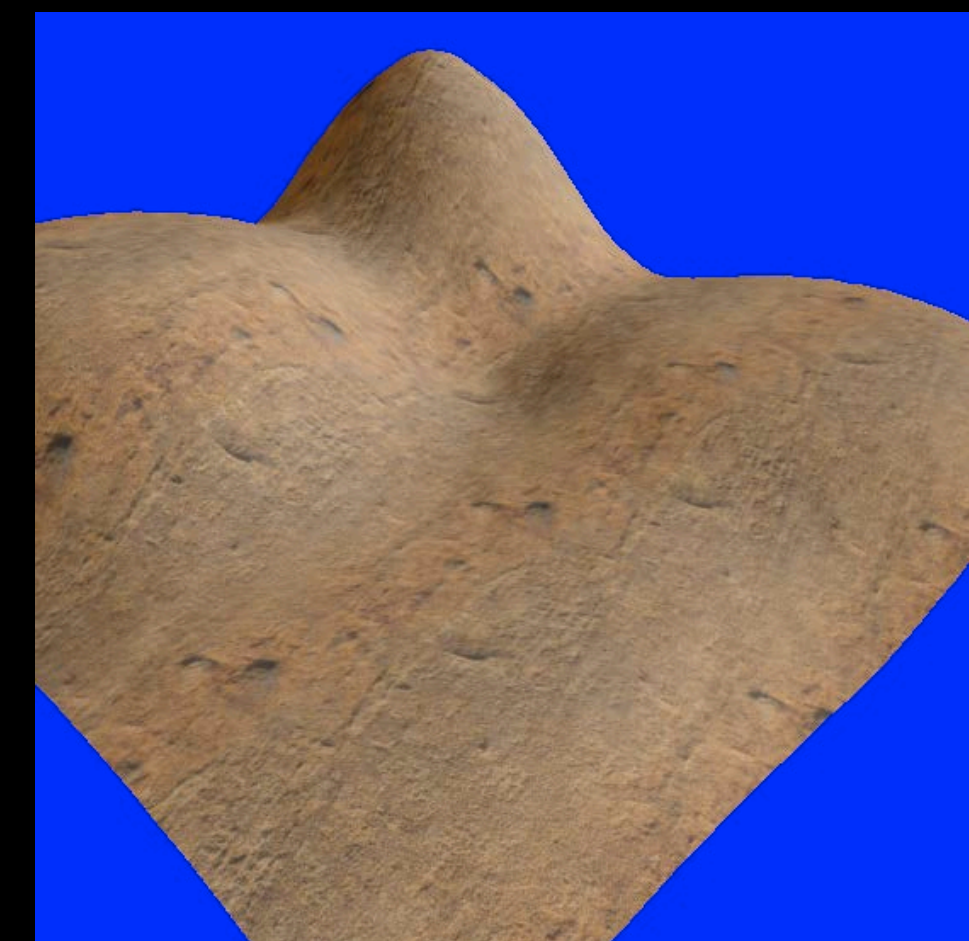


A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;  
uniform sampler2D heightMap;  
  
...  
  
main()  
{  
    vec4 tempPos = texture2D(heightMap, xzPos);  
    tempPos.xz = xzPos;
```

```
// Transform tempPos to from model space to  
// clip space with modelviewProjection matrix  
  
...  
  
// output to gl_Position  
gl_Position = tempPos;
```

```
}
```



A Simple Vertex Texture Sampling Example

```
attribute vec2 xzPos;
uniform sampler2D heightMap;
...
main()
{
    vec4 tempPos = texture2D(heightMap, xzPos);
    tempPos.xz = xzPos;

    // Transform tempPos to from model space to
    // clip space with modelviewProjection matrix
    ...

    // output to gl_Position
    gl_Position = tempPos;
}
```



Vertex Texture Sampling for Generic Data

- Can store any kind of data in a texture for shader access
 - Large store of random access memory
 - Data normally passed via `glUniform`, passed via texture

Texture Sampling vs. Uniforms

- Larger storage than a uniform array
- Potentially less API calls to set data
 - Bind large number of uniforms as group
- A variety of types
 - `GL_UNSIGNED_BYTE`, `GL_HALF_FLOAT`, and `GL_FLOAT`
- Use with filtering and wrapping
 - Averages sequential values
- Render to texture
 - GPU can produce data

Demo

Instanced Asteroids

Instanced Asteroids

Instancing with shader instance ID

- Calculates transformation matrix in vertex shader
 - `gl_InstanceID` modded by a constant gives a spin value
 - `cos` and `sin` used to generate rotation matrix
 - `gl_InstanceID` also used to calculate position
 - Translation applied to rotation matrix
- Matrix calculations are done per vertex

Instanced Asteroids

Instancing with instanced arrays

- Two vertex arrays store positions and rotations
 - Calculated once at app startup
- Attribute divisor passes parameters for each asteroid
 - Not passed per vertex

Instanced Asteroids

Instance ID with texture sampling vs. instanced arrays

- Advantages of instance ID
 - Little memory or bandwidth required
 - Use GPU for computation

Instanced Asteroids

Instance ID with texture sampling vs. instanced arrays

- Advantages of instance ID
 - Little memory or bandwidth required
 - Use GPU for computation
- Advantages of instanced arrays
 - Generally faster than computing on GPU
 - Lots of flexibility in types

Instanced Asteroids

Instance ID with texture sampling vs. instanced arrays

- Advantages of instance ID
 - Little memory or bandwidth required
 - Use GPU for computation
- Advantages of instanced arrays
 - Generally faster than computing on GPU
 - Lots of flexibility in types
- Advantages of using instance ID with vertex texture sampling
 - Random access
 - Often logically simplest to store data
 - Look up tables, bone matrices

Instancing and Vertex Texture Sampling

Summary

- Instancing
 - Many models, single draw
 - Different parameters each instance
 - Performance gains
- Vertex texture sampling
 - Texture data access in vertex shader
 - Use with instance ID for per-instance parameters
- Available on all iOS 7 devices

sRGB

The other colorspace

sRGB Texture Formats

Gamma encoded texture formats

- sRGB is an alternate colorspace
- More perceptually correct
 - Matches gamma curve of displays
- Perceptually linear color mixing
 - Mixing with RGB weighs brighter colors more than darker ones



sRGB



Linear

sRGB Texture Formats

In the API

- New formats
 - `GL_SRGB_EXT` // External Format
 - `GL_SRGB_ALPHA_EXT` // External Format
 - `GL_SRGB8_ALPHA8_EXT` // Internal Format
 - `GL_COMPRESSED_SRGB_PVRTC_..._APPLE` // Compressed Formats
- Noncompressed sRGB textures are renderable
 - Linear blending
 - Color calculations in shaders
- Check for `GL_EXT_sRGB` in the extension string
 - Not supported on iPhone 4

sRGB Texture Formats

When they should be used

- The perceptually “correct” colorspace to use for mixing
- Author texture images with sRGB colorspace in mind
 - Otherwise may look different than intended
- Only use for color data

GPU Tools

Building a solid foundation

OpenGL ES Frame Debugger

The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with the title bar indicating "InstancedAsteroids.xcodeproj - InstancingDemo.gputrace". The main view is split into three panels:

- Left Panel (Issues):** A list of 16 issues under the heading "InstancingDemo". The issues include "Logical Buffer Load", "Redundant Call", "Mipmapping Usage", and "Dependent Texture Sampling".
- Center Panel (3D View):** Two side-by-side 3D renderings of a scene. The left rendering shows a green sphere in a space environment with stars and a nebula. The right rendering shows the same scene with a white background and a green sphere. A "Color" control panel is visible below the renderings, showing a range from 0 to 1.
- Right Panel (Shader Editor):** A code editor showing a fragment shader. The code includes comments and a main function. A red error message is visible: "Use of undeclared identifier 'foo'".

At the bottom, there are two panels:

- GL Context:** A list of GL state settings, including "Error Status No GL Error", "Viewport (0, 0, 2048, 1536) - (0, 1)", "Active Texture Unit Texture Unit 0", "Stencil Off", "Blending Off", "Depth Less - (Write On) - Clear(1.000000)", "Culling Back - CCW", "Framebuffer Write - RGBA, Clear (0.5, 0.5, 0.5, 1)", "Polygon Offset Off", "Multisampling Default Coverage", "Scissor Off", "Misc", and "Current Vertex".
- Program #12:** A list of uniforms and attributes for the current program. It includes "Uniforms 6 Uniforms" with values for "amb", "dif", "ModelViewProjection", "ModelView", "lightPos", and "tex". It also lists "Active Attributes 2 Attributes", "Attribute Bindings 1 Bindings", and "Frame Statistics 30003 draw calls".

OpenGL ES Frame Debugger

Scrubber bar and Framebuffer view

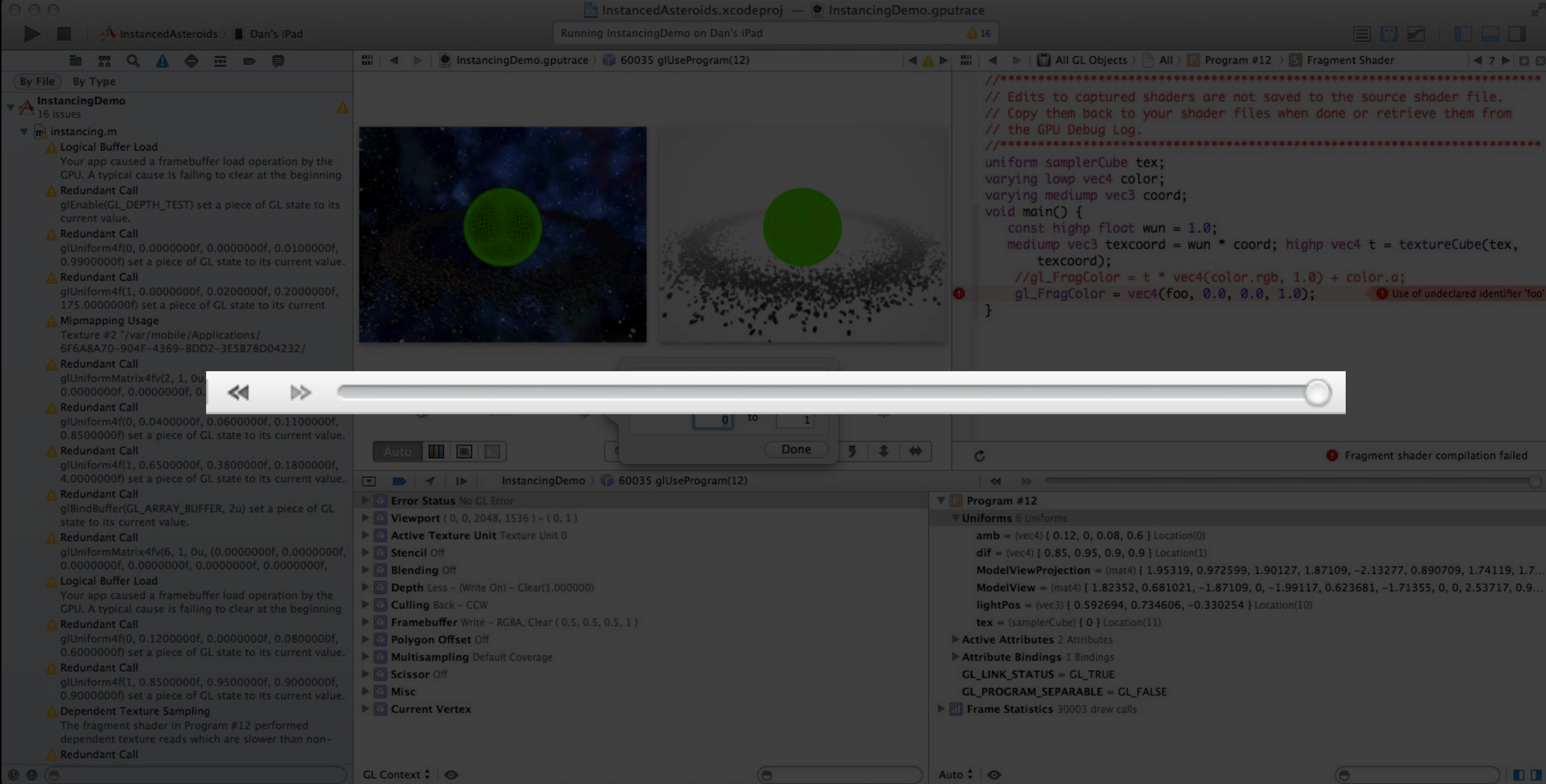
The screenshot displays the OpenGL ES Frame Debugger interface. The main window shows a scene with a green sphere and a scrubber bar. The scrubber bar is located at the bottom of the main window and is used to navigate through the frame buffer. The scrubber bar has a play button, a scrubber, and a stop button. The scrubber is currently positioned at the beginning of the frame buffer. The scrubber bar also has a 'Color' button and a 'Range' button. The 'Color' button is currently selected and shows a color picker. The 'Range' button is currently set to 0 to 1. The scrubber bar also has a 'Done' button. The scrubber bar is currently showing the first frame of the scene.

The interface is divided into several panels:

- Left Panel (Issues):** Lists 16 issues, including "Logical Buffer Load", "Redundant Call", "Mipmapping Usage", and "Dependent Texture Sampling".
- Center Panel (Scene):** Shows two views of the scene: a 3D view on the left and a 2D view on the right. A "Color" dialog box is open over the 2D view, showing a range from 0 to 1.
- Right Panel (Code):** Displays the source code for the fragment shader, showing a compilation error: "Use of undeclared identifier 'foo'".
- Bottom Panel (GL Context):** Shows the current GL context settings, including "Error Status", "Viewport", "Active Texture Unit", "Stencil", "Blending", "Depth", "Culling", "Framebuffer", "Polygon Offset", "Multisampling", "Scissor", "Misc", and "Current Vertex".
- Bottom Right Panel (Program #12):** Shows the state of the program, including uniforms, active attributes, attribute bindings, and frame statistics.

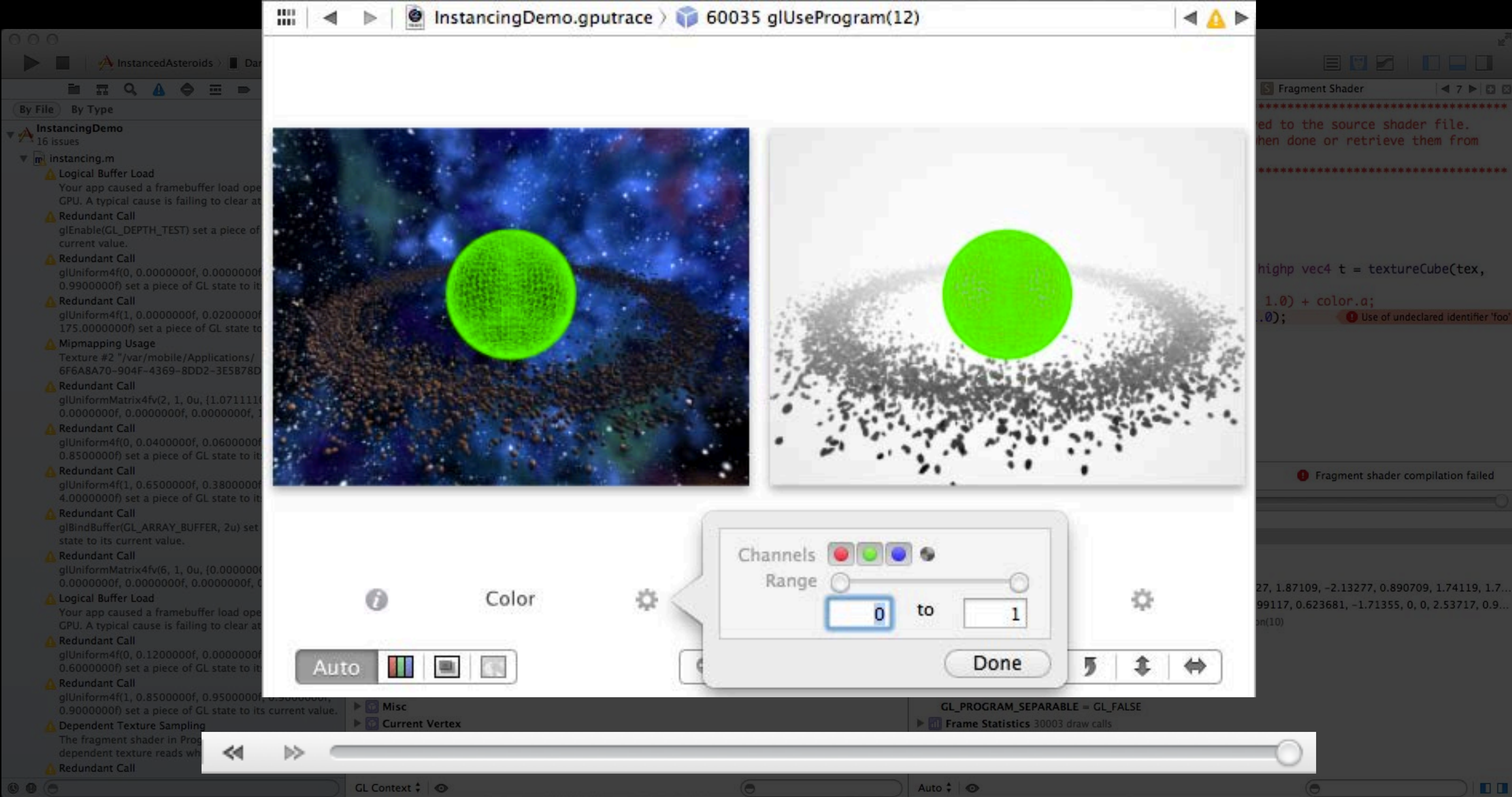
OpenGL ES Frame Debugger

Scrubber bar and Framebuffer view



OpenGL ES Frame Debugger

Scrubber bar and Framebuffer view



OpenGL ES Frame Debugger

Scrubber bar and Framebuffer view

The screenshot displays the OpenGL ES Frame Debugger interface. The main window shows a 3D scene with a green sphere and a starry background. A scrubber bar at the bottom allows navigation through the frame. The right side features a code editor showing the fragment shader source code, with a red error message: "Use of undeclared identifier 'foo'". The bottom right panel shows the state of Program #12, including uniforms and active attributes. The left panel lists 16 issues, such as "Logical Buffer Load" and "Redundant Call".

```
*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

Fragment shader compilation failed

Program #12

- Uniforms 6 Uniforms
 - amb = (vec4) { 0.12, 0, 0.08, 0.6 } Location(0)
 - dif = (vec4) { 0.85, 0.95, 0.9, 0.9 } Location(1)
 - ModelViewProjection = (mat4) { 1.95319, 0.972599, 1.90127, 1.87109, -2.13277, 0.890709, 1.74119, 1.7...
 - ModelView = (mat4) { 1.82352, 0.681021, -1.87109, 0, -1.99117, 0.623681, -1.71355, 0, 0, 2.53717, 0.9...
 - lightPos = (vec3) { 0.592694, 0.734606, -0.330254 } Location(10)
 - tex = (samplerCube) { 0 } Location(11)
- Active Attributes 2 Attributes
- Attribute Bindings 1 Bindings
 - GL_LINK_STATUS = GL_TRUE
 - GL_PROGRAM_SEPARABLE = GL_FALSE
- Frame Statistics 30003 draw calls

GL Context

OpenGL ES Frame Debugger

GL context state and auto context view

The screenshot displays the OpenGL ES Frame Debugger interface, showing a 3D scene with a green sphere and a starry background. The interface is divided into several panels:

- Left Panel (Issues):** Lists 16 issues, including "Logical Buffer Load", "Redundant Call", "Mipmapping Usage", and "Dependent Texture Sampling".
- Center Panel (3D View):** Shows two views of the scene: a perspective view on the left and a top-down view on the right. A "Color" control panel is visible below the views, showing a range from 0 to 1.
- Right Panel (Shader Editor):** Displays the source code for a Fragment Shader. The code includes comments and a main function. A red error message is visible: "Use of undeclared identifier 'foo'".
- Bottom Panel (GL Context State):** Shows the current state of the GL context, including "Error Status", "Viewport", "Active Texture Unit", "Blending", "Depth", "Culling", "Framebuffer", "Polygon Offset", "Multisampling", "Scissor", "Misc", and "Current Vertex".
- Bottom Right Panel (Program #12):** Shows the state of Program #12, including "Uniforms", "Active Attributes", "Attribute Bindings", and "Frame Statistics".

```
*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

Fragment shader compilation failed

OpenGL ES Frame Debugger

GL context state and auto context view

The screenshot displays the OpenGL ES Frame Debugger interface. The top bar shows the project name "InstancedAsteroids.xcodeproj" and the current frame "60035 glUseProgram(12)". The left sidebar lists 16 issues, including "Logical Buffer Load" and "Redundant Call". The central view shows a 3D scene with a green sphere and a blue sky. A white pop-up window displays the current GL context state:

- Error Status No GL Error
- Viewport (0, 0, 2048, 1536) - (0, 1)
- Active Texture Unit Texture Unit 0
- Stencil Off
- Blending Off
- Depth Less - (Write On) - Clear(1.000000)
- Culling Back - CCW
- Framebuffer Write - RGBA, Clear (0.5, 0.5, 0.5, 1)
- Polygon Offset Off
- Multisampling Default Coverage
- Scissor Off
- Misc
- Current Vertex

The right sidebar shows the current program state, including "ModelView = (mat4) { 1.82352, 0.681021, -1.87109, 0, -1.99117, 0.623681, -1.71355, 0, 0, 2.53717, 0.9...", "lightPos = (vec3) { 0.592694, 0.734606, -0.330254 } Location(10)", and "tex = (samplerCube) { 0 } Location(11)". A red error message at the bottom right indicates "Fragment shader compilation failed" with the message "Use of undeclared identifier 'foo'".

OpenGL ES Frame Debugger

GL context state and auto context view



The screenshot displays the OpenGL ES Frame Debugger interface. The main window shows a 3D scene of a planet in space. The interface includes a toolbar at the top, a left sidebar with a file explorer, and a central area for viewing the current frame. Two pop-up windows are overlaid on the main view:

- GL Context:** A window showing the current state of the OpenGL context. It lists various settings such as Viewport (0, 0, 2048, 1536) - (0, 1), Active Texture Unit (Texture Unit 0), Stencil (Off), Blending (Off), Depth (Less - (Write On) - Clear(1.000000)), Culling (Back - CCW), Framebuffer Write (RGBA, Clear (0.5, 0.5, 0.5, 1)), Polygon Offset (Off), Multisampling (Default Coverage), Scissor (Off), Misc, and Current Vertex.
- Program #12:** A window showing the state of the current program. It lists uniforms (amb, dif, ModelViewProjection, ModelView, lightPos, tex), active attributes, attribute bindings (GL_LINK_STATUS = GL_TRUE, GL_PROGRAM_SEPARABLE = GL_FALSE), and frame statistics (30003 draw calls).

The background interface also shows a list of issues on the left, including "Logical Buffer Load" and "Redundant Call". The top status bar indicates "Running InstancingDemo on Dan's iPad" and "60035 glUseProgram(12)".

OpenGL ES Frame Debugger

GL context state and auto context view



The screenshot displays the OpenGL ES Frame Debugger interface. The main window shows a 3D scene with a green sphere and a blue sky. The interface is divided into several panels:

- Issues Panel (Left):** Lists 16 issues, including "Logical Buffer Load" and "Redundant Call".
- GL Context State Panel (Bottom Left):** Shows the current GL context state, including:
 - Error Status: No GL Error
 - Viewport: (0, 0, 2048, 1536) - (0, 1)
 - Active Texture Unit: Texture Unit 0
 - Stencil: Off
 - Blending: Off
 - Depth: Less - (Write On) - Clear(1.000000)
 - Culling: Back - CCW
 - Framebuffer Write: - RGBA, Clear (0.5, 0.5, 0.5, 1)
 - Polygon Offset: Off
 - Multisampling: Default Coverage
 - Scissor: Off
 - Misc
 - Current Vertex
- Program #12 Panel (Bottom Right):** Shows the state of Program #12, including:
 - Uniforms: 6 Uniforms
 - amb = (vec4) { 0.12, 0, 0.08, 0.6 } Location(0)
 - dif = (vec4) { 0.85, 0.95, 0.9, 0.9 } Location(1)
 - ModelViewProjection = (mat4) { 1.95319, 0.972599, 1.90127, 1.87109, -2.13277, 0.890709, 1.74119, 1.7...
 - ModelView = (mat4) { 1.82352, 0.681021, -1.87109, 0, -1.99117, 0.623681, -1.71355, 0, 0, 2.53717, 0.9...
 - lightPos = (vec3) { 0.592694, 0.734606, -0.330254 } Location(10)
 - tex = (samplerCube) { 0 } Location(11)
 - Active Attributes: 2 Attributes
 - Attribute Bindings: 1 Bindings
 - GL_LINK_STATUS = GL_TRUE
 - GL_PROGRAM_SEPARABLE = GL_FALSE
 - Frame Statistics: 30003 draw calls

The "Auto" dropdown menu in the bottom right panel is highlighted with a yellow box.

OpenGL ES Frame Debugger

GL context state and auto context view



The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with a toolbar and navigation controls. The main area is divided into three panes:

- Left Pane (Issues):** Lists 16 issues for the "InstancingDemo" application, including "Logical Buffer Load", "Redundant Call", "Mipmapping Usage", and "Dependent Texture Sampling".
- Center Pane (Visuals):** Shows two side-by-side views of a 3D scene. The left view is a rendered scene with a green sphere and a starry background. The right view is a wireframe or debug view of the same scene. A "Color" control panel is overlaid on the bottom of this pane, showing a range from 0 to 1.
- Right Pane (Shader Editor):** Displays the source code for a "Fragment Shader". The code includes comments and a main function. A red error message is visible: "Use of undeclared identifier 'foo'".

At the bottom, the "GL Context" pane shows the current state of the graphics pipeline, including:

- Error Status: No GL Error
- Viewport: (0, 0, 2048, 1536) - (0, 1)
- Active Texture Unit: Texture Unit 0
- Stencil: Off
- Blending: Off
- Depth: Less - (Write On) - Clear(1.000000)
- Culling: Back - CCW
- Framebuffer: Write - RGBA, Clear (0.5, 0.5, 0.5, 1)
- Polygon Offset: Off
- Multisampling: Default Coverage
- Scissor: Off
- Misc
- Current Vertex

The "Program #12" pane shows the state of the active program, including uniforms, active attributes, and attribute bindings.

OpenGL ES Frame Debugger

Object view and Shader editor

The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with the title bar indicating "InstancedAsteroids.xcodeproj - InstancingDemo.gputrace". The main window is divided into several panels:

- Issues Panel (Left):** Lists 16 issues, including "Logical Buffer Load", "Redundant Call", "Mipmapping Usage", and "Dependent Texture Sampling".
- Object View (Center-Left):** Shows a 3D scene with a green sphere and a grey sphere, with a "Color" dialog box open over the green sphere.
- Shader Editor (Center-Right):** Displays the fragment shader code for "Program #12". A red error message highlights a "Use of undeclared identifier 'foo'" in the line `gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);`.
- GL Context (Bottom-Left):** Shows the current GL state, including "Error Status No GL Error", "Viewport (0, 0, 2048, 1536)", "Active Texture Unit Texture Unit 0", "Blending Off", "Depth Less - (Write On) - Clear(1.000000)", "Culling Back - CCW", "Framebuffer Write - RGBA, Clear (0.5, 0.5, 0.5, 1)", "Polygon Offset Off", "Multisampling Default Coverage", "Scissor Off", "Misc", and "Current Vertex".
- Program #12 (Bottom-Right):** Shows the uniform and attribute settings for the current program, including "Uniforms 6 Uniforms" and "Active Attributes 2 Attributes".

OpenGL ES Frame Debugger

Object view and Shader editor

The screenshot displays the OpenGL ES Frame Debugger interface. The main window shows a list of captured frames, with the current frame selected. The 'Object View' on the left shows the hierarchy of objects, including 'InstancingDemo' and 'instancing.m'. The 'Shader Editor' window is open, showing the source code of a fragment shader. The shader code is as follows:

```
//*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
//*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

The error message 'Use of undeclared identifier 'foo'' is highlighted in red. The 'Fragment Shader' window also shows a 'Fragment shader compilation failed' error message. The 'GL Context' window at the bottom shows the current state of the GPU, including 'GL_LINK_STATUS = GL_TRUE', 'GL_PROGRAM_SEPARABLE = GL_FALSE', and 'Frame Statistics 30003 draw calls'.

OpenGL ES Frame Debugger

Object view and Shader editor

The screenshot displays the OpenGL ES Frame Debugger interface. On the left, a list of 16 issues is shown, including 'Logical Buffer Load' and 'Redundant Call'. The main area is a shader editor window titled 'Fragment Shader' showing the following code:

```
//*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
//*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

A red error message is visible at the bottom of the shader editor: 'Fragment shader compilation failed' with a red exclamation mark icon. The error message also points to the line 'gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);' with the text 'Use of undeclared identifier 'foo''. A yellow circle highlights a refresh icon (a circular arrow) at the bottom left of the shader editor window.

OpenGL ES Frame Debugger

Object view and Shader editor

The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with the title bar indicating 'InstancedAsteroids.xcodeproj - InstancingDemo.gputrace'. The main window is divided into several panels:

- Issues Panel (Left):** Lists 16 issues, including 'Logical Buffer Load', 'Redundant Call', 'Mipmapping Usage', and 'Dependent Texture Sampling'. Each issue includes a brief description of the problem.
- Object View (Center-Left):** Shows a 3D scene with a green sphere and a grey sphere, set against a dark blue background with stars.
- Shader Editor (Center-Right):** Displays the source code for a fragment shader. A red error message is visible: 'Use of undeclared identifier 'foo''. The code includes comments and uniform declarations.
- GL Context (Bottom):** Shows the current state of the OpenGL context, including the viewport, active texture unit, stencil, blending, depth, culling, framebuffer, polygon offset, multisampling, scissor, and misc settings.

```
*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

OpenGL ES Frame Debugger

GL issues navigator

The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with the current frame being 60035. The left sidebar lists 16 issues, including Logical Buffer Load, Redundant Call, Mipmapping Usage, and Dependent Texture Sampling. The main view shows a 3D scene with a green sphere and a grey sphere, with a color picker dialog open. The right sidebar shows the current program's uniforms and frame statistics. The bottom view shows the GL context, including the viewport, active texture unit, and various state settings.

```
*****  
// Edits to captured shaders are not saved to the source shader file.  
// Copy them back to your shader files when done or retrieve them from  
// the GPU Debug Log.  
*****  
uniform samplerCube tex;  
varying lowp vec4 color;  
varying mediump vec3 coord;  
void main() {  
    const highp float wun = 1.0;  
    mediump vec3 texcoord = wun * coord; highp vec4 t = textureCube(tex,  
        texcoord);  
    //gl_FragColor = t * vec4(color.rgb, 1.0) + color.a;  
    gl_FragColor = vec4(foo, 0.0, 0.0, 1.0);  
}
```

Fragment shader compilation failed

Program #12

- Uniforms 6 Uniforms
 - amb = (vec4) { 0.12, 0, 0.08, 0.6 } Location(0)
 - dif = (vec4) { 0.85, 0.95, 0.9, 0.9 } Location(1)
 - ModelViewProjection = (mat4) { 1.95319, 0.972599, 1.90127, 1.87109, -2.13277, 0.890709, 1.74119, 1.7...
 - ModelView = (mat4) { 1.82352, 0.681021, -1.87109, 0, -1.99117, 0.623681, -1.71355, 0, 0, 2.53717, 0.9...
 - lightPos = (vec3) { 0.592694, 0.734606, -0.330254 } Location(10)
 - tex = (samplerCube) { 0 } Location(11)
- Active Attributes 2 Attributes
- Attribute Bindings 1 Bindings
 - GL_LINK_STATUS = GL_TRUE
 - GL_PROGRAM_SEPARABLE = GL_FALSE
- Frame Statistics 30003 draw calls

GL Context

- Error Status No GL Error
- Viewport (0, 0, 2048, 1536) - (0, 1)
- Active Texture Unit Texture Unit 0
- Stencil Off
- Blending Off
- Depth Less - (Write On) - Clear(1.000000)
- Culling Back - CCW
- Framebuffer Write - RGBA, Clear (0.5, 0.5, 0.5, 1)
- Polygon Offset Off
- Multisampling Default Coverage
- Scissor Off
- Misc
- Current Vertex

OpenGL ES Frame Debugger

GL issues navigator

The screenshot displays the OpenGL ES Frame Debugger interface, which is used for debugging GPU issues. The interface is divided into several panels:

- Issues Navigator (Left):** A list of 16 issues categorized by file and type. The issues include:
 - Logical Buffer Load: Your app caused a framebuffer load operation by the GPU. A typical cause is failing to clear at the beginning.
 - Redundant Call: glEnable(GL_DEPTH_TEST) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(0, 0.000000f, 0.000000f, 0.0100000f, 0.9900000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.000000f, 0.0200000f, 0.2000000f, 175.0000000f) set a piece of GL state to its current value.
 - Mipmapping Usage: Texture #2 "/var/mobile/Applications/6F6A8A70-904F-4369-8DD2-3E5B78D04232/
 - Redundant Call: glUniformMatrix4fv(2, 1, 0u, {1.0711110f, 0.0000000f, 0.0000000f, 0.0000000f, 0.0000000f, 1.4281480f,
 - Redundant Call: glUniform4f(0, 0.0400000f, 0.0600000f, 0.1100000f, 0.8500000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.6500000f, 0.3800000f, 0.1800000f, 4.0000000f) set a piece of GL state to its current value.
 - Redundant Call: glBindBuffer(GL_ARRAY_BUFFER, 2u) set a piece of GL state to its current value.
 - Redundant Call: glUniformMatrix4fv(6, 1, 0u, {0.0000000f, 0.0000000f, 0.0000000f, 0.0000000f,
 - Logical Buffer Load: Your app caused a framebuffer load operation by the GPU. A typical cause is failing to clear at the beginning.
 - Redundant Call: glUniform4f(0, 0.1200000f, 0.0000000f, 0.0800000f, 0.6000000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.8500000f, 0.9500000f, 0.9000000f, 0.9000000f) set a piece of GL state to its current value.
 - Dependent Texture Sampling: The fragment shader in Program #12 performed dependent texture reads which are slower than non-
 - Redundant Call
- 3D Scene (Center):** A 3D rendering of a green sphere in a space environment with a starry background.
- GL Context (Bottom):** A list of current GL state settings, including:
 - Error Status: No GL Error
 - Viewport: (0, 0, 2048, 1536) - (0, 1
 - Active Texture Unit: Texture Unit 0
 - Stencil: Off
 - Blending: Off
 - Depth: Less - (Write On) - Clear(1.000
 - Culling: Back - CCW
 - Framebuffer: Write - RGBA, Clear (0.3
 - Polygon Offset: Off
 - Multisampling: Default Coverage
 - Scissor: Off
 - Misc
 - Current Vertex
- Issues List (Middle):** A detailed list of 16 issues, including:
 - Logical Buffer Load
 - Redundant Call: glEnable(GL_DEPTH_TEST) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(0, 0.0000000f, 0.0000000f, 0.0100000f, 0.9900000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.0000000f, 0.0200000f, 0.2000000f, 175.0000000f) set a piece of GL state to its current value.
 - Mipmapping Usage: Texture #2 "/var/mobile/Applications/6F6A8A70-904F-4369-8DD2-3E5B78D04232/
 - Redundant Call: glUniformMatrix4fv(2, 1, 0u, {1.0711110f, 0.0000000f, 0.0000000f, 0.0000000f, 0.0000000f, 1.4281480f,
 - Redundant Call: glUniform4f(0, 0.0400000f, 0.0600000f, 0.1100000f, 0.8500000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.6500000f, 0.3800000f, 0.1800000f, 4.0000000f) set a piece of GL state to its current value.
 - Redundant Call: glBindBuffer(GL_ARRAY_BUFFER, 2u) set a piece of GL state to its current value.
 - Redundant Call: glUniformMatrix4fv(6, 1, 0u, {0.0000000f, 0.0000000f, 0.0000000f, 0.0000000f,
 - Logical Buffer Load: Your app caused a framebuffer load operation by the GPU. A typical cause is failing to clear at the beginning.
 - Redundant Call: glUniform4f(0, 0.1200000f, 0.0000000f, 0.0800000f, 0.6000000f) set a piece of GL state to its current value.
 - Redundant Call: glUniform4f(1, 0.8500000f, 0.9500000f, 0.9000000f, 0.9000000f) set a piece of GL state to its current value.
 - Dependent Texture Sampling: The fragment shader in Program #12 performed dependent texture reads which are slower than non-
 - Redundant Call
- Shader Editor (Right):** A code editor showing the fragment shader for Program #12. The code includes uniforms for color and texture, and a main function that calculates the final color. A red error message is visible: "Use of undeclared identifier 'foo'".

OpenGL ES Frame Debugger

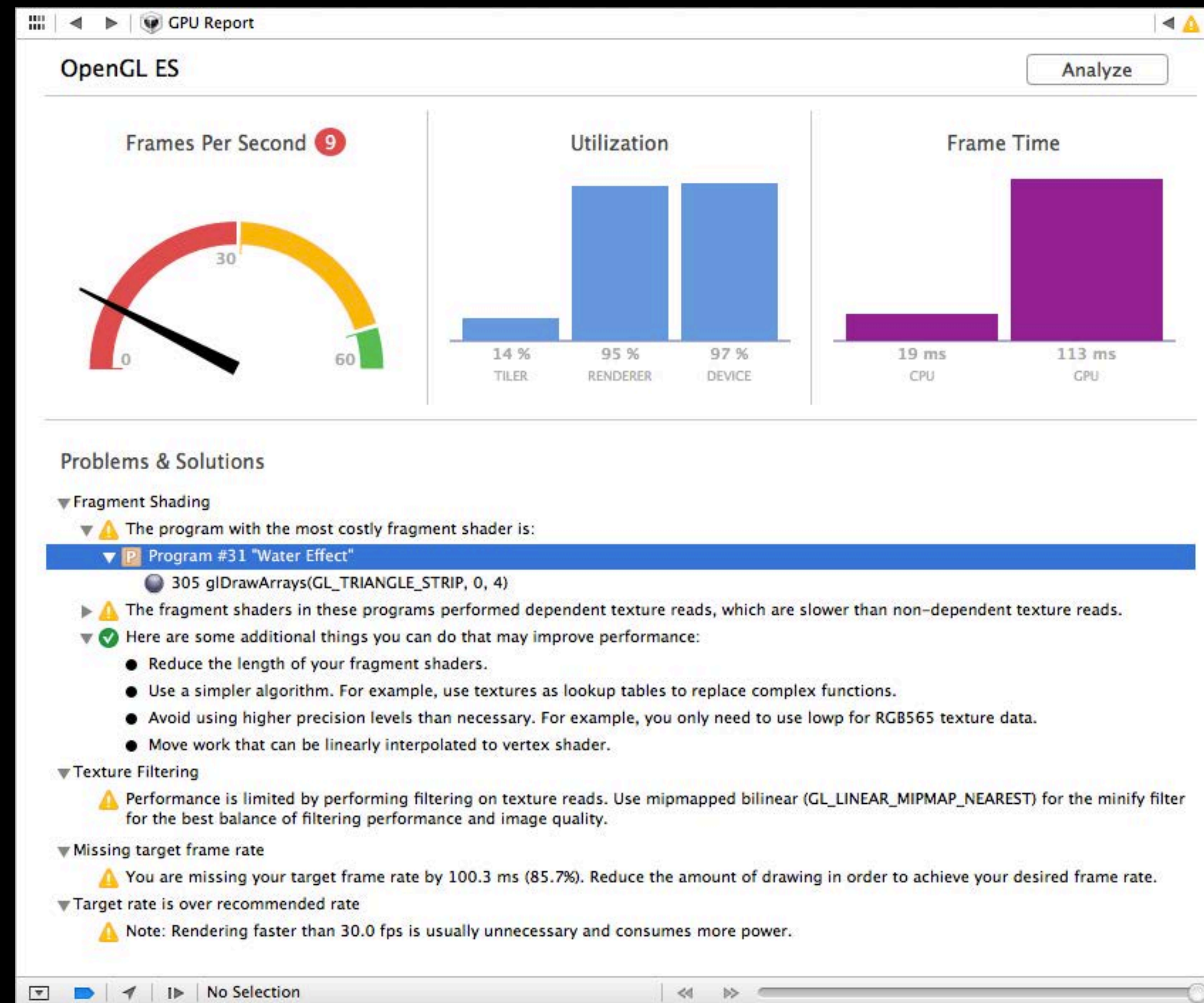
GL issues navigator

The screenshot displays the OpenGL ES Frame Debugger interface. The top window shows the application running on a device, with the current frame being 60035. The main interface is divided into several panels:

- Issues Navigator (Left):** Lists 16 issues for the 'InstancingDemo' application. Issues include 'Logical Buffer Load', 'Redundant Call' (multiple instances of glUniform4f and glBindBuffer), 'Mipmapping Usage', and 'Dependent Texture Sampling'. A 'Color' dialog box is open over the scene view, showing a range from 0 to 1.
- Scene View (Center):** Displays a 3D scene with a green sphere and a grey textured sphere. A 'Color' dialog box is open over the scene view, showing a range from 0 to 1.
- Shader Editor (Right):** Shows the source code for the 'Fragment Shader'. A red error message is visible: 'Use of undeclared identifier 'foo''. The code includes comments about capturing and saving shaders, and defines uniforms and attributes for a cube texture.
- State Inspector (Bottom):** Shows the current state of the GL context. It includes an 'Error Status' section (No GL Error), a 'Viewports' section (Viewport (0, 0, 2048, 1536) - (0, 1)), an 'Active Texture Unit' section (Texture Unit 0), and a 'Current Vertex' section. The 'Program #12' section shows uniforms, active attributes, and attribute bindings.

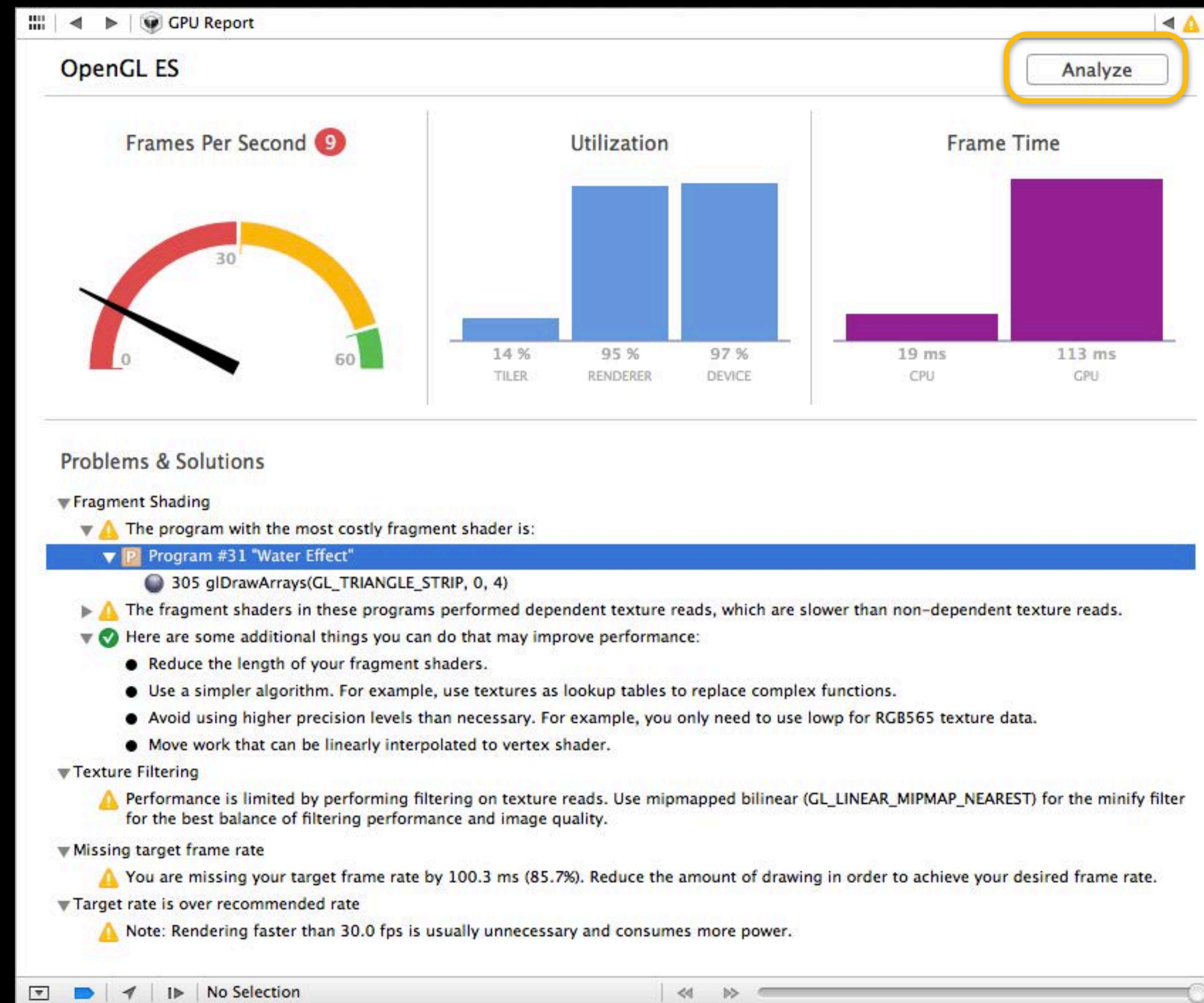
OpenGL ES Frame Debugger

OpenGL ES performance analysis



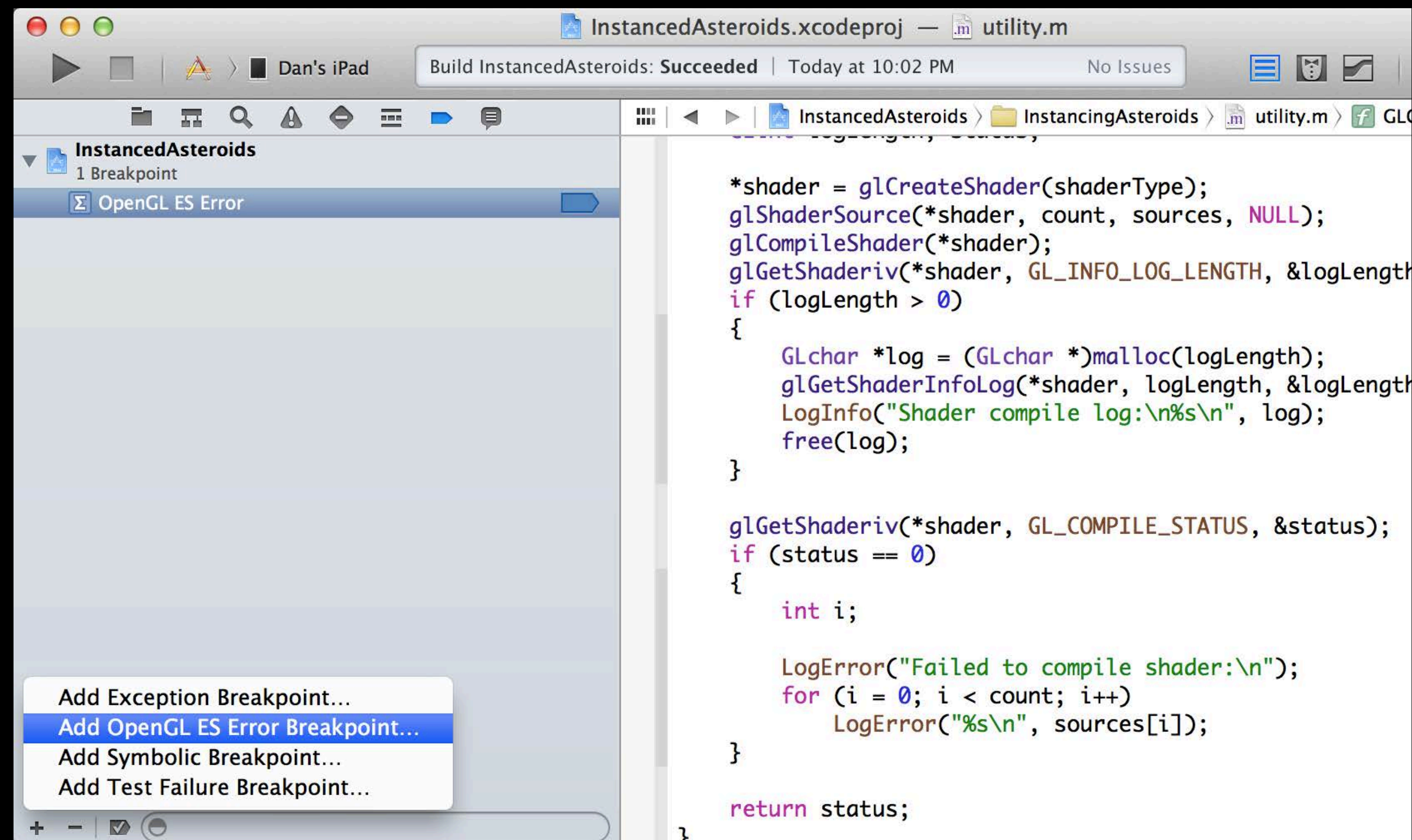
OpenGL ES Frame Debugger

OpenGL ES performance analysis



Xcode Breakpoints

Break on OpenGL ES error



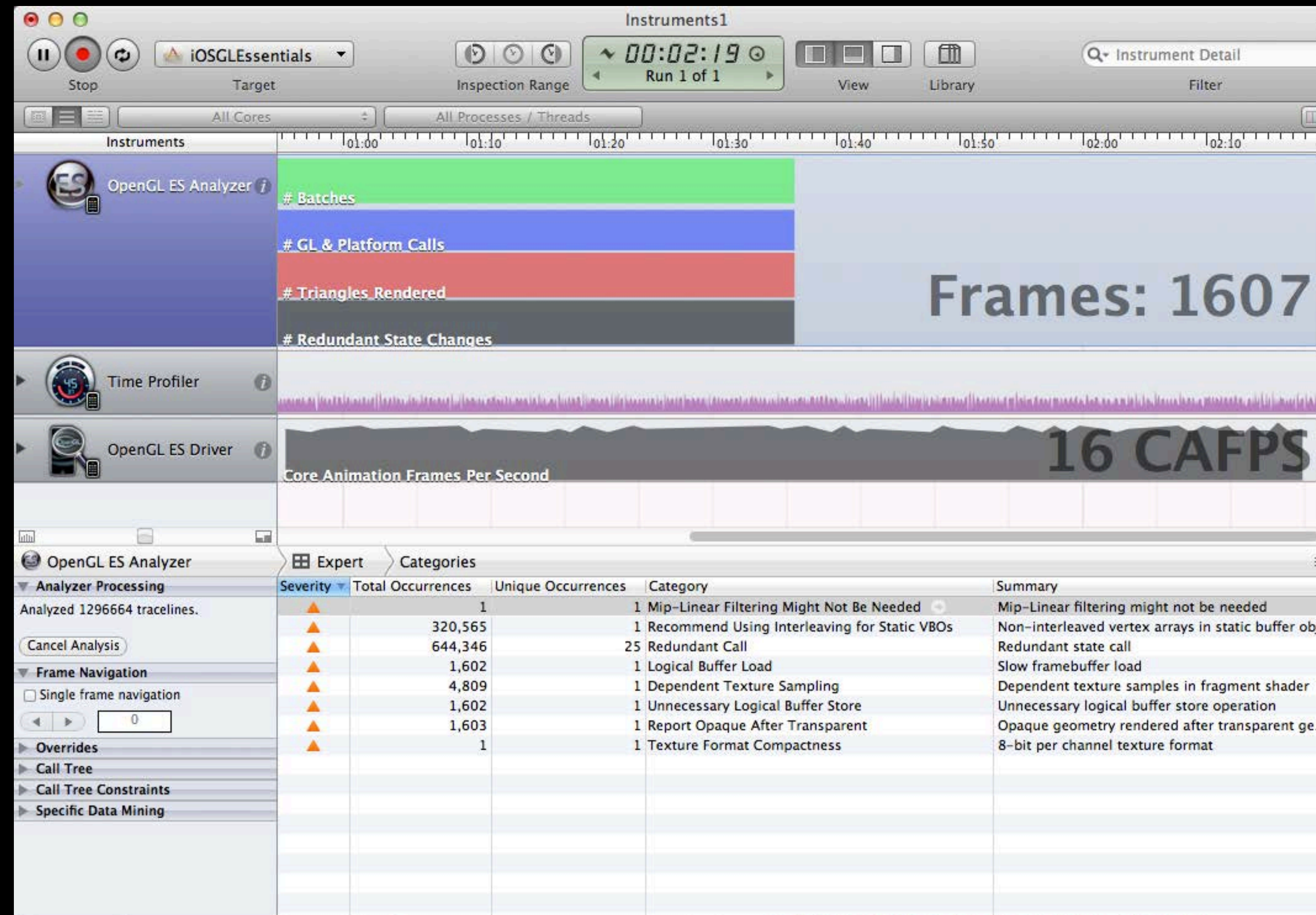
OpenGL ES Analyzer Instrument

The screenshot displays the OpenGL ES Analyzer instrument interface. At the top, it shows the target 'iOSGLEssentials' and a timer at 00:01:26. The main area features a timeline with four instrumented metrics: # Batches (green), # GL & Platform Calls (blue), # Triangles Rendered (red), and # Redundant State Changes (grey). A large blue box indicates 'Frames: 1013' and another grey box shows '16 CAFPS'. A context menu is open over the 'OpenGL ES Driver' instrument, listing options: Expert, Frame Statistics, Trace, Call Trees, and API Statistics (which is selected). Below the timeline, the 'OpenGL ES Analyzer' panel shows 'Analyzed 820866 tracelines' and a 'Cancel Analysis' button. The bottom section displays a table of API statistics.

Function	Call Count	Total Time (µs)	Average Time (µs)
EAGLContext_presentRenderBuffer	1,014	49,859,565	49,171
glDrawElements	202,961	1,168,700	5
glUniformMatrix4fv	202,961	239,890	1
glClear	1,016	63,107	62
glEnable	202,963	30,455	0.0
glUseProgram	1,017	15,460	15
EAGLContext_initWithAPI_properties	1	14,604	14,604
glValidateProgram	1	9,725	9,725
glDisable	202,961	8,176	0.0
glCompileShader	2	6,515	3,257
glBindFramebuffer	1,016	6,432	6
glGenerateMipmap	1	5,824	5,824
glBindVertexArray	1,017	4,408	4
EAGLContext_renderbufferStorage_fromDrawable	2	3,850	1,925
glTexImage2D	1	3,663	3,663
glBindTexture	1,017	2,552	2

OpenGL ES Analyzer Instrument

OpenGL ES expert



Maximizing GPU Performance

Understanding the GPU architecture

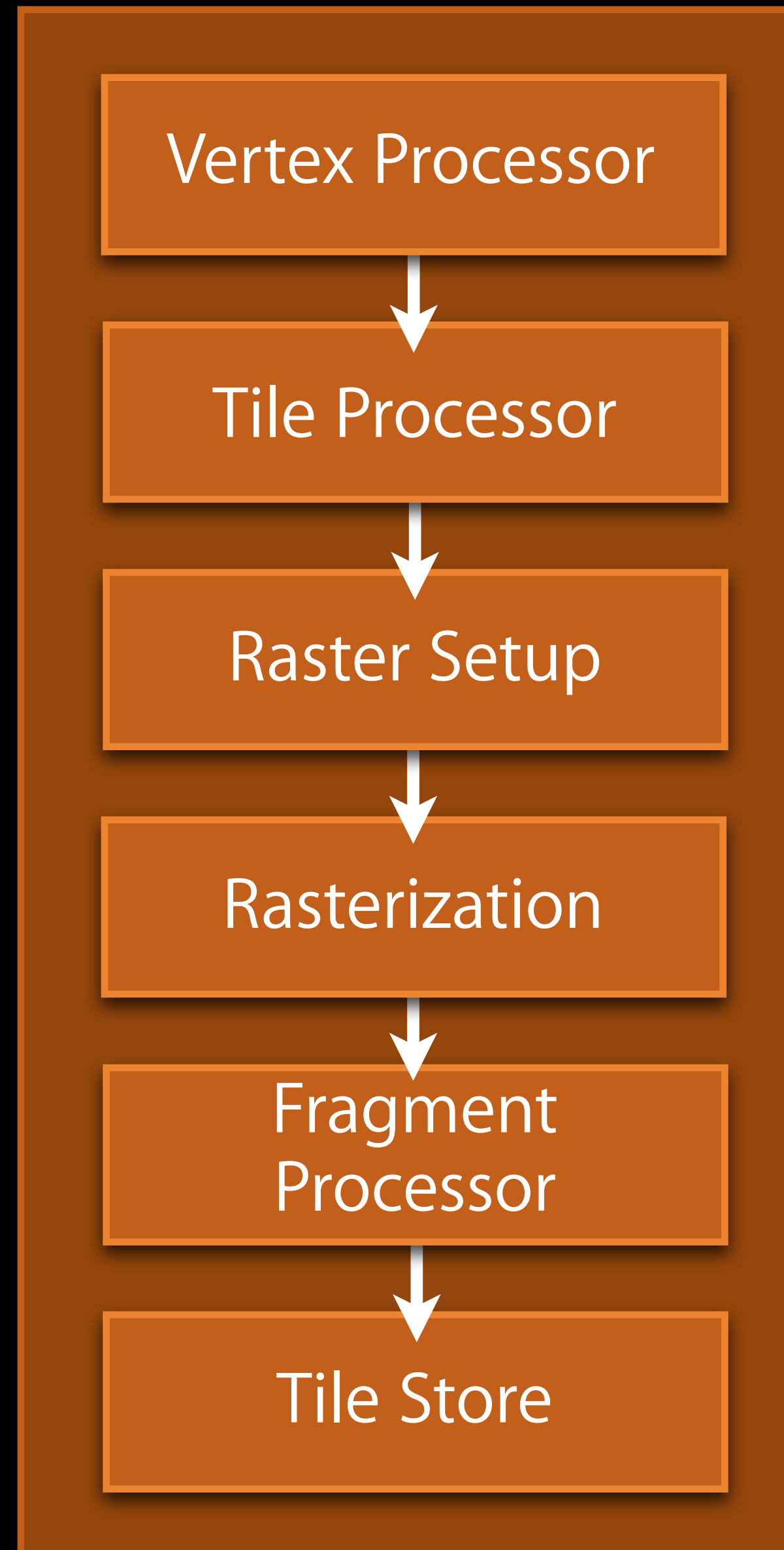
Tile-Based Deferred Renderer

High performance, low power

- TBDR Pipeline different than traditional streaming GPUs
- Optimizations to reduce processing load
 - Increases performance
 - Saves power
- Depends heavily on caches
 - Large transfers to unified memory costly
- Certain operations prevent optimizations or cause cache misses
 - Operations avoidable

The Tile-Based Deferred Rendering Pipeline

GPU



The Tile-Based Deferred Rendering Pipeline

GPU



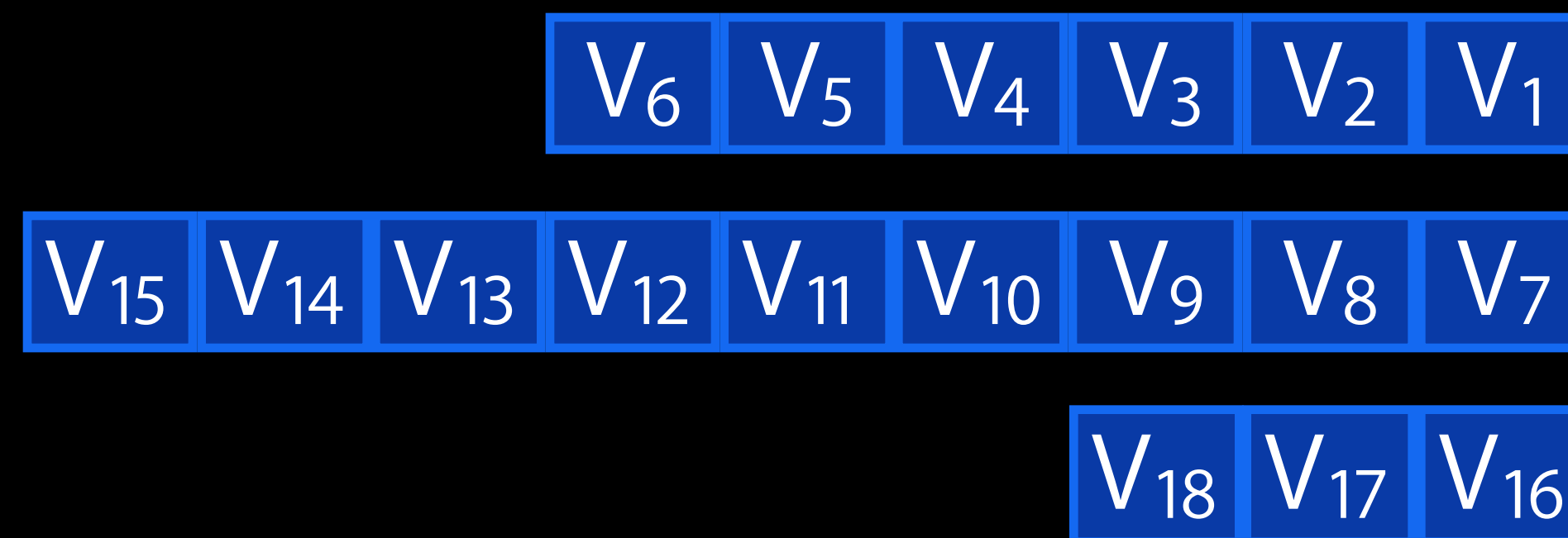
Vertex Processor

Vertex Shading

- Draw call submits vertices to GPU's vertex processor

Unified Memory

Vertex Arrays



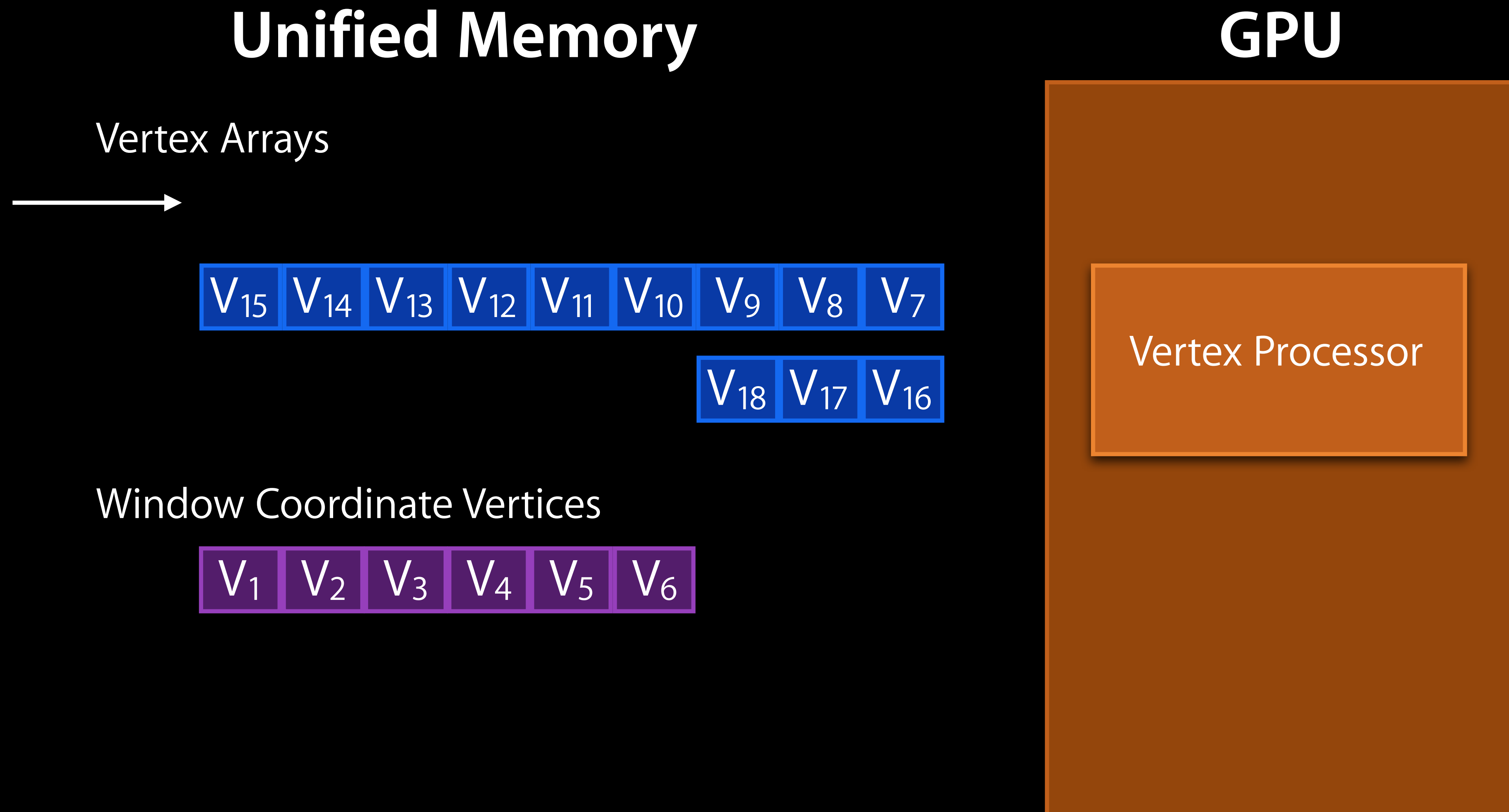
GPU

The GPU is represented by a large orange rectangle. Inside this rectangle, centered, is a smaller orange rectangle labeled "Vertex Processor".

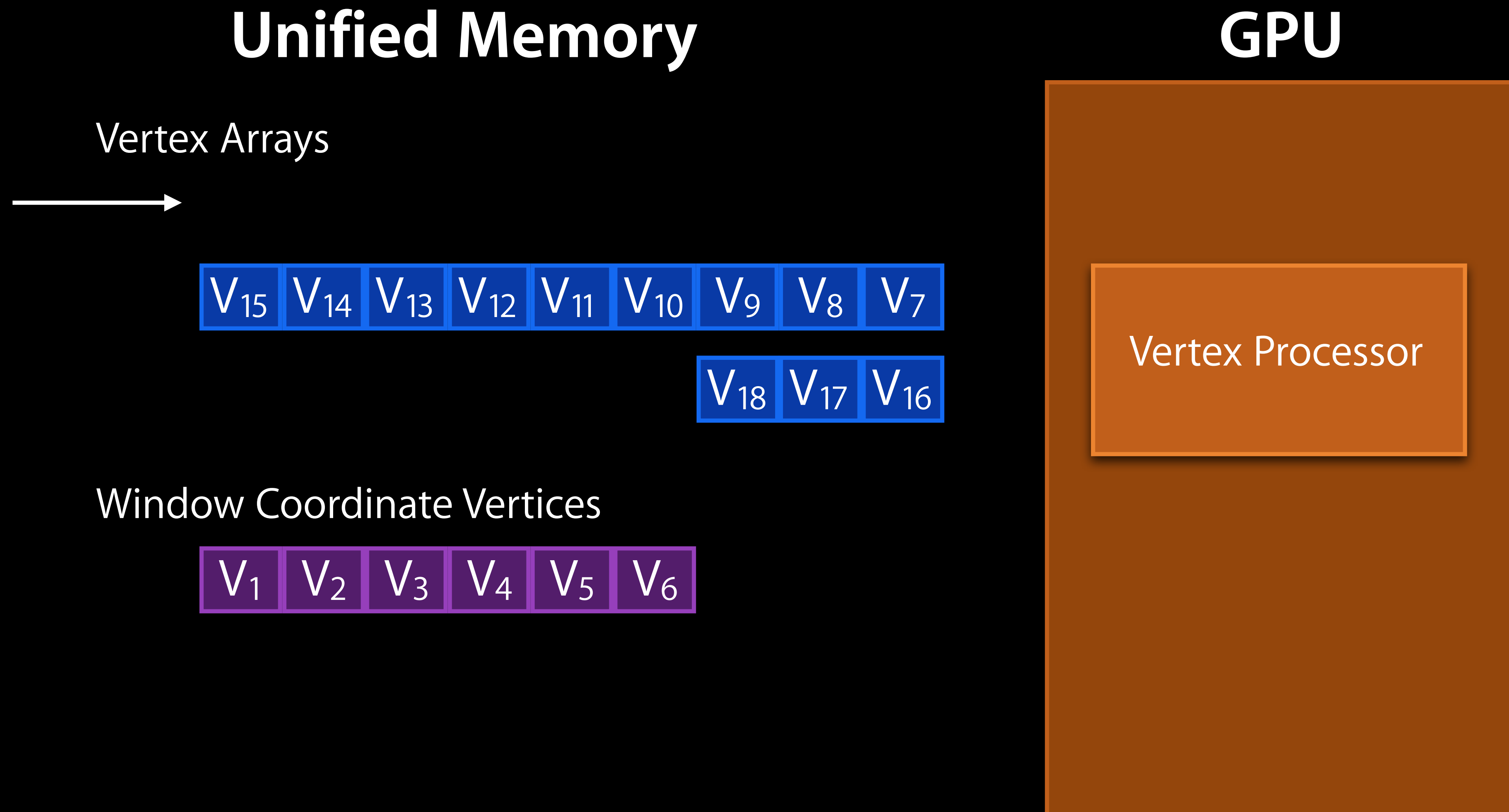
Vertex Processor

Vertex Shading

- Draw call submits vertices to GPU's vertex processor



- Vertices shaded and transformed to window coords
- Shaded, transform vertices, stored out to unified memory



- Vertices shaded and transformed to window coords
- Shaded, transform vertices, stored out to unified memory

Unified Memory

Vertex Arrays



Window Coordinate Vertices



GPU

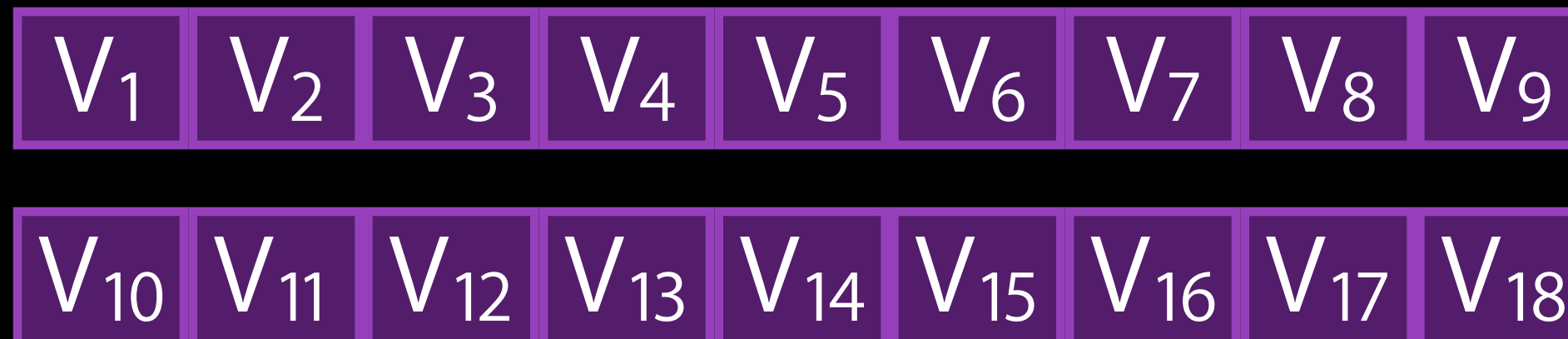
Vertex Processor

- A frames' worth of vertices stored
- Rasterization deferred until `presentRenderbuffer` or `renderbuffer` changed

Unified Memory

GPU

Window Coordinate Vertices



Vertex Processor

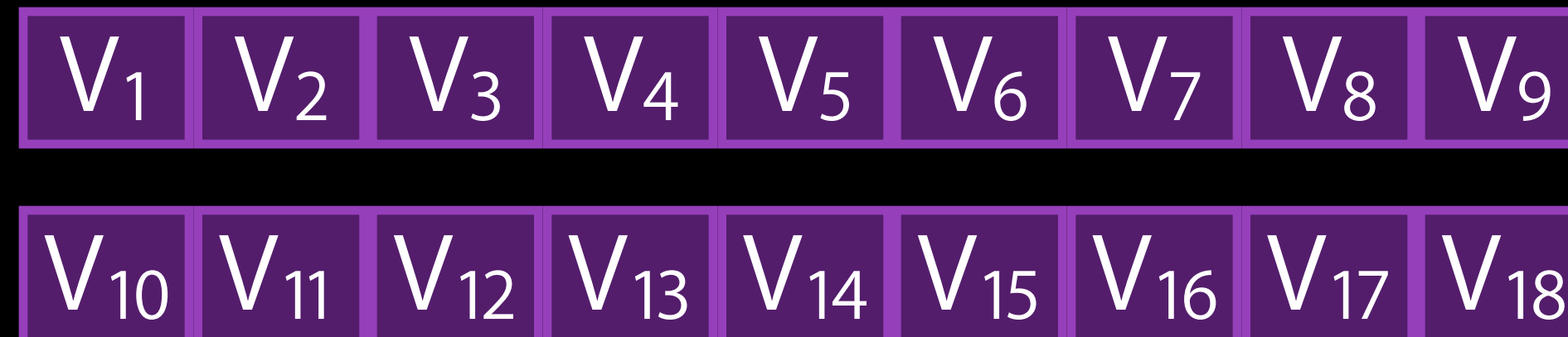
Tiling Processor

Unified Memory

GPU

Vertex Processor

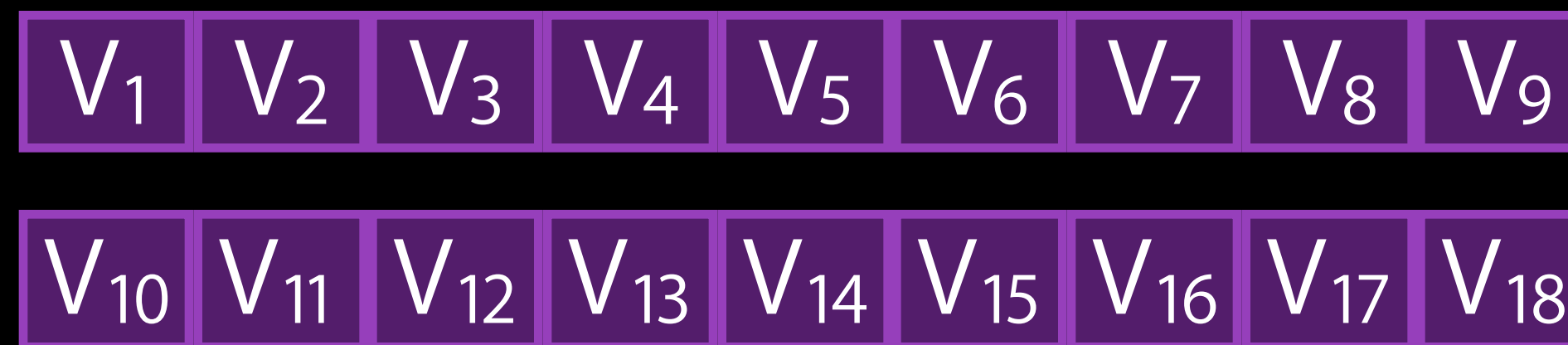
Window Coordinate Vertices



Tiling Processor

Unified Memory

Window Coordinate Vertices



GPU

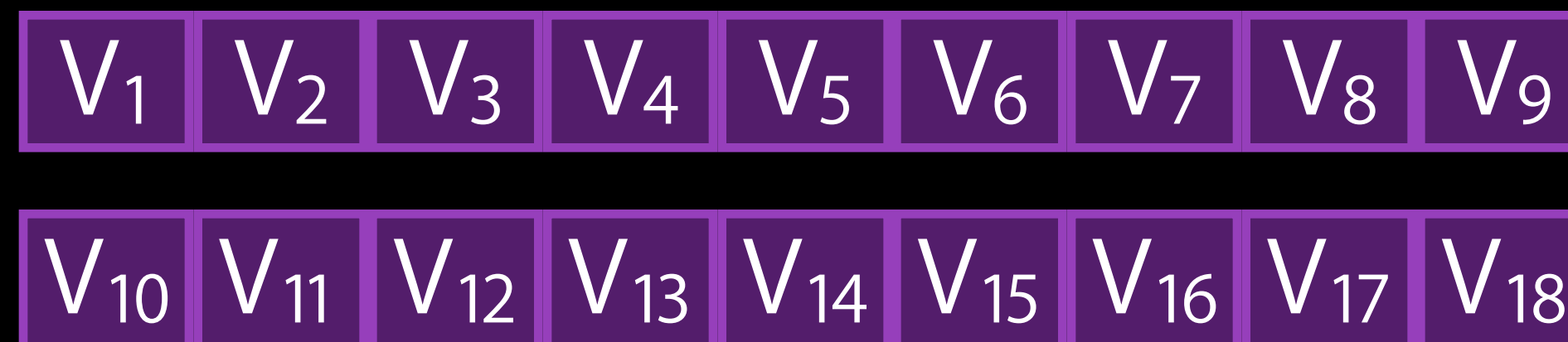
A diagram of a GPU represented as a large orange rectangle. Inside this rectangle, centered, is a smaller orange rectangle labeled "Tiling Processor".

Tiling Processor

- Render buffers split into tiles
- Allows rasterization and fragment shading on embedded GPU memory

Unified Memory

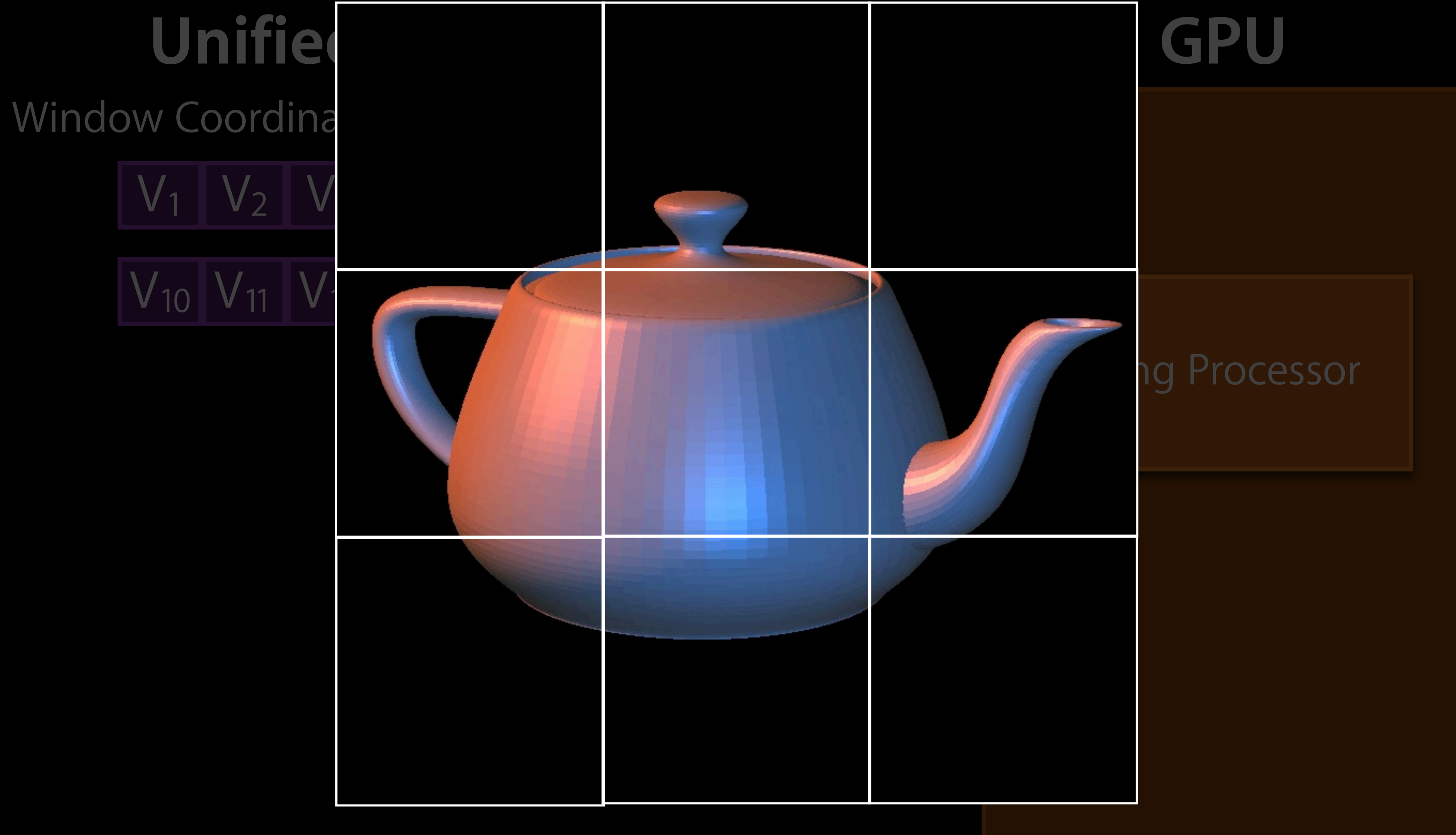
Window Coordinate Vertices



GPU

A diagram of a GPU architecture. A large orange rectangle represents the GPU. Inside this rectangle, a smaller orange rectangle is labeled "Tiling Processor".

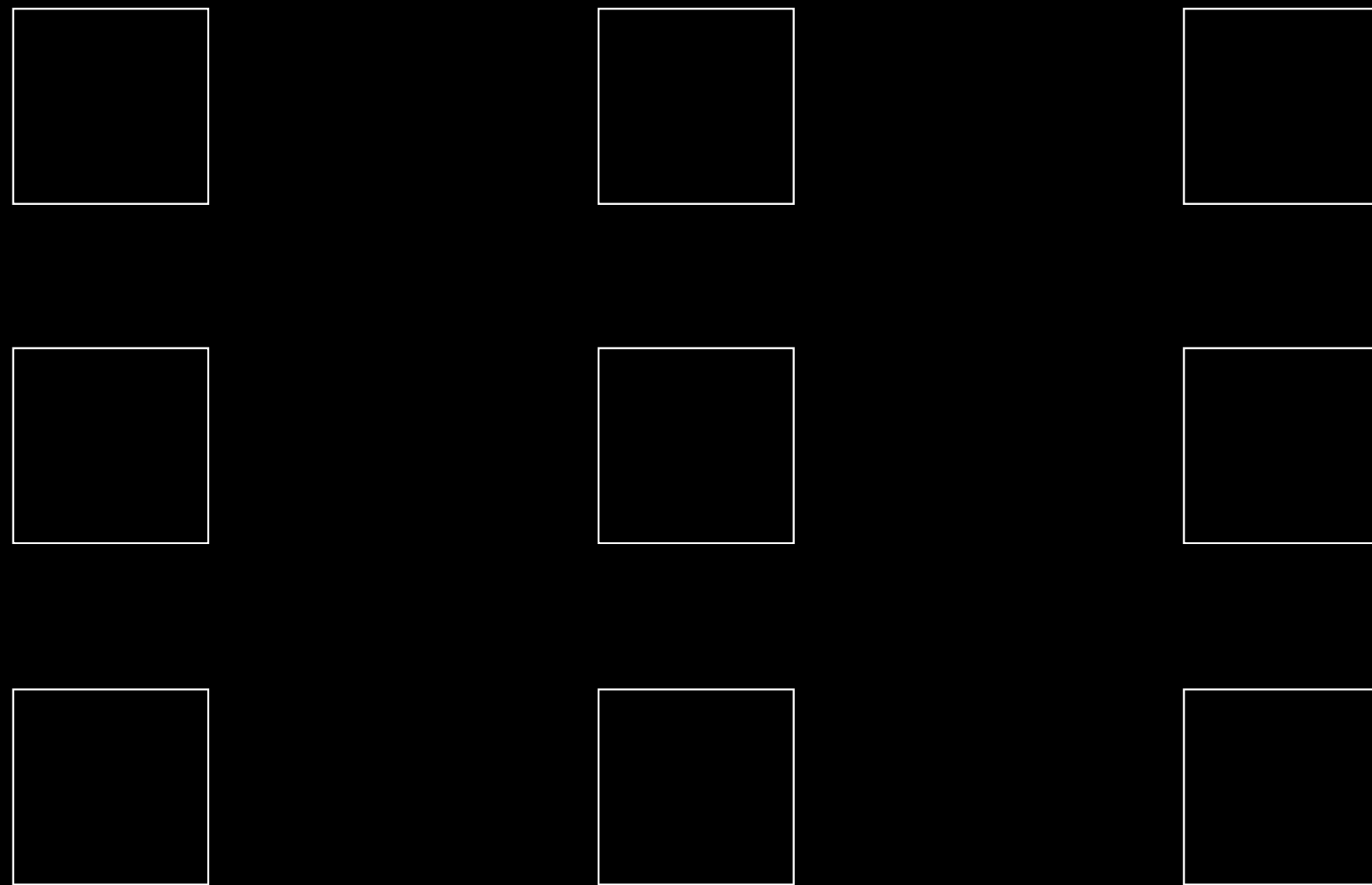
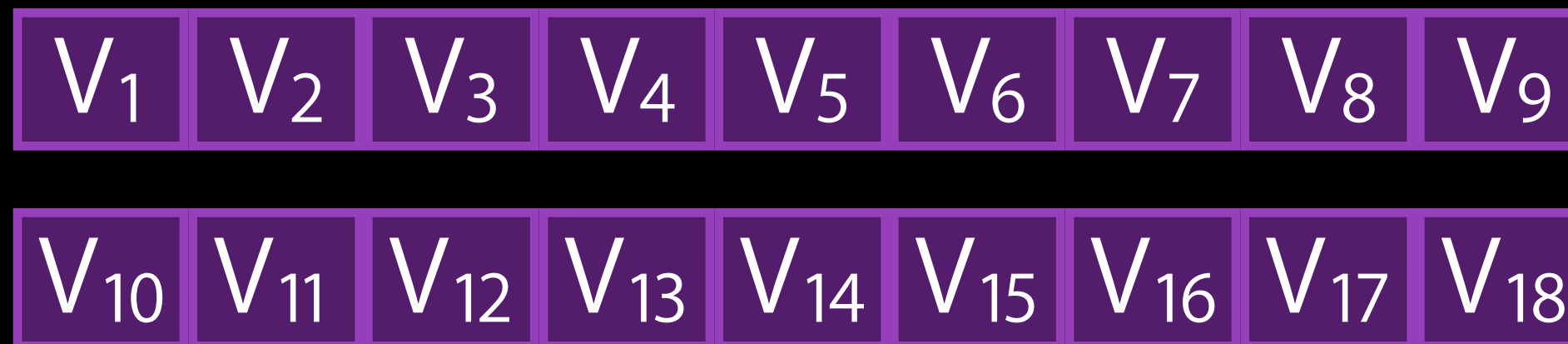
- Render buffers split into tiles
- Allows rasterization and fragment shading on embedded GPU memory



- Render buffers split into tiles
- Allows rasterization and fragment shading on embedded GPU memory

Unified Memory

Window Coordinate Vertices



GPU



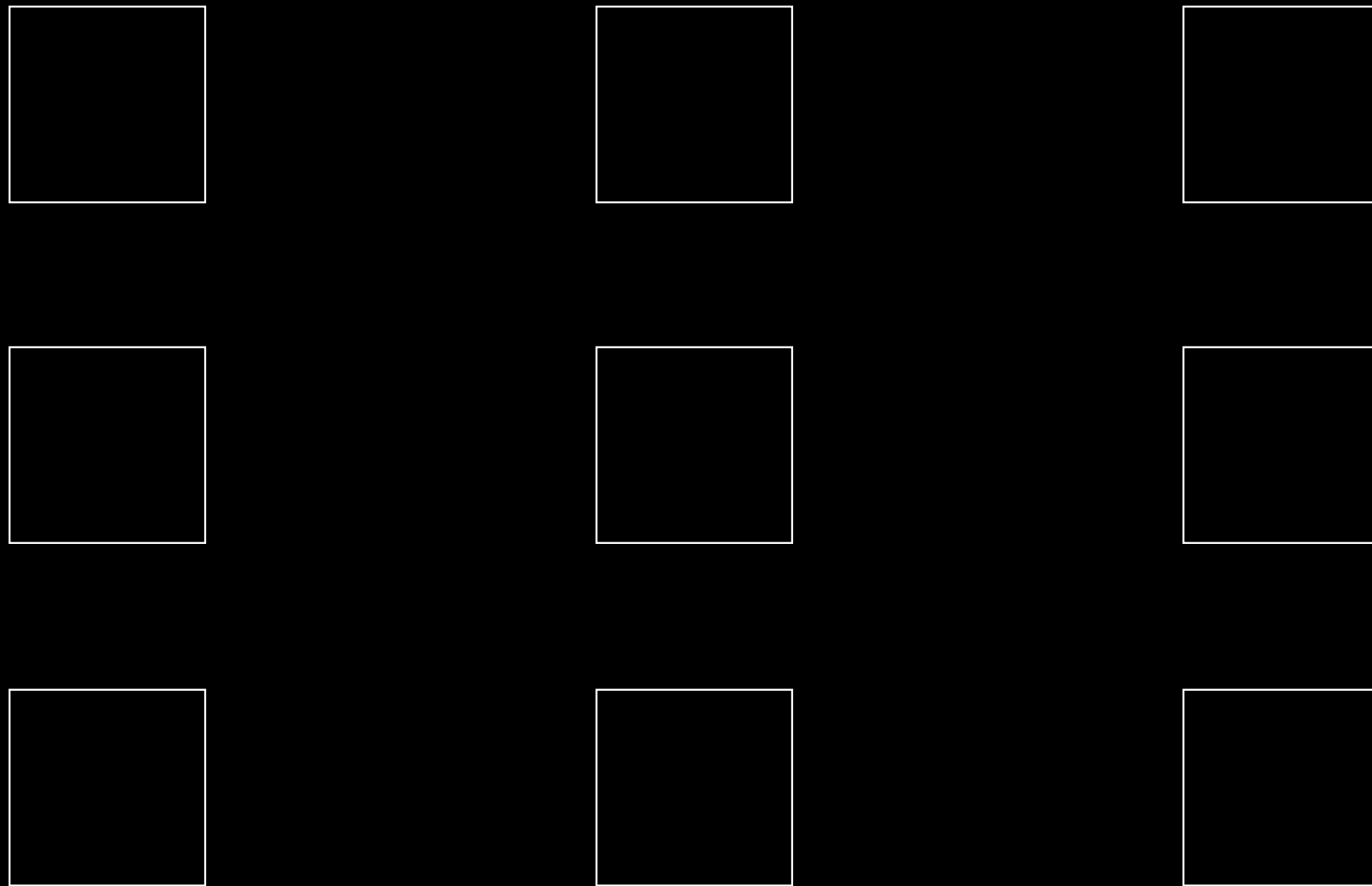
- Vertices processed in groups of triangles
- Tiling processor bins triangles into tiles in which they are located

Unified Memory

Window Coordinate Vertices

V₁ V₂ V₃ V₄ V₅ V₆ V₇ V₈ V₉

V₁₀ V₁₁ V₁₂ V₁₃ V₁₄ V₁₅ V₁₆ V₁₇ V₁₈



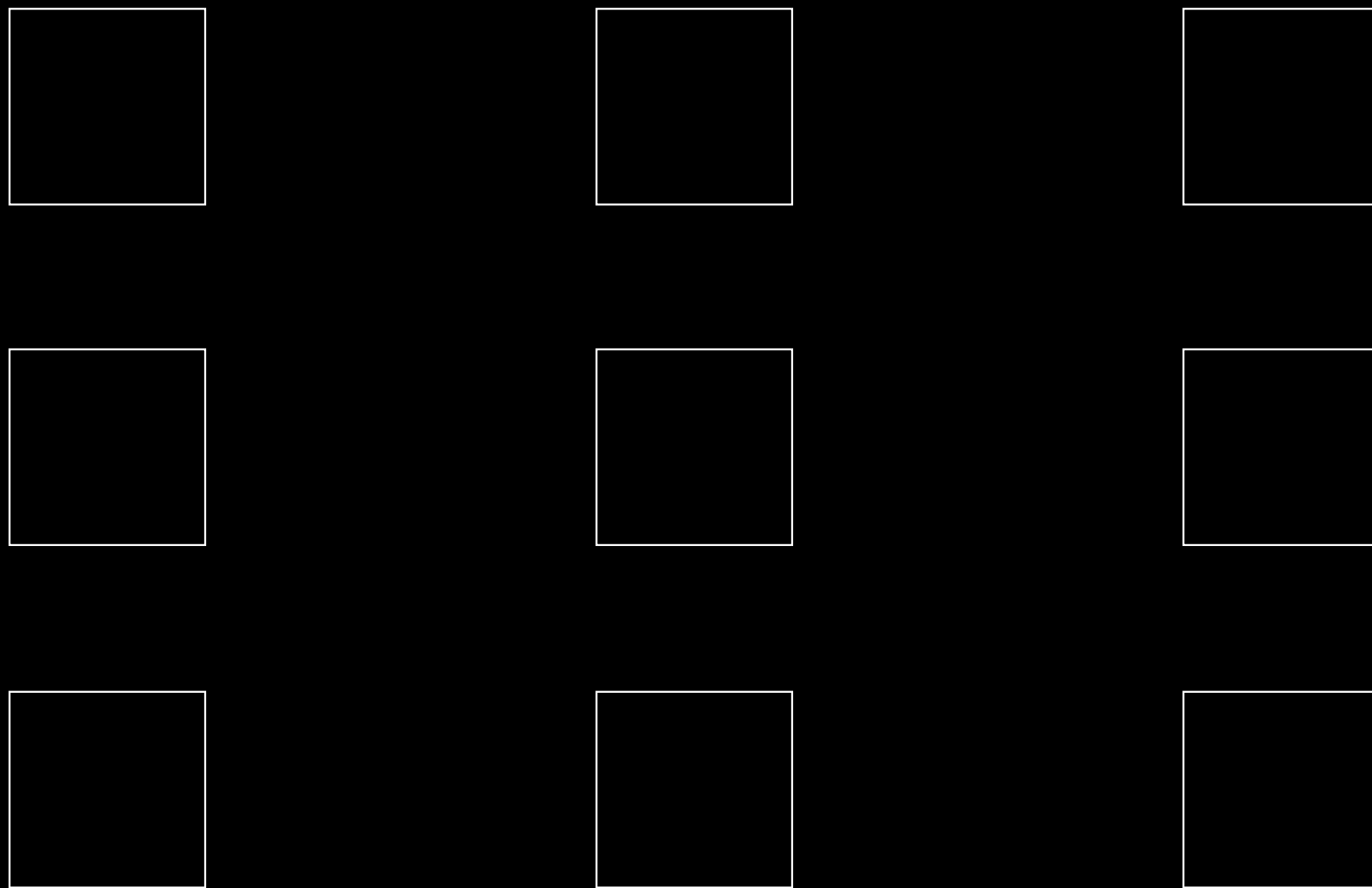
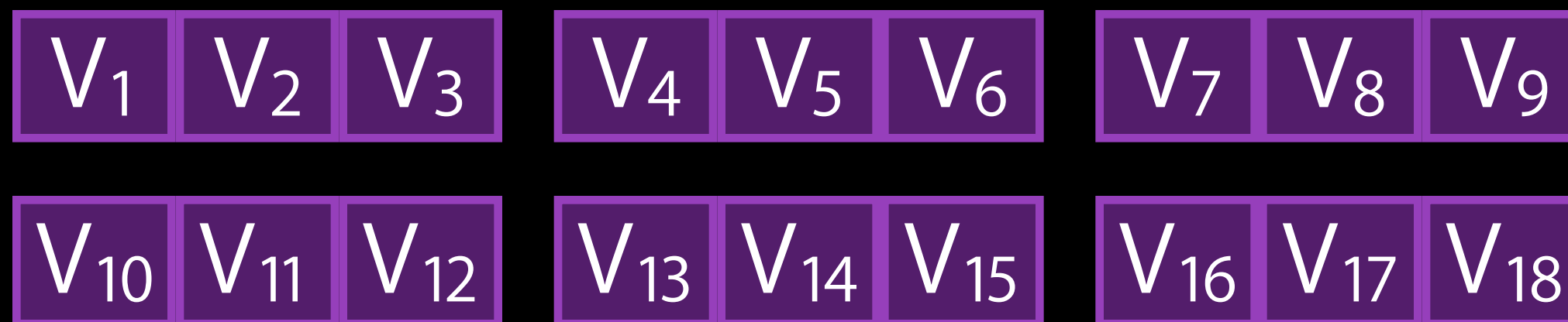
GPU

Tiling Processor

- Vertices processed in groups of triangles
- Tiling processor bins triangles into tiles in which they are located

Unified Memory

Window Coordinate Vertices



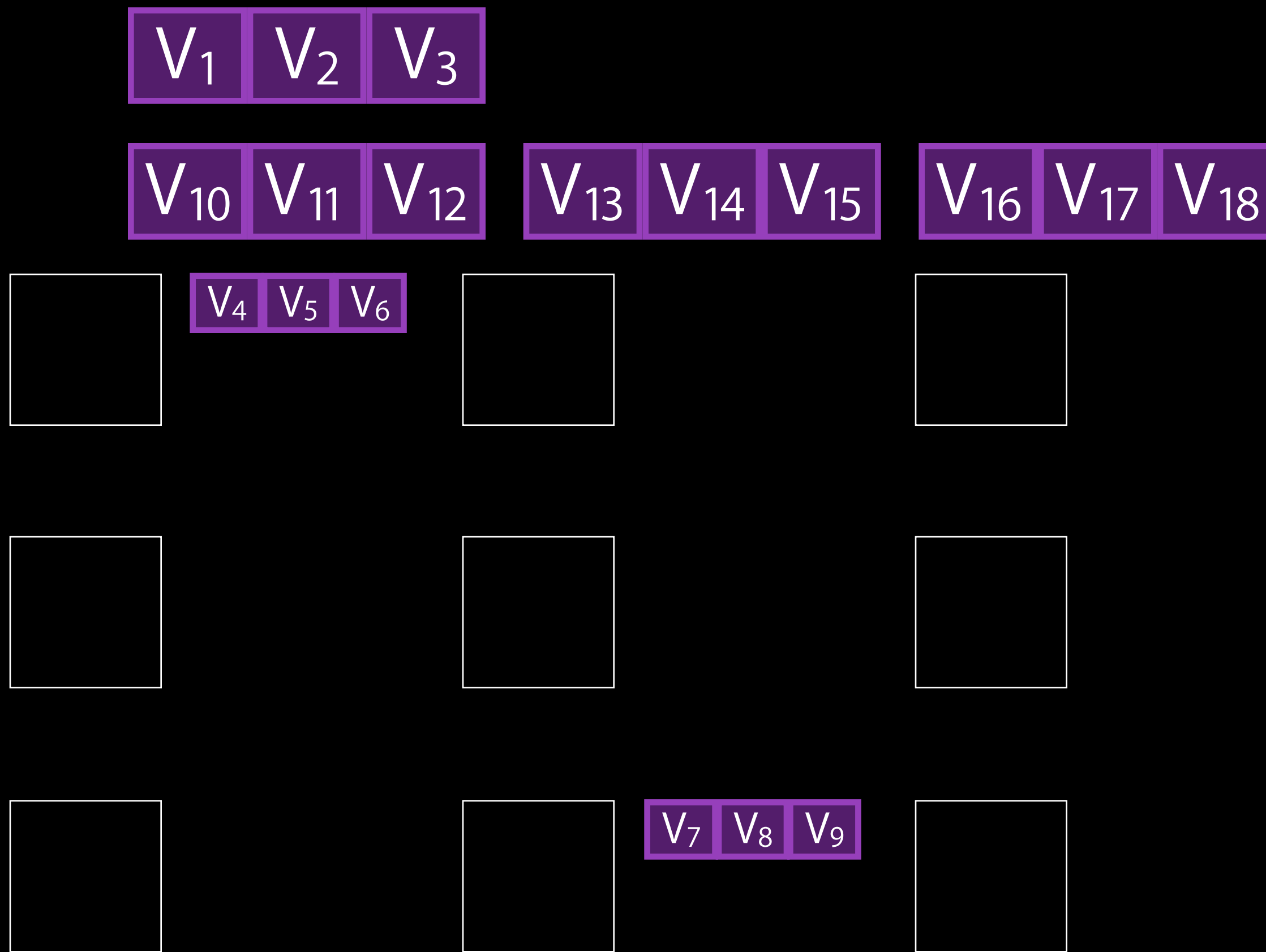
GPU

Tiling Processor

- Vertices processed in groups of triangles
- Tiling processor bins triangles into tiles in which they are located

Unified Memory

Window Coordinate Vertices



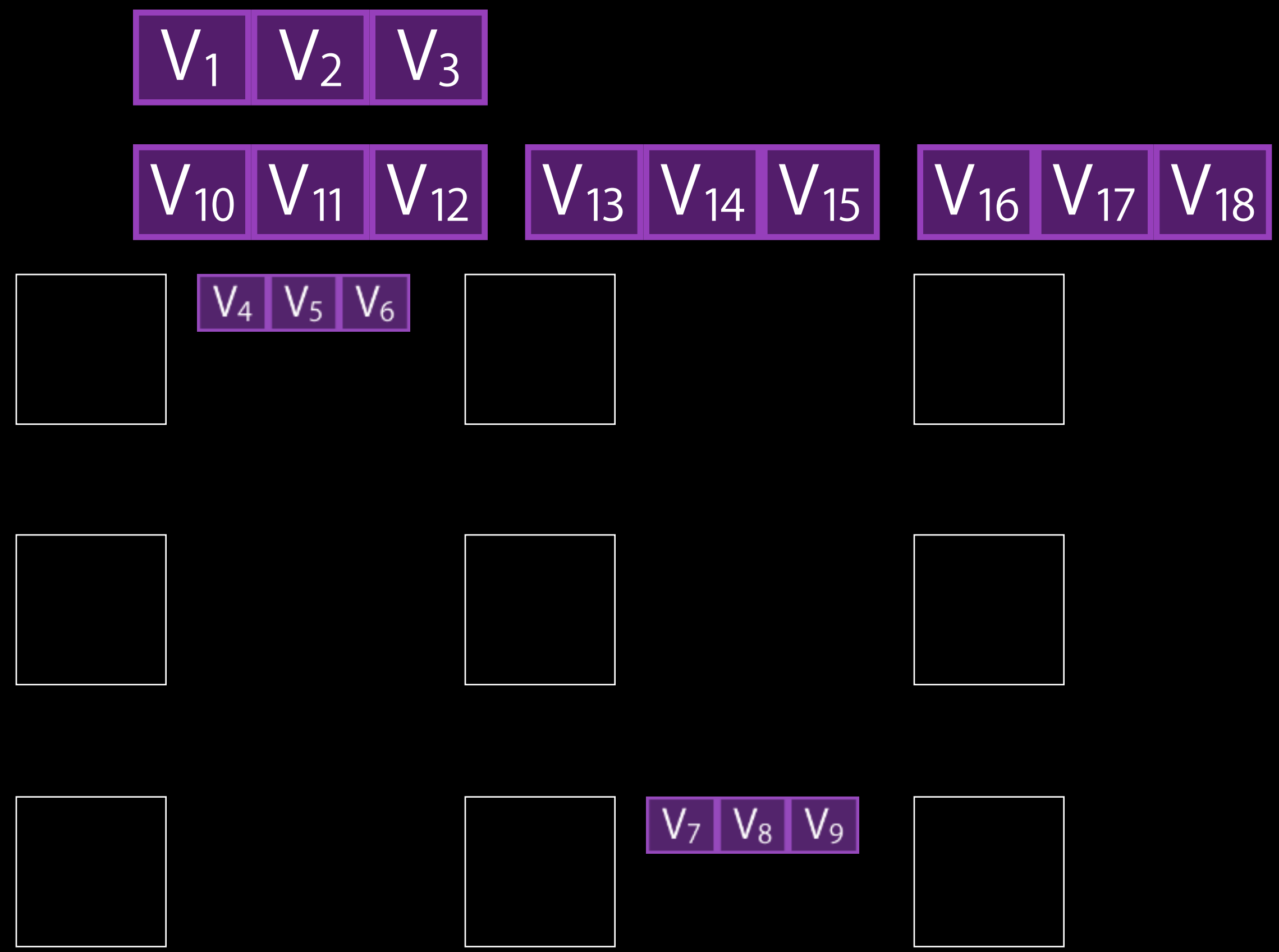
GPU

The GPU is represented by a large orange rectangle. Inside it, a smaller orange rectangle is labeled "Tiling Processor".

- GPU may bin a larger triangle into multiple tiles

Unified Memory

Window Coordinate Vertices

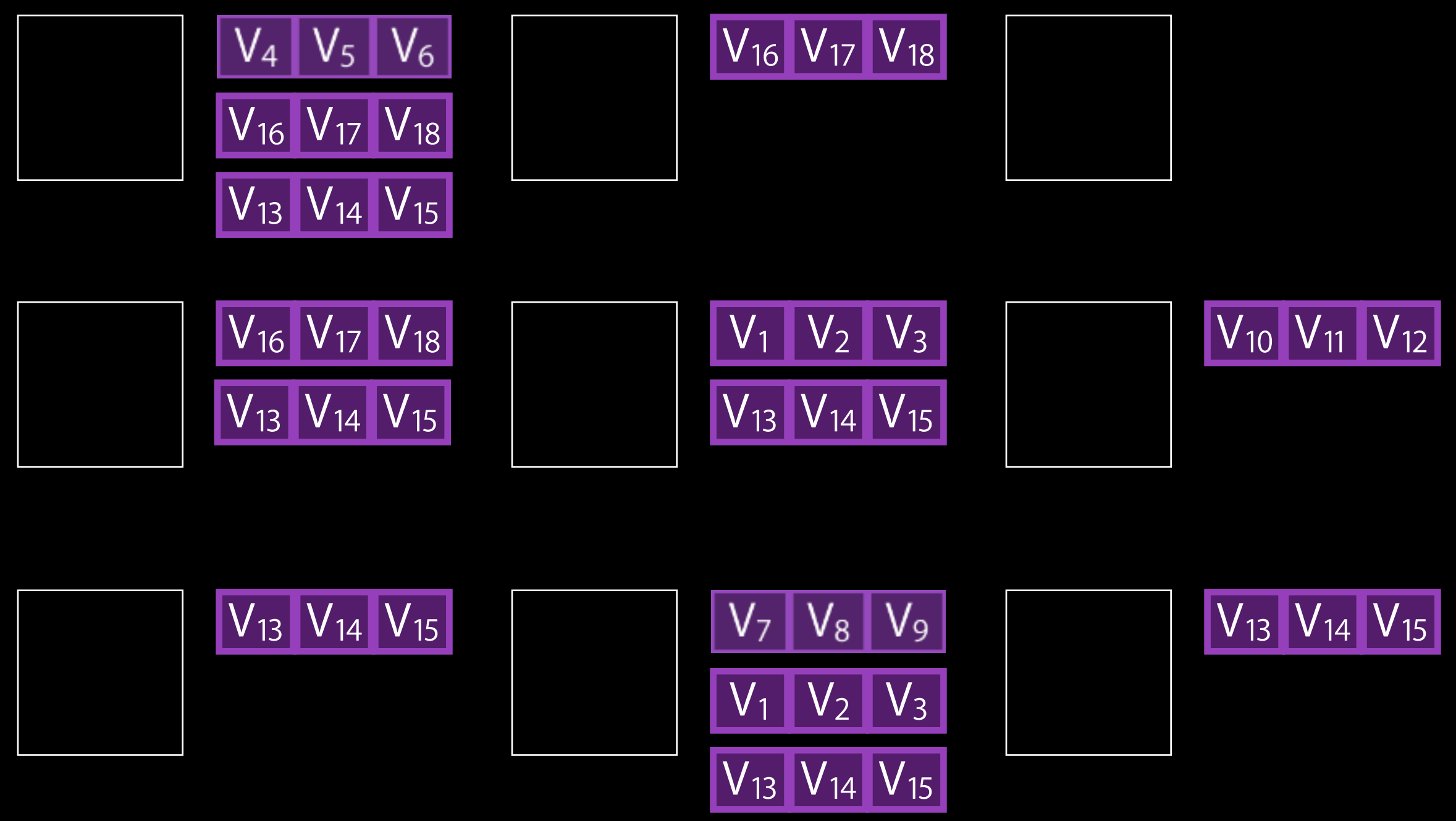


GPU



- GPU may bin a larger triangle into multiple tiles

Unified Memory

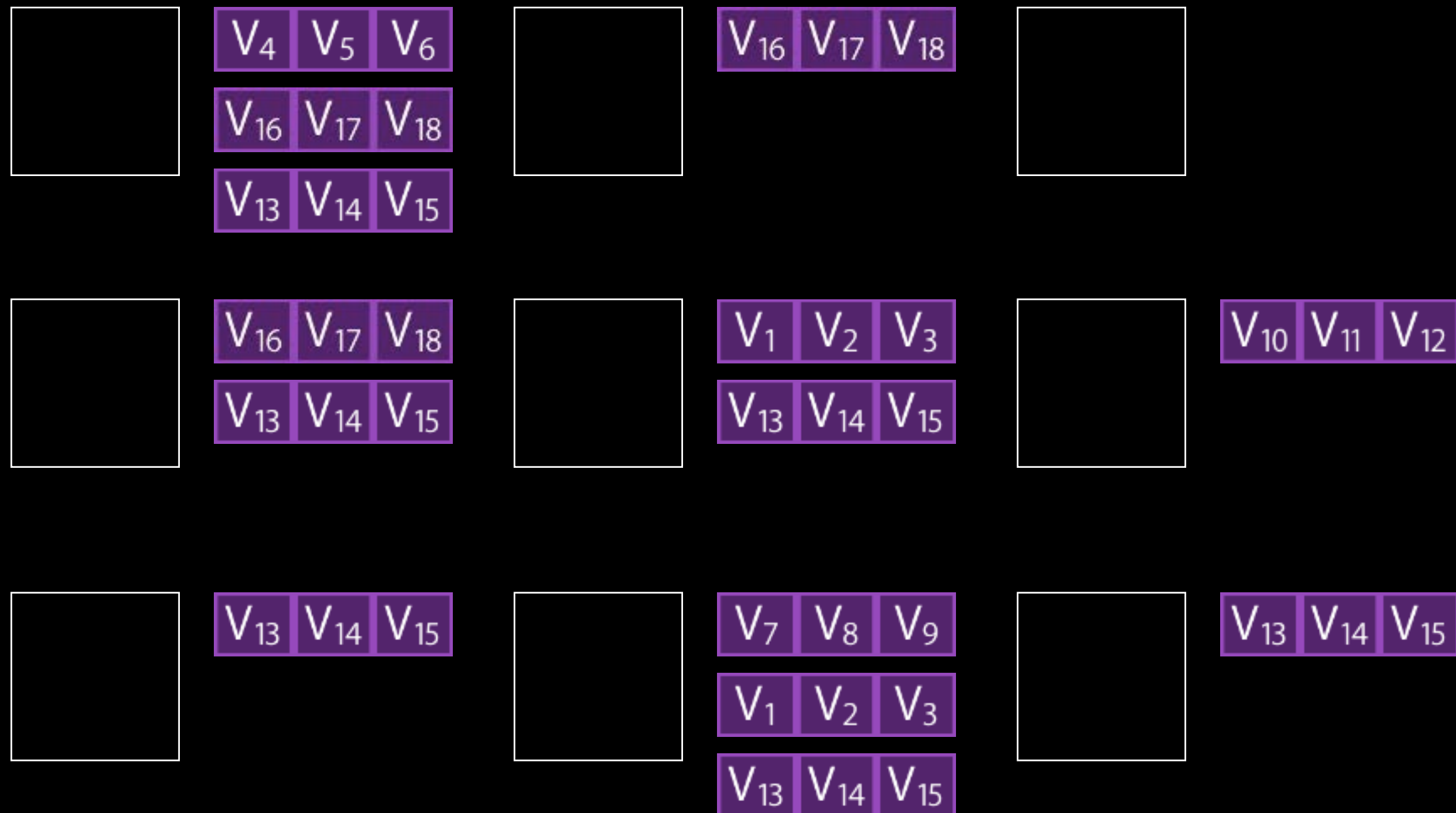


GPU



Raster Setup

Unified Memory

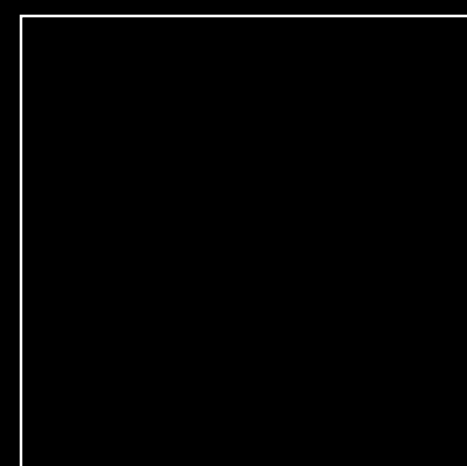
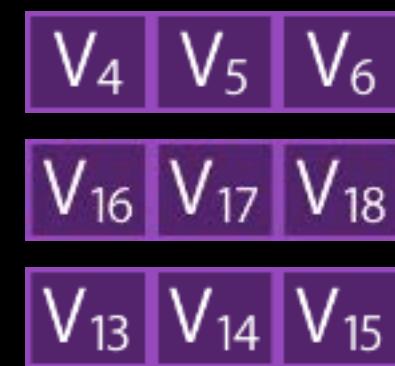


GPU

Tiling Processor

Raster Setup

Unified Memory

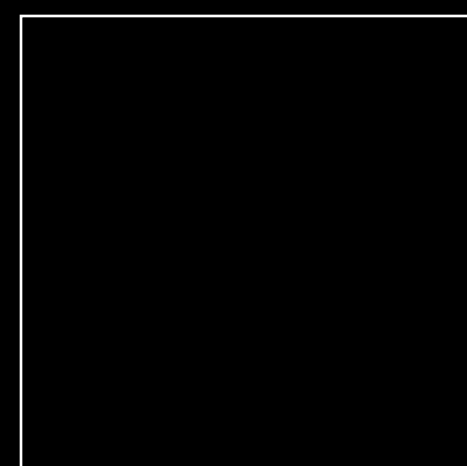
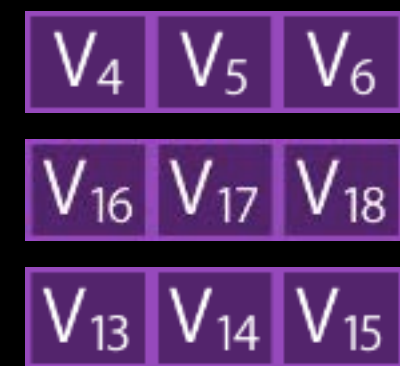


GPU

Raster Setup

Raster Setup

Unified Memory



GPU

Raster Setup

Raster Setup

Unified Memory

GPU

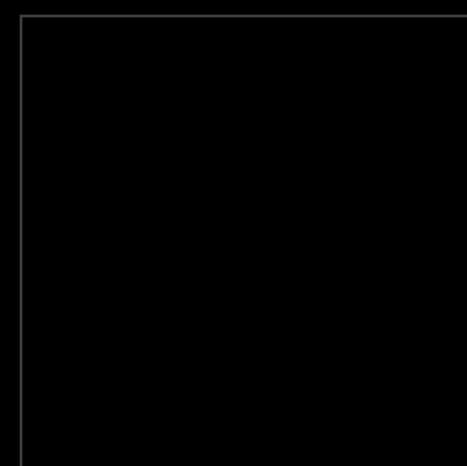
V₄ V₅ V₆



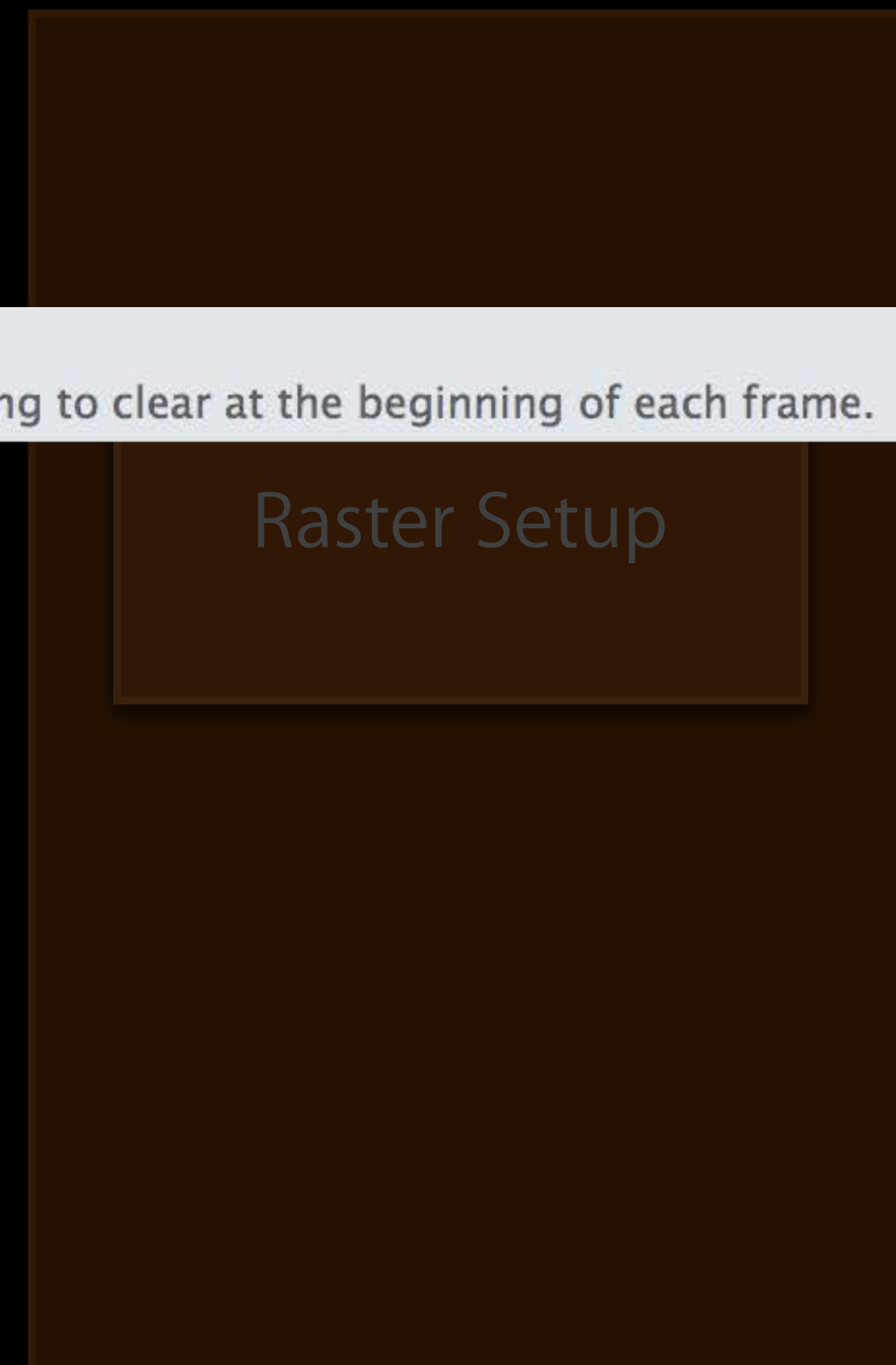
Logical Buffer Load

Your app caused a framebuffer load operation by the GPU. A typical cause is failing to clear at the beginning of each frame.

V₁₃ V₁₄ V₁₅

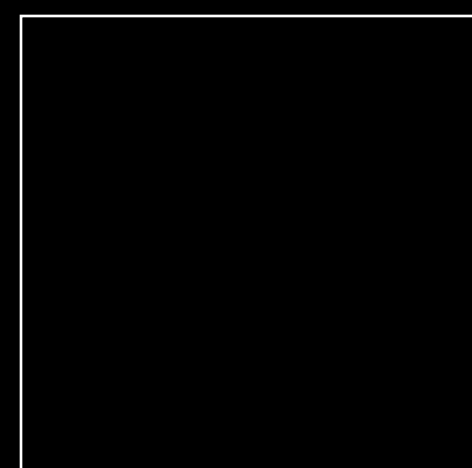
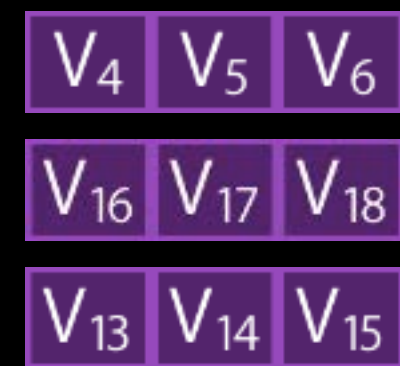


Raster Setup

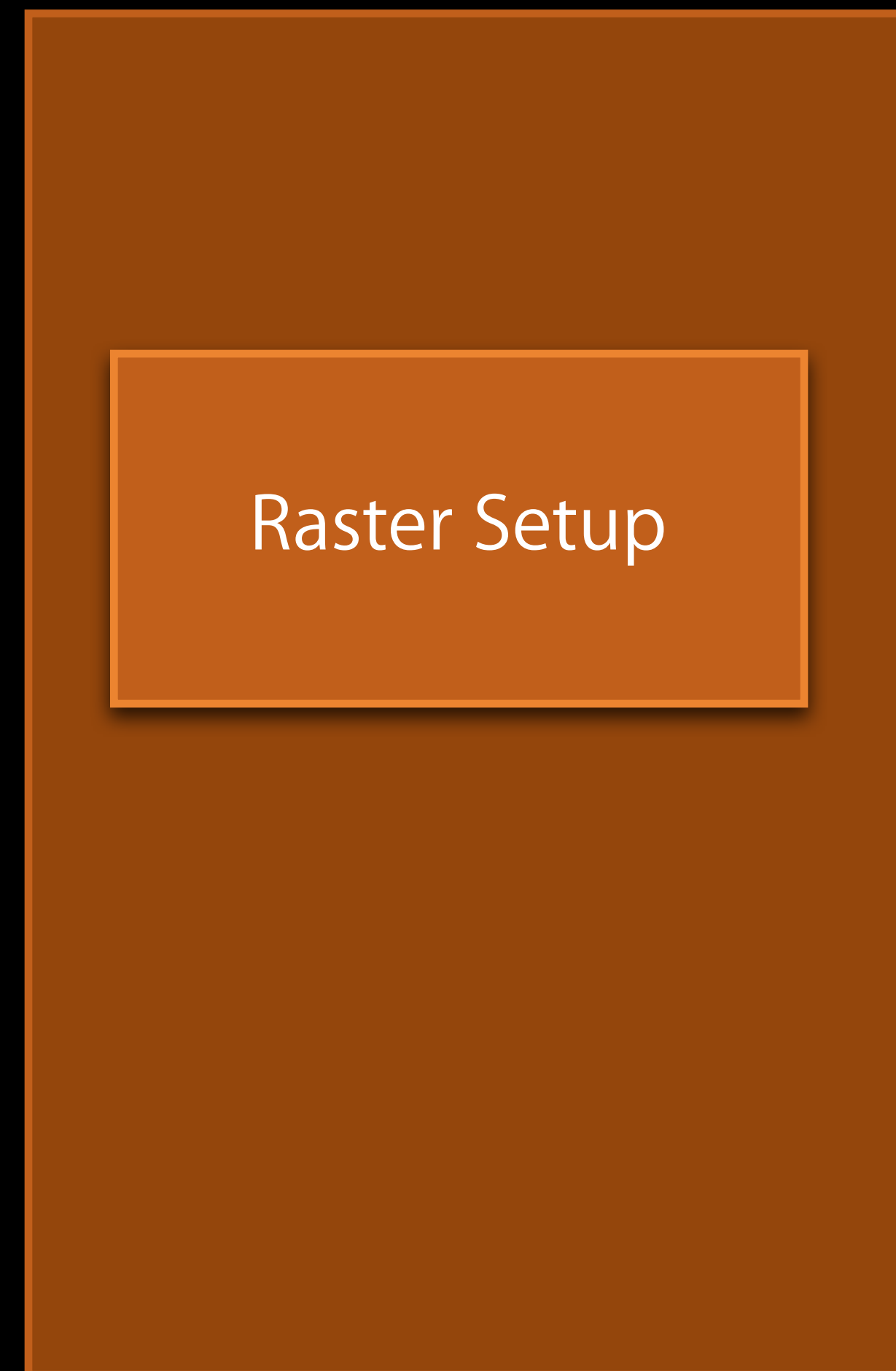


- Rasterizer uses tile-sized embedded memory

Unified Memory

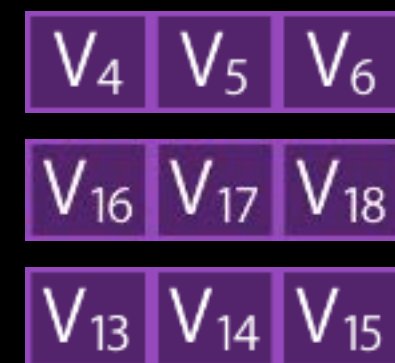


GPU

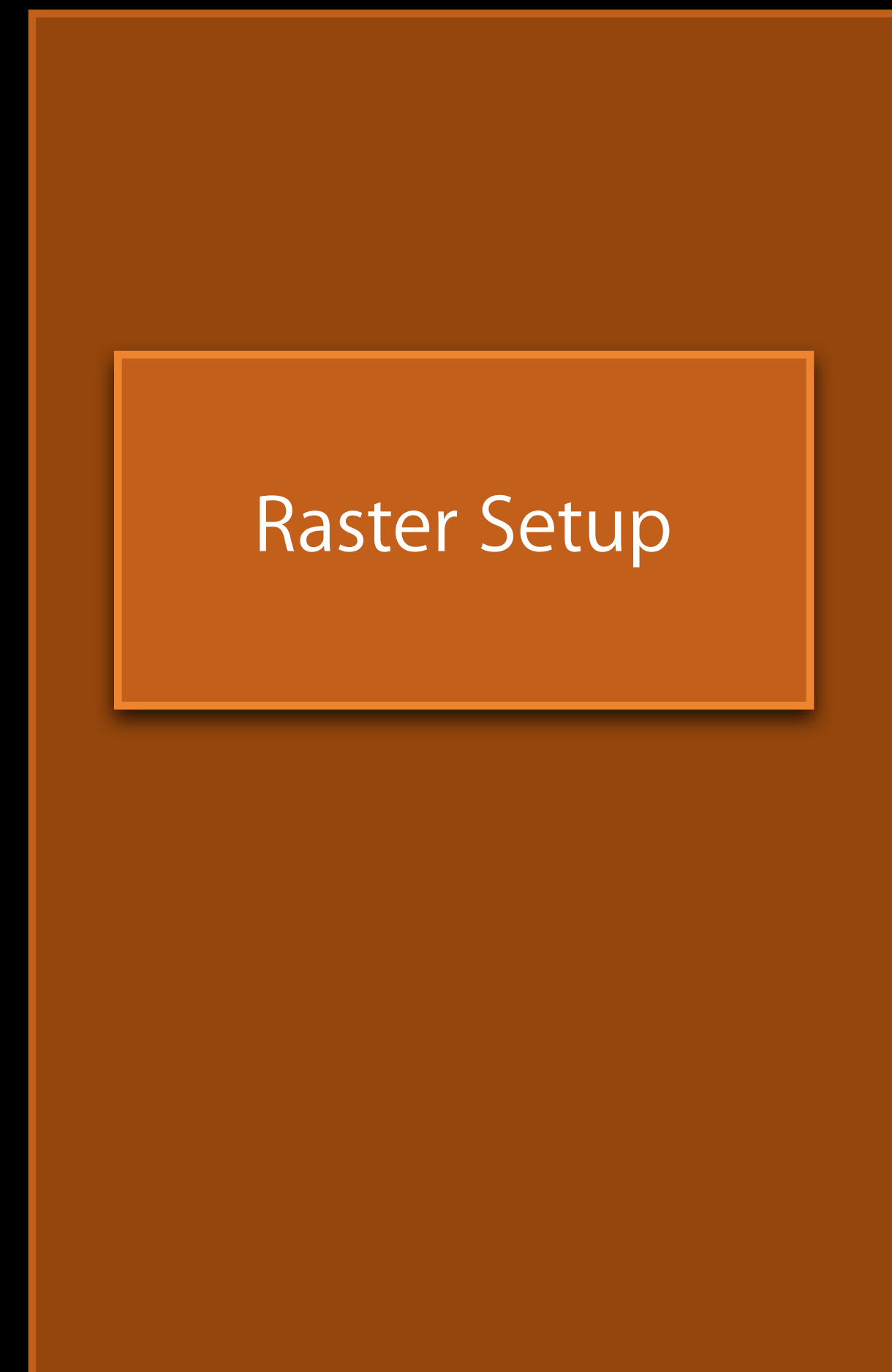


- Rasterizer uses tile-sized embedded memory
- If data is in render buffer, GPU must perform a costly load

Unified Memory

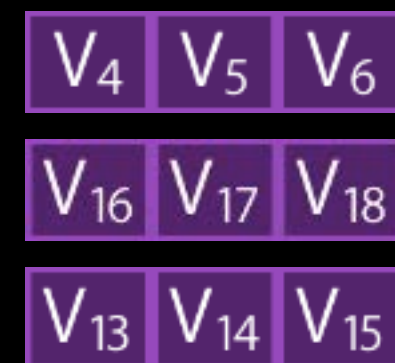


GPU

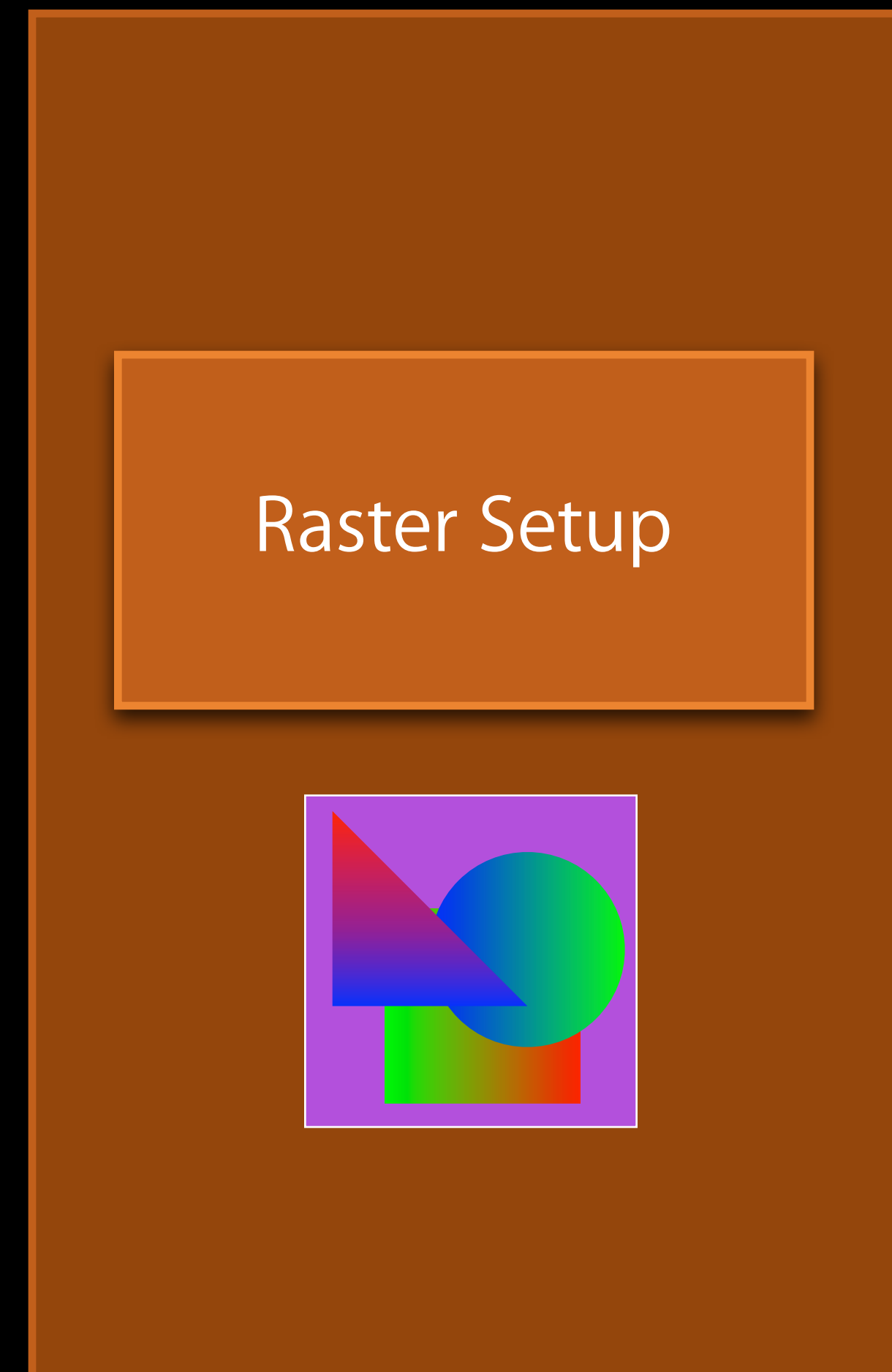


- Rasterizer uses tile-sized embedded memory
- If data is in render buffer, GPU must perform a costly load

Unified Memory

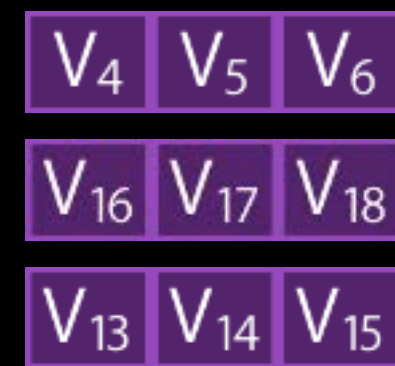


GPU

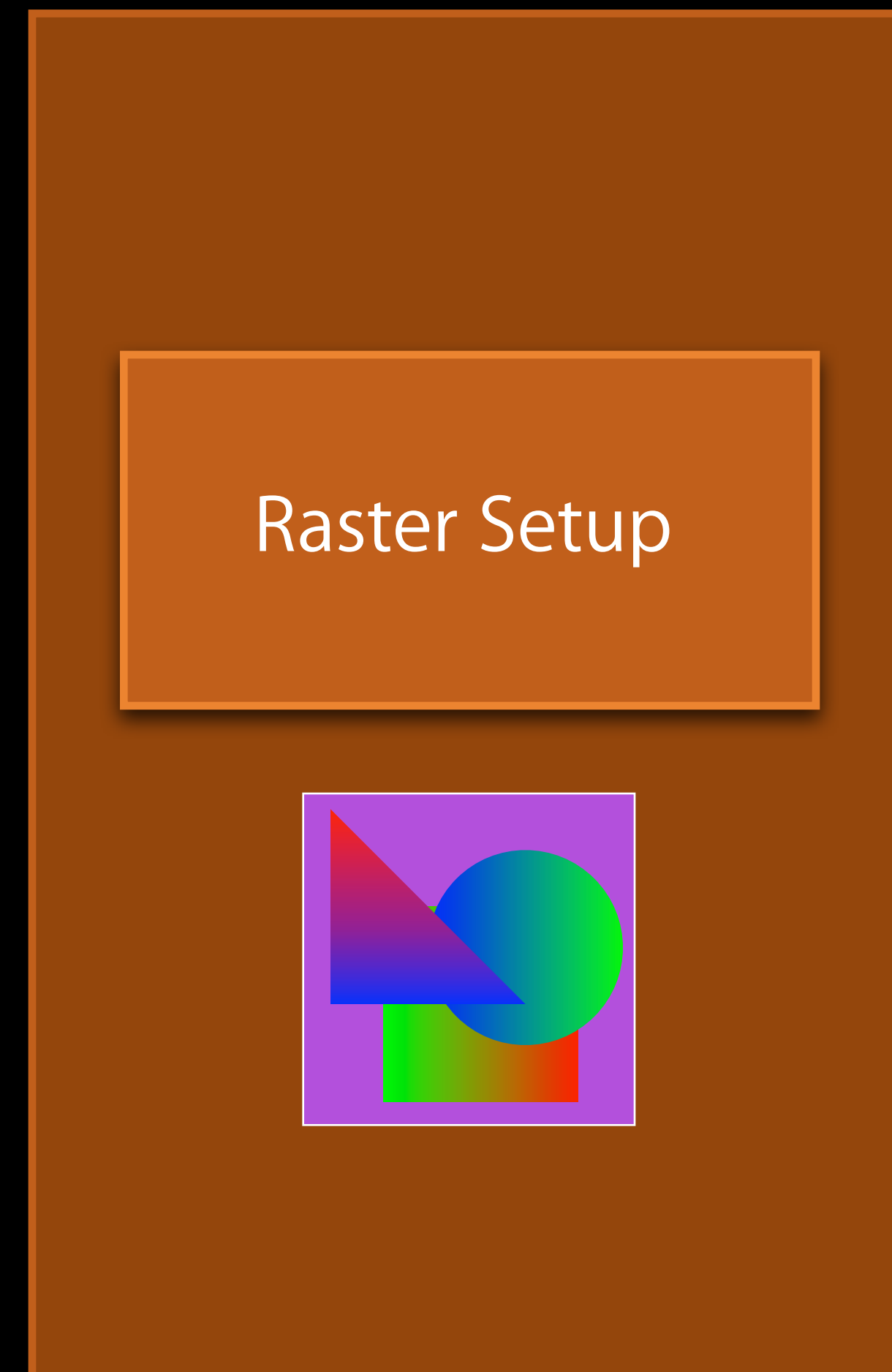


- If there is data in the depth buffer, GPU must load it also

Unified Memory

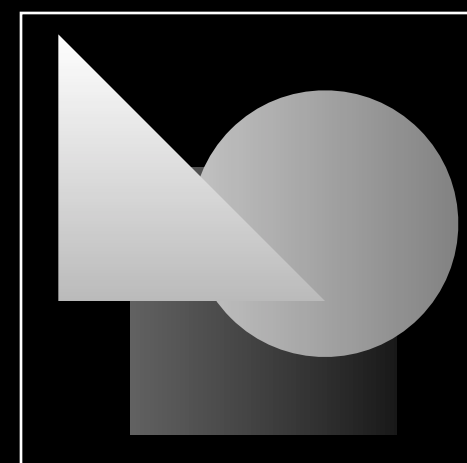
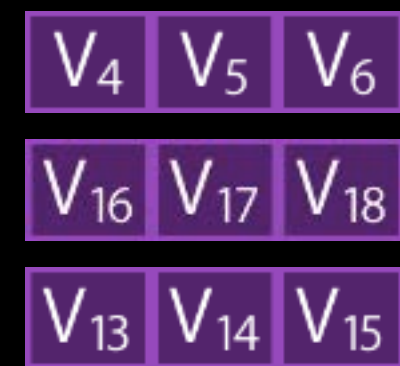


GPU



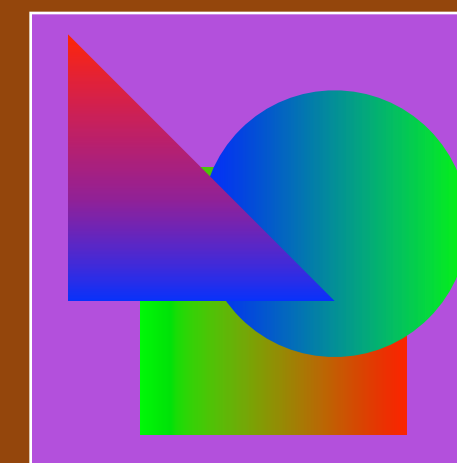
- If there is data in the depth buffer, GPU must load it also

Unified Memory



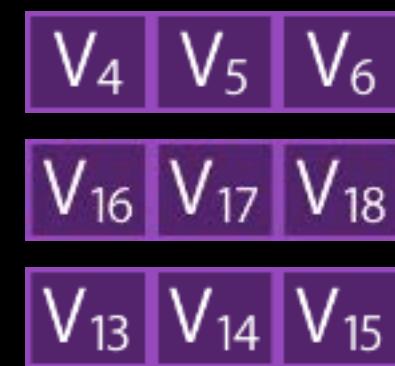
GPU

Raster Setup

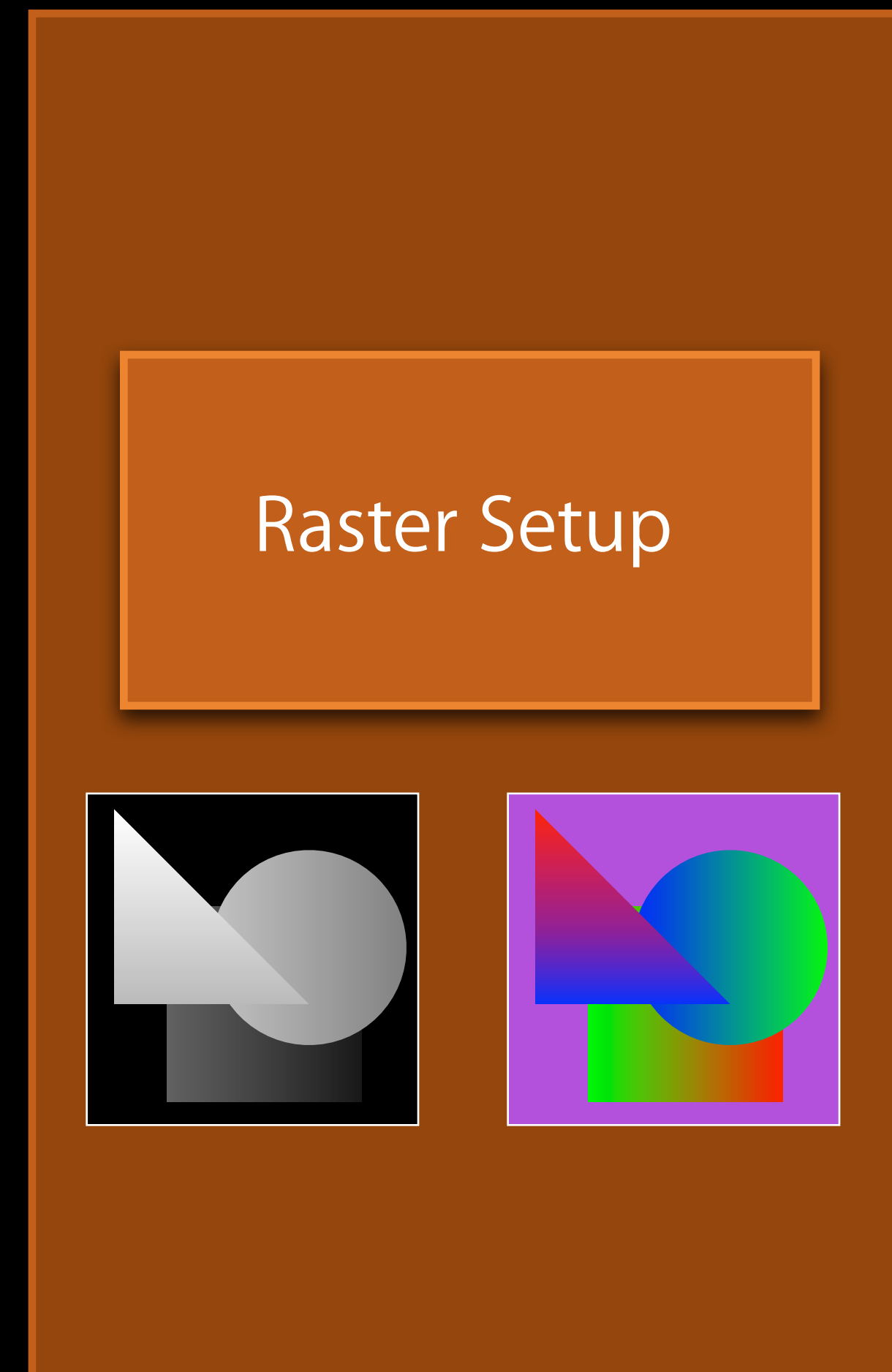


- If there is data in the depth buffer, GPU must load it also

Unified Memory

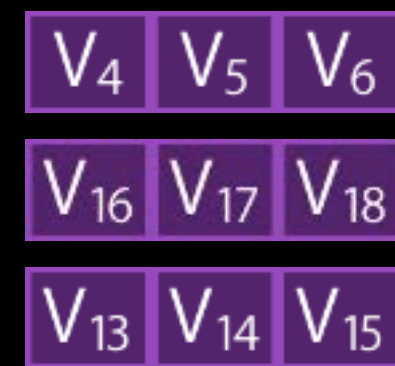


GPU

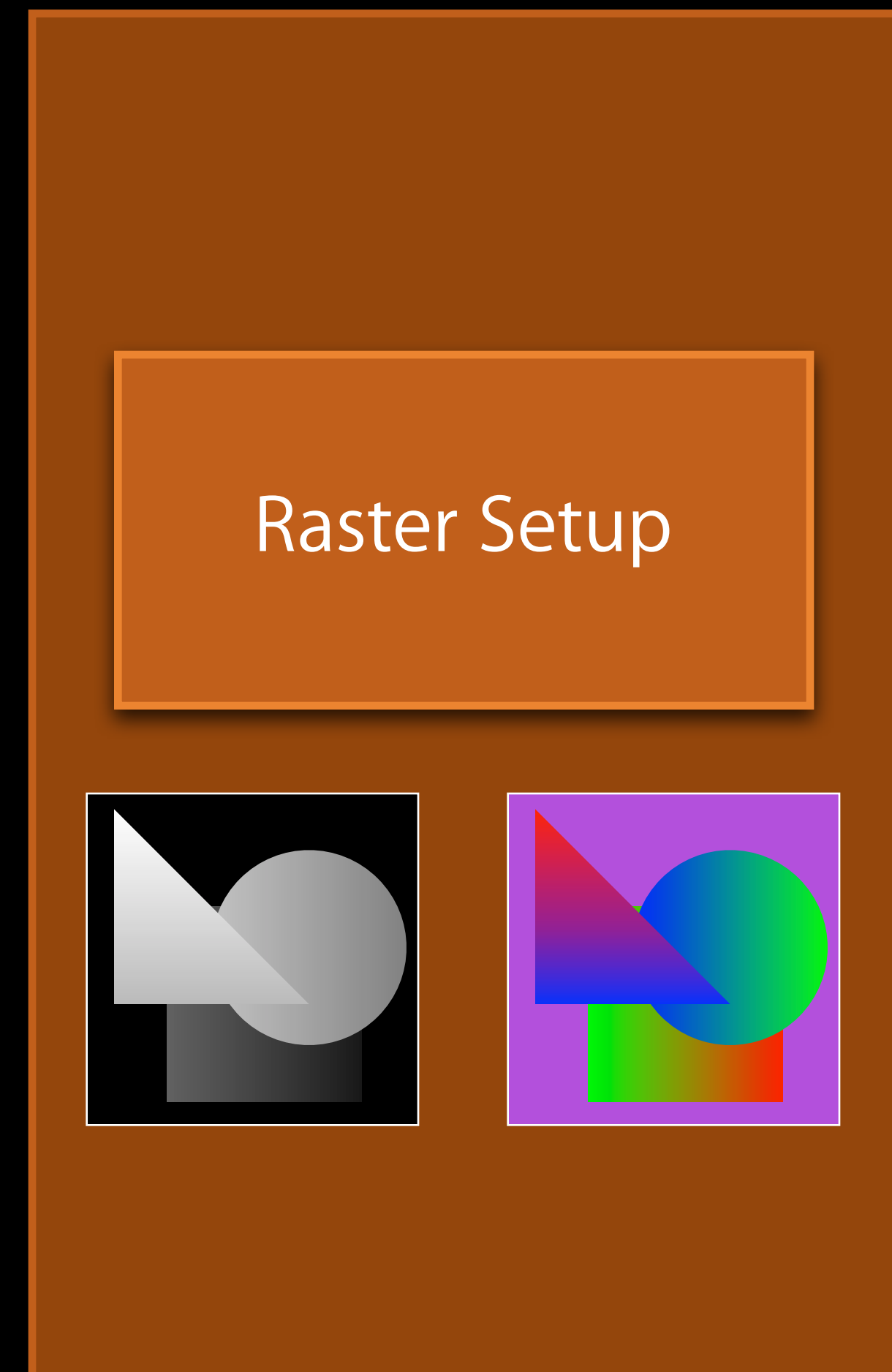


- Loading a tile is called a Logical Buffer Load
 - Developers can avoid these

Unified Memory

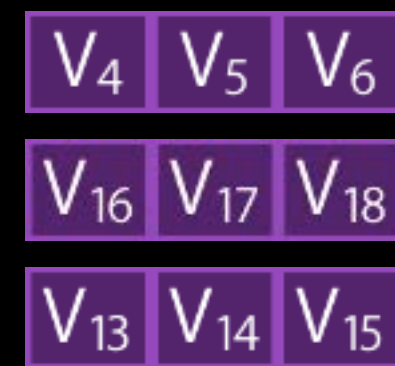


GPU



- Calling `glClear` before rendering skips Logical Buffer Load

Unified Memory

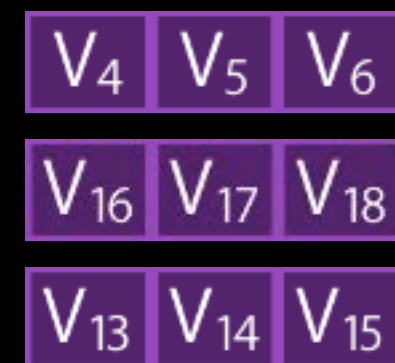


GPU

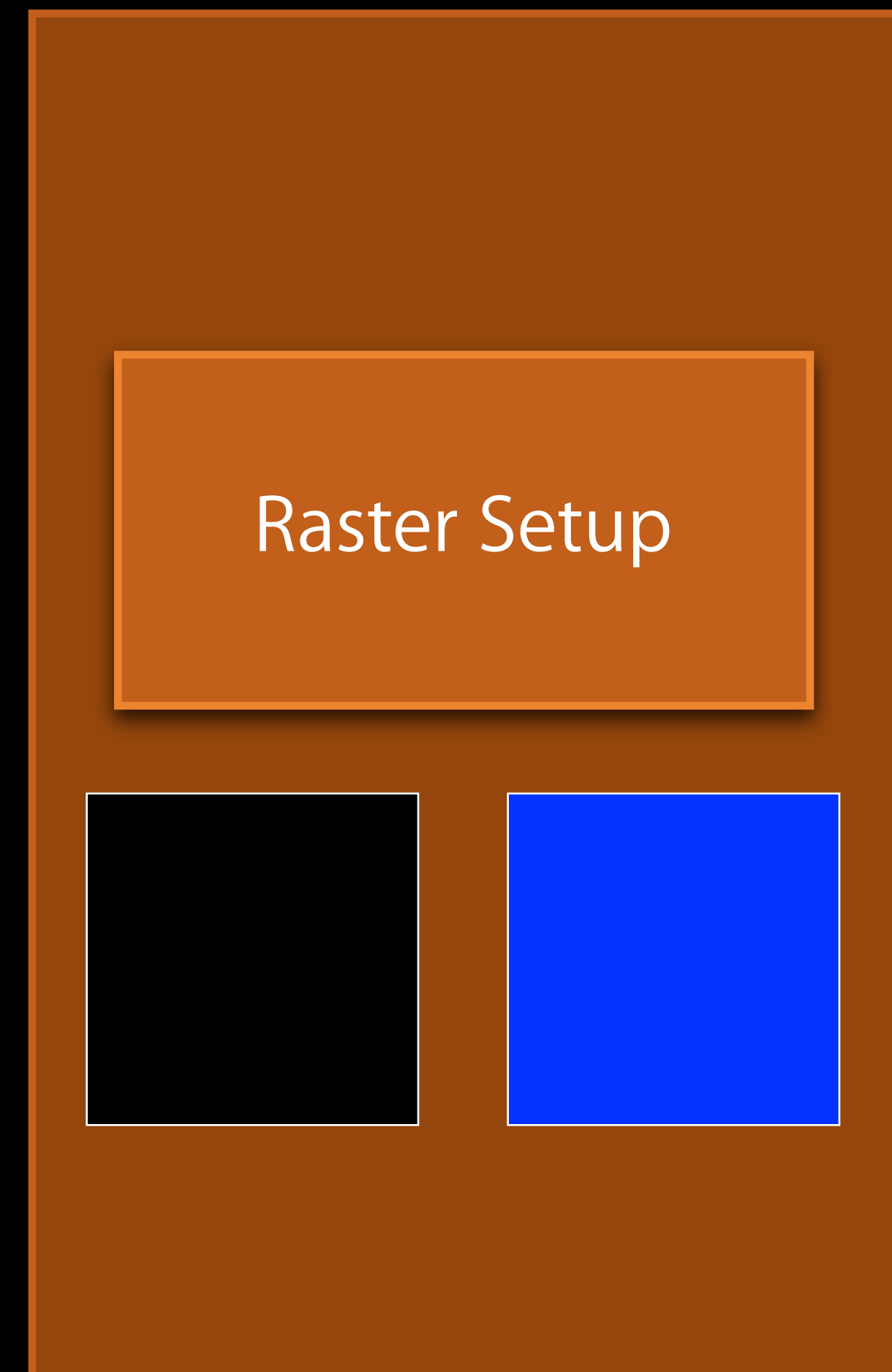


- Calling `glClear` before rendering skips Logical Buffer Load
 - Can immediately rasterize to embedded memory

Unified Memory

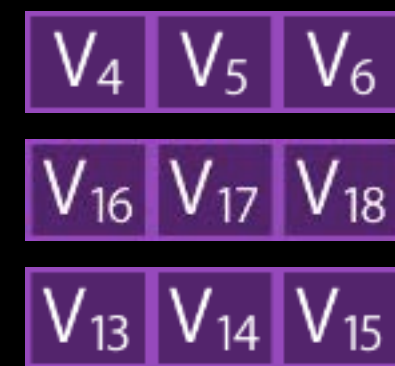


GPU

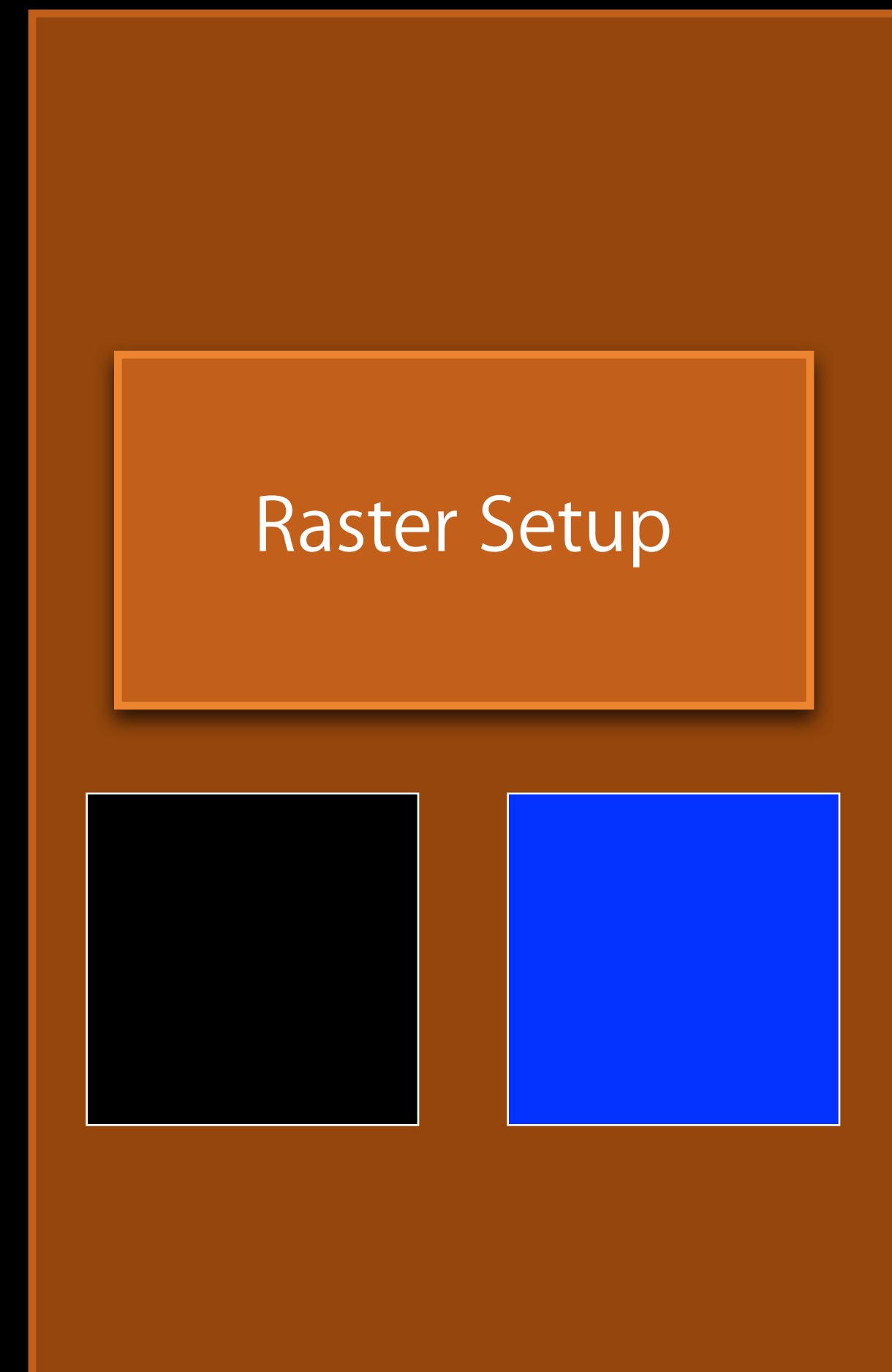


- Logical Buffer Loads also happen when switching render buffer

Unified Memory

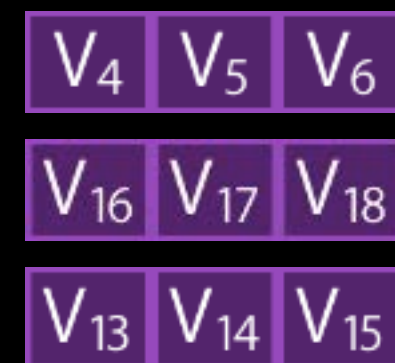


GPU

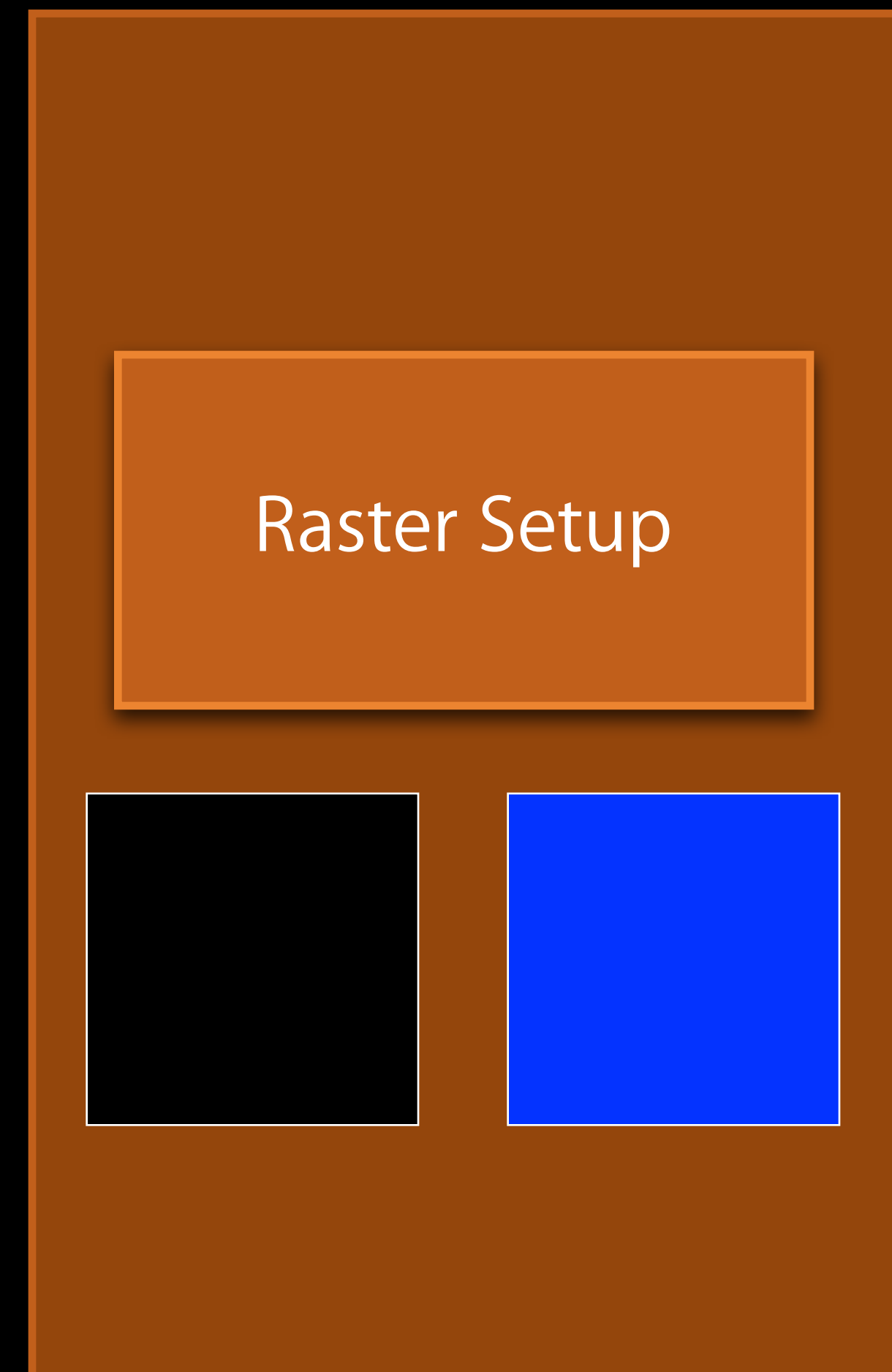


- Logical Buffer Loads also happen when switching render buffer
 - Render to texture, render to new buffer, render to texture again

Unified Memory

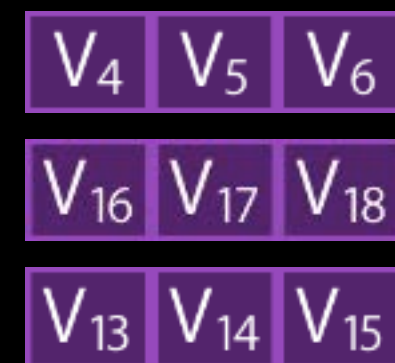


GPU

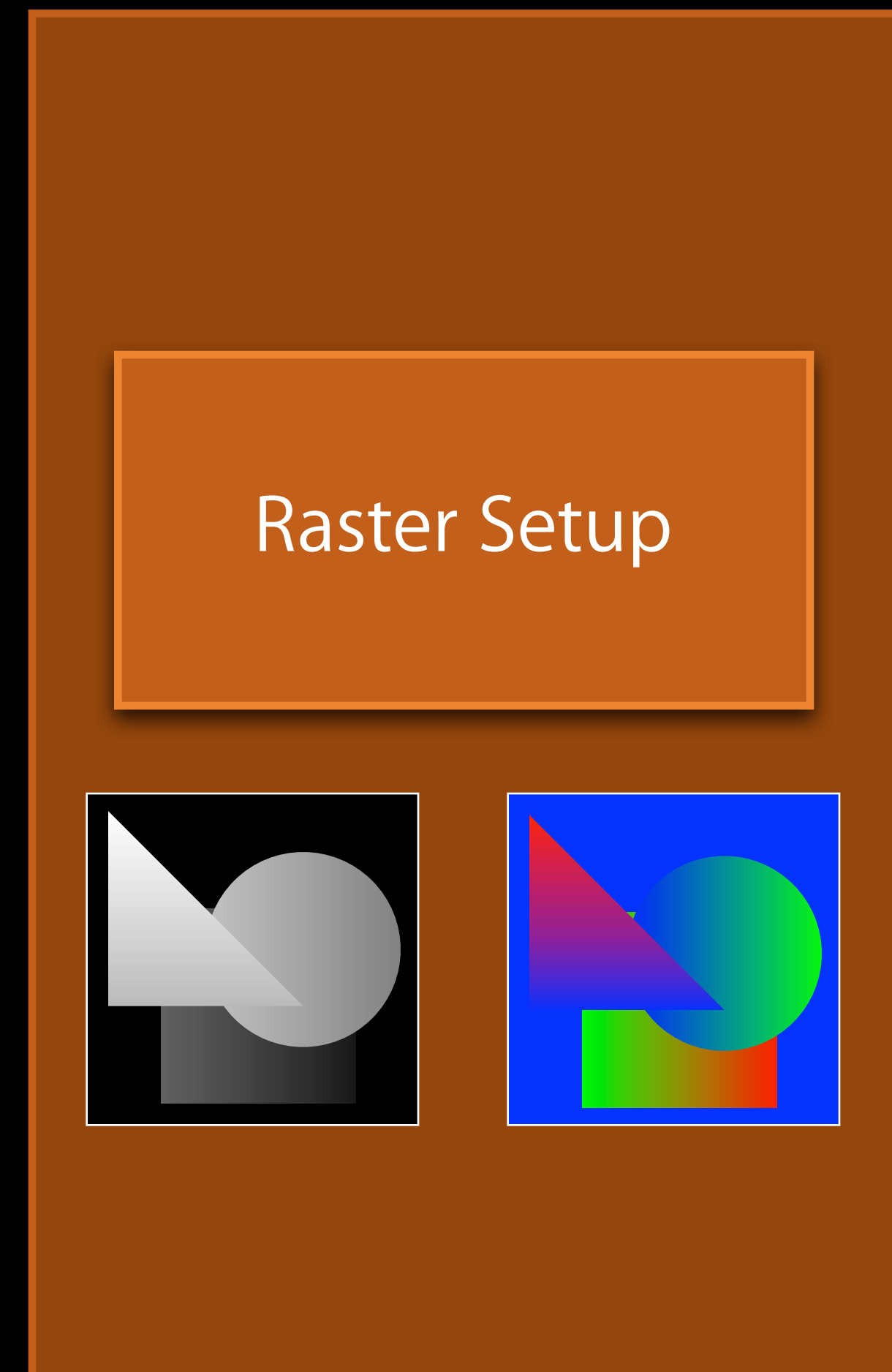


- Logical Buffer Loads also happen when switching render buffer
 - Render to texture, render to new buffer, render to texture again

Unified Memory

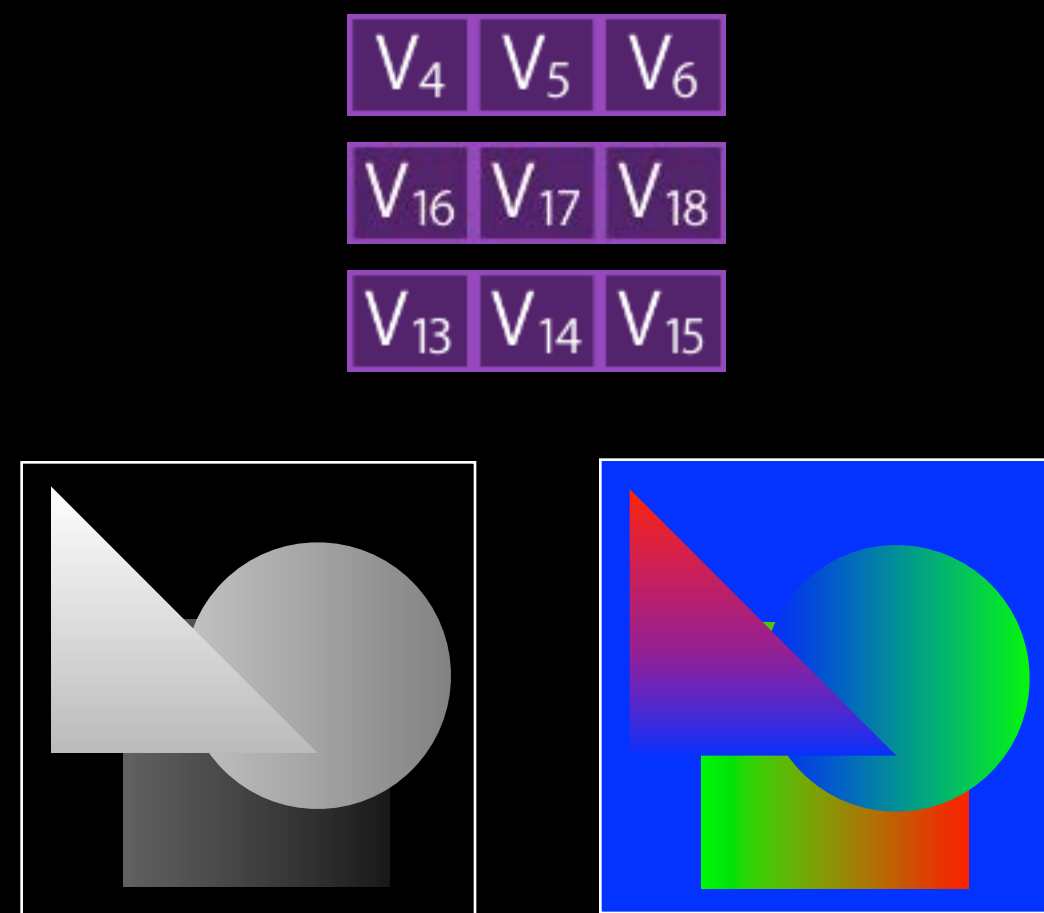


GPU



- Logical Buffer Loads also happen when switching render buffer
 - Render to texture, render to new buffer, render to texture again

Unified Memory

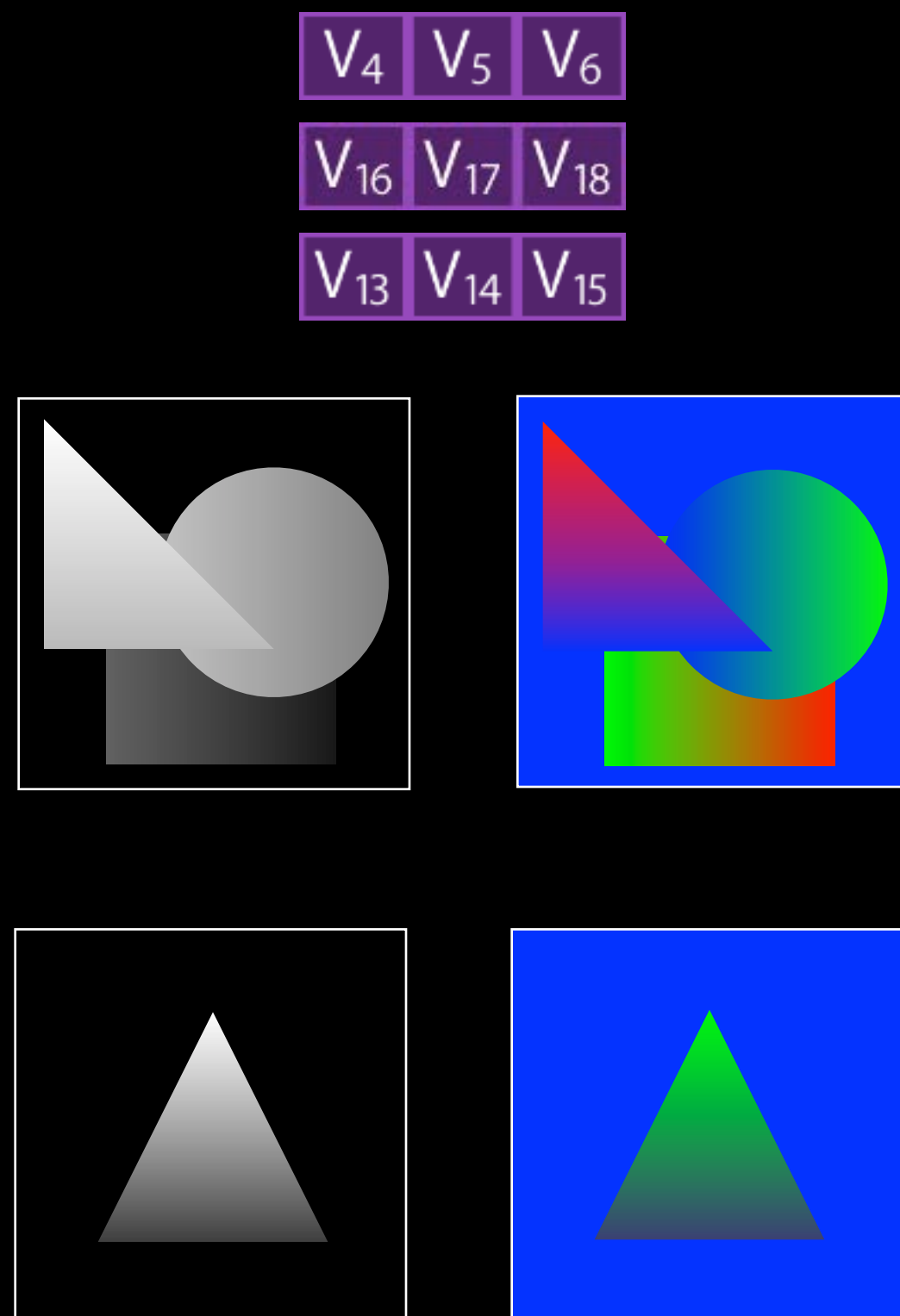


GPU

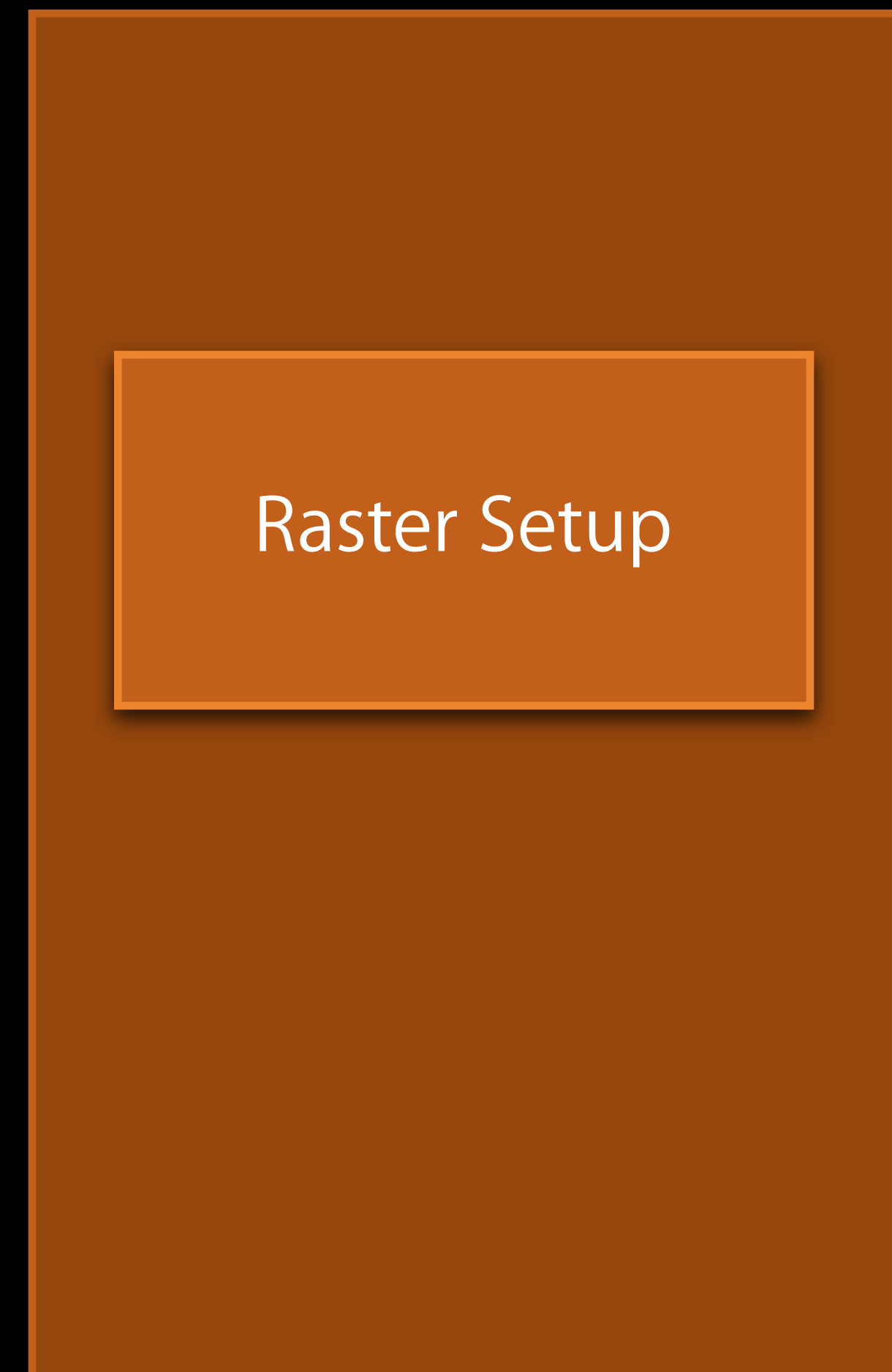


- Logical Buffer Loads also happen when switching render buffer
 - Render to texture, render to new buffer, render to texture again

Unified Memory

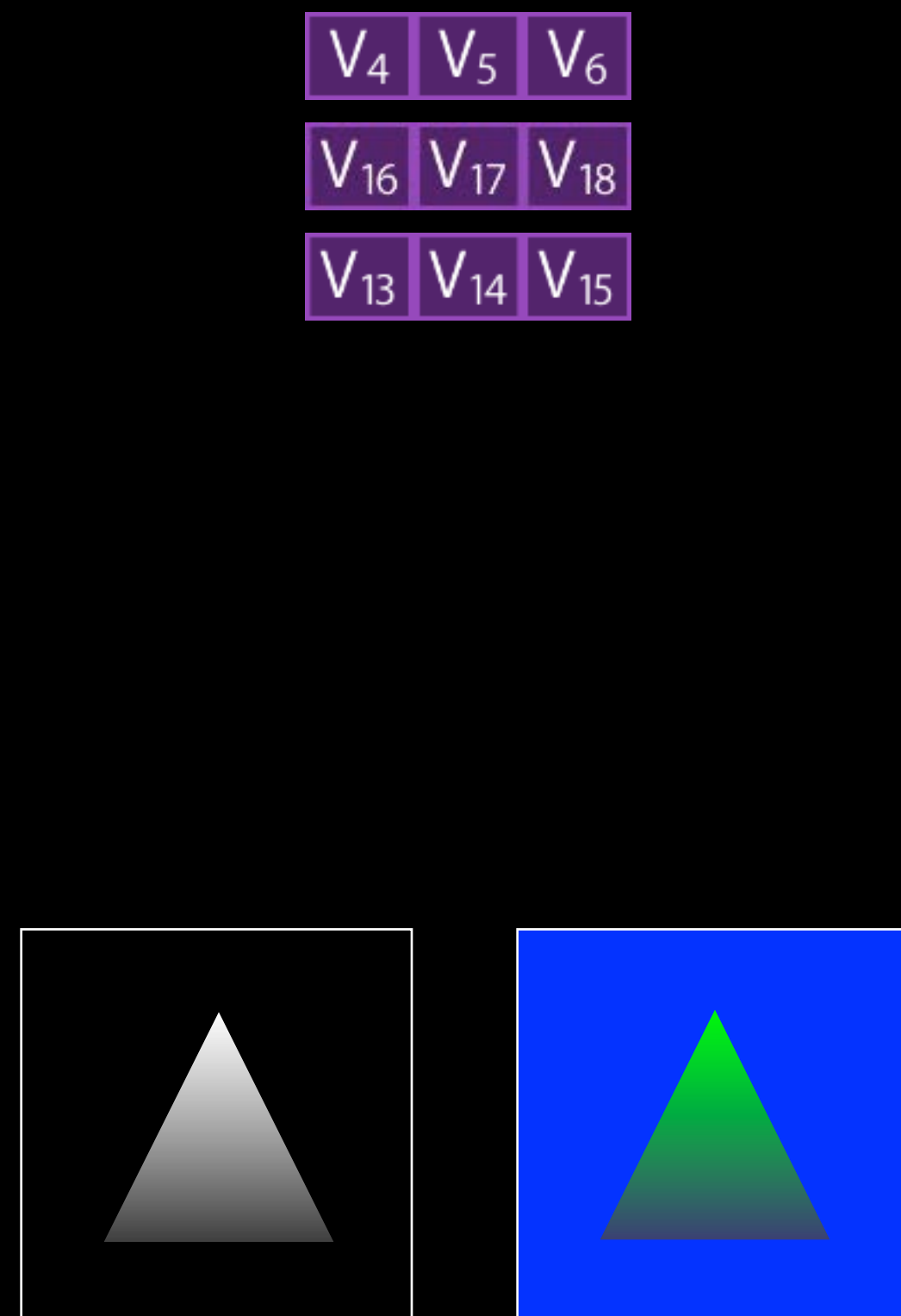


GPU

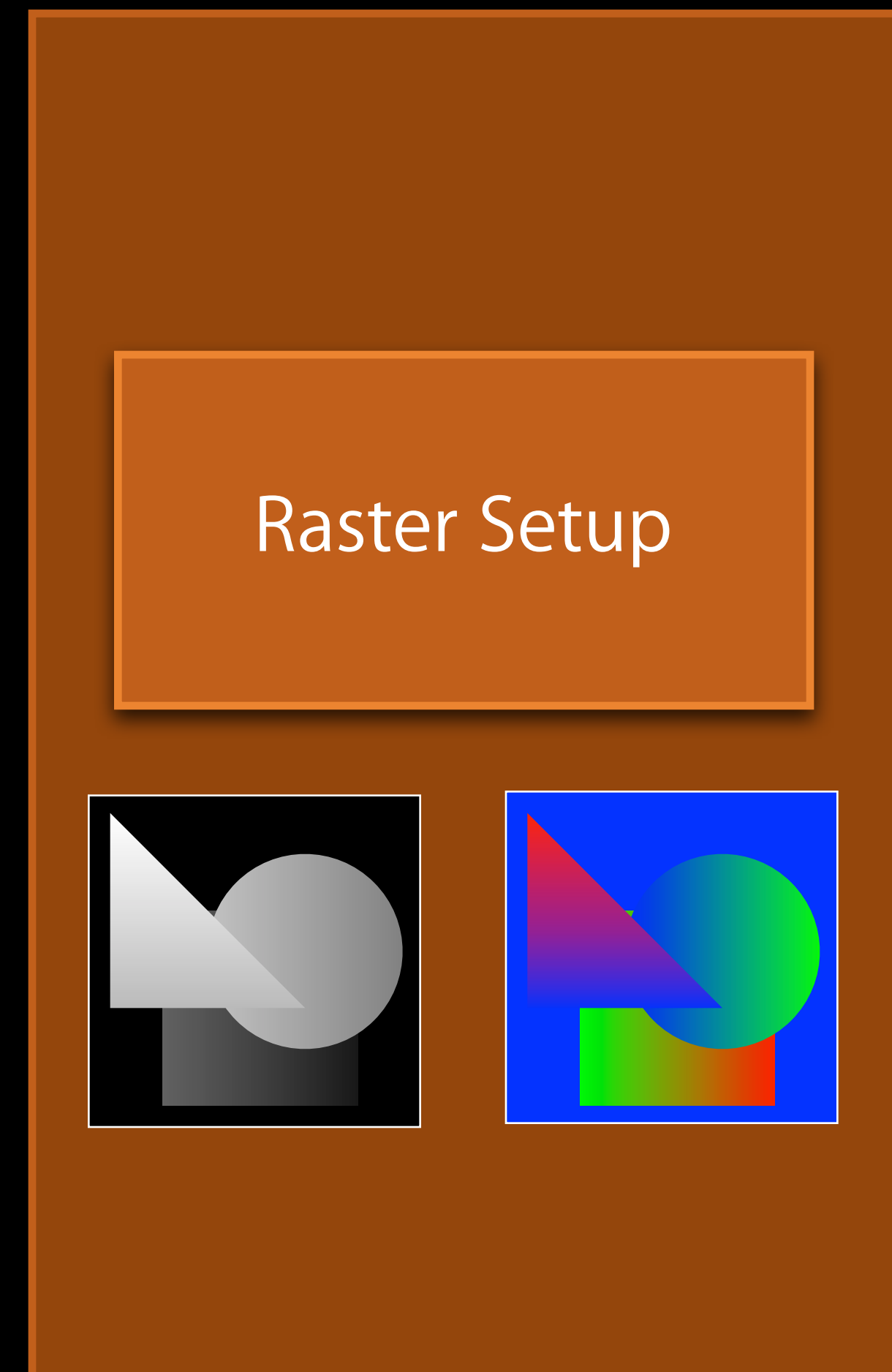


- Logical Buffer Loads also happen when switching render buffer
 - Render to texture, render to new buffer, render to texture again

Unified Memory

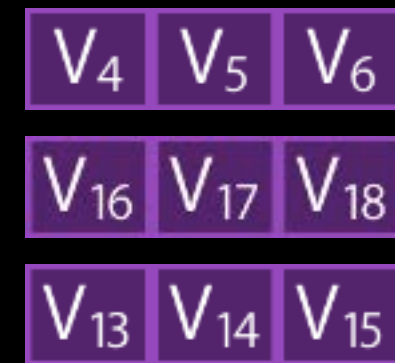


GPU

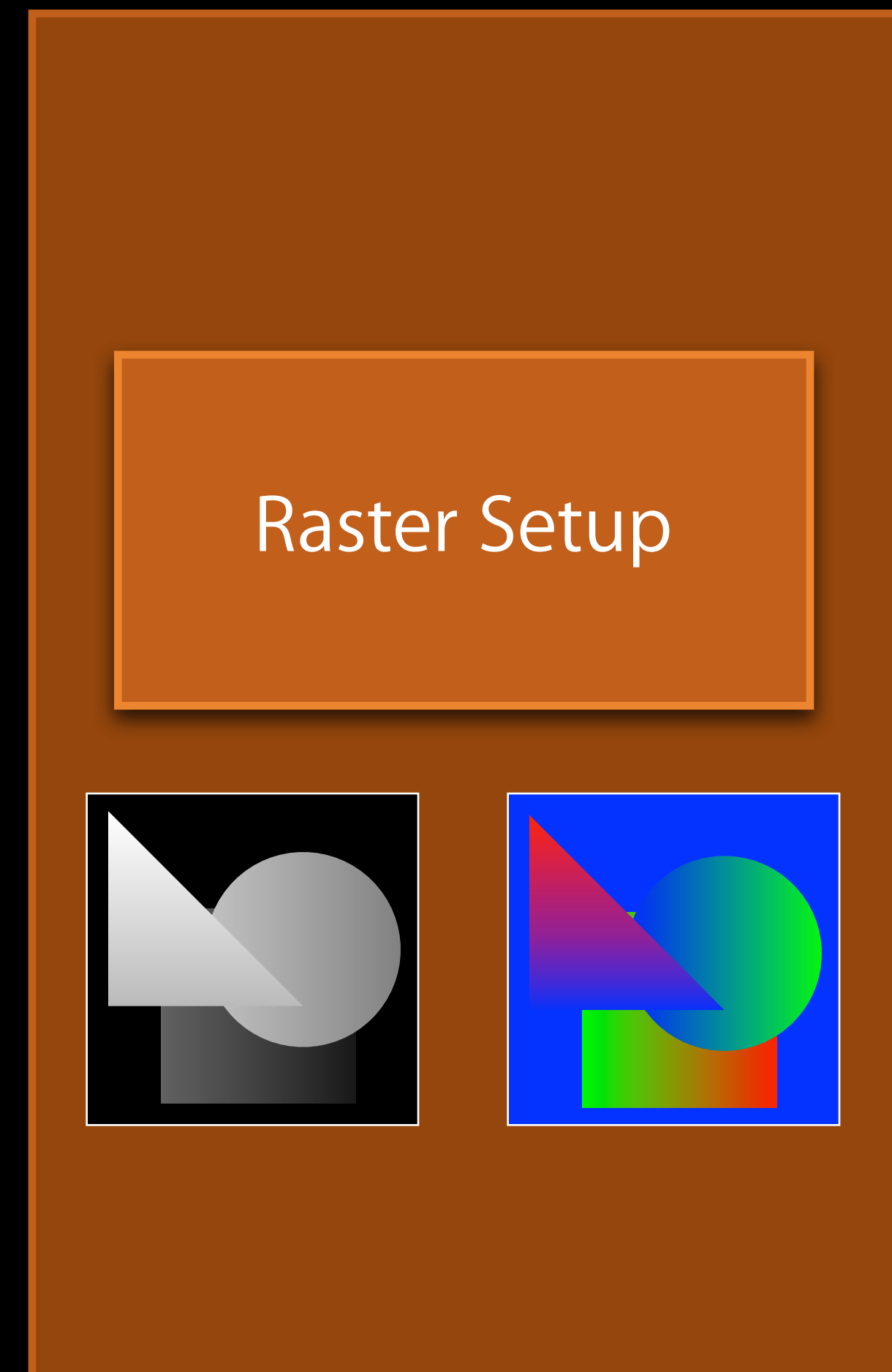


- Developers should avoid frequent switching of renderbuffers
 - Complete rendering to one buffer before switching to another

Unified Memory

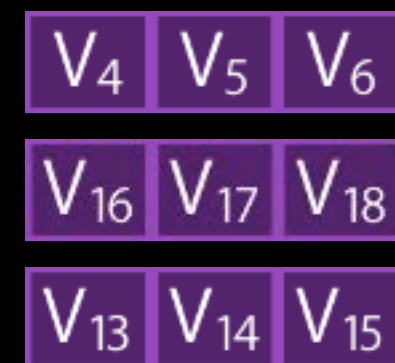


GPU



Rasterization

Unified Memory



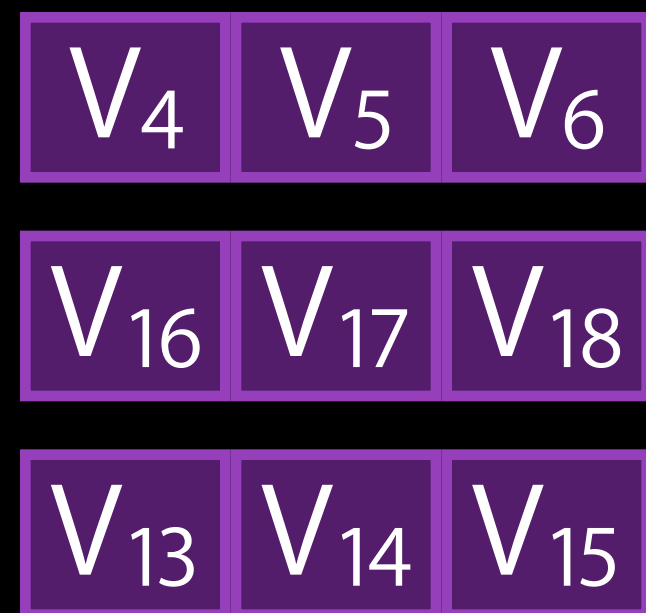
GPU

A diagram of a GPU. It is represented by a large orange rectangle. Inside this rectangle, centered, is a smaller orange rectangle with a thin white border. The text "Raster Setup" is written in white inside this smaller rectangle.

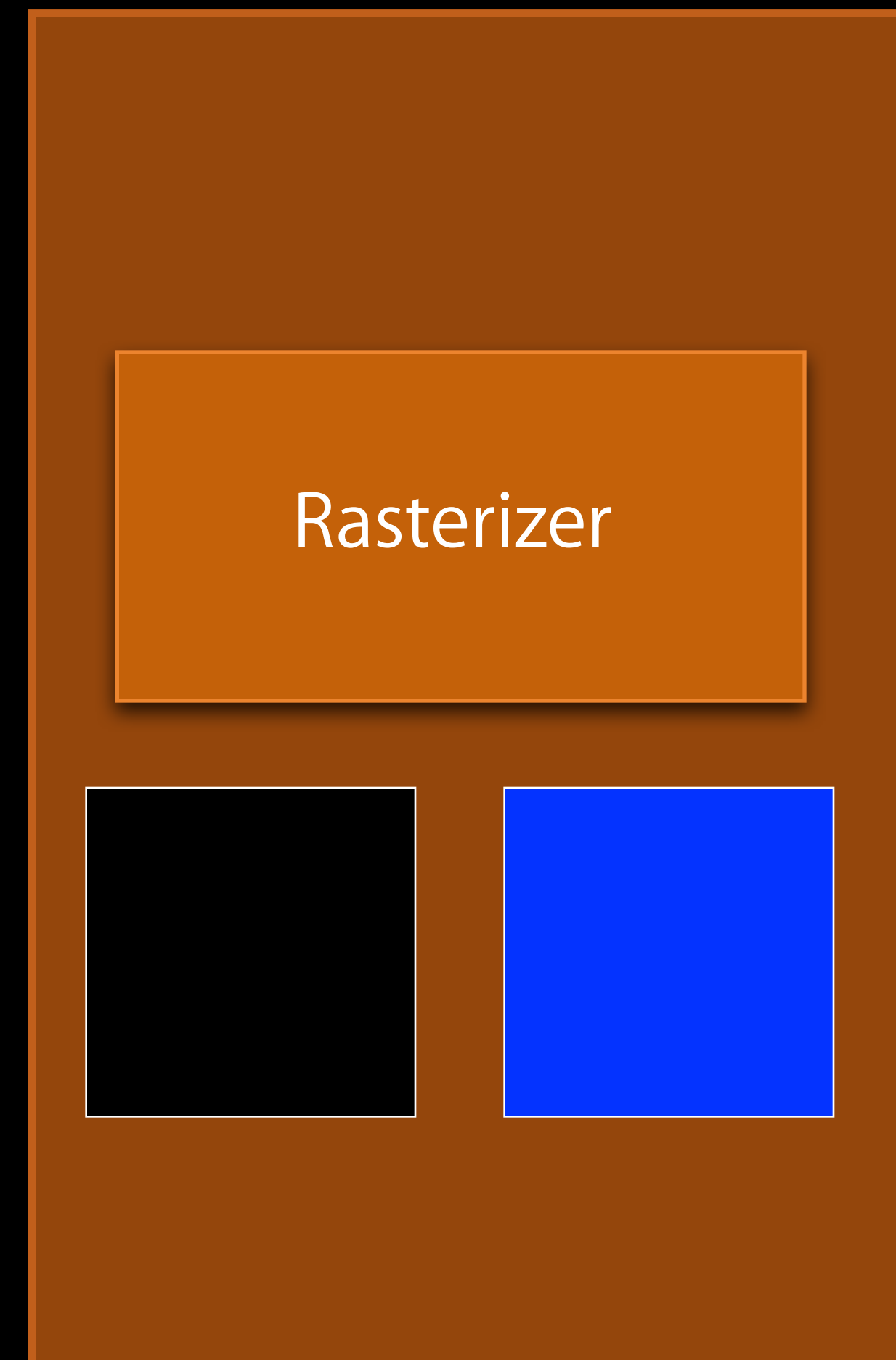
Raster Setup

Rasterization

Unified Memory

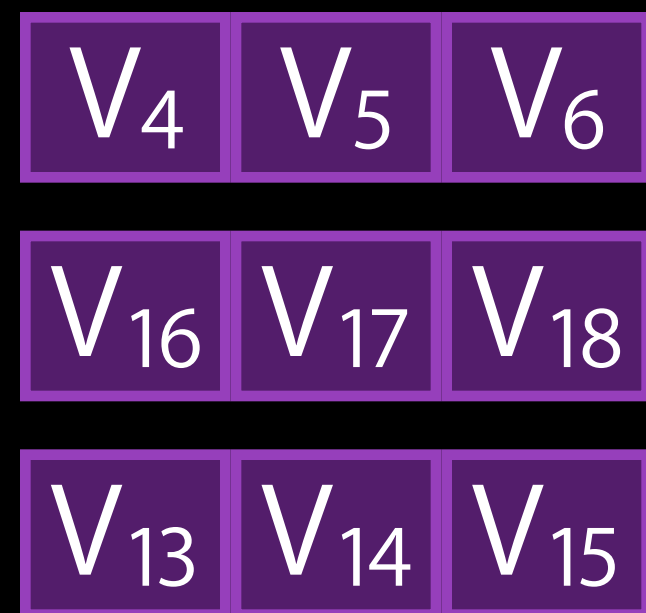


GPU

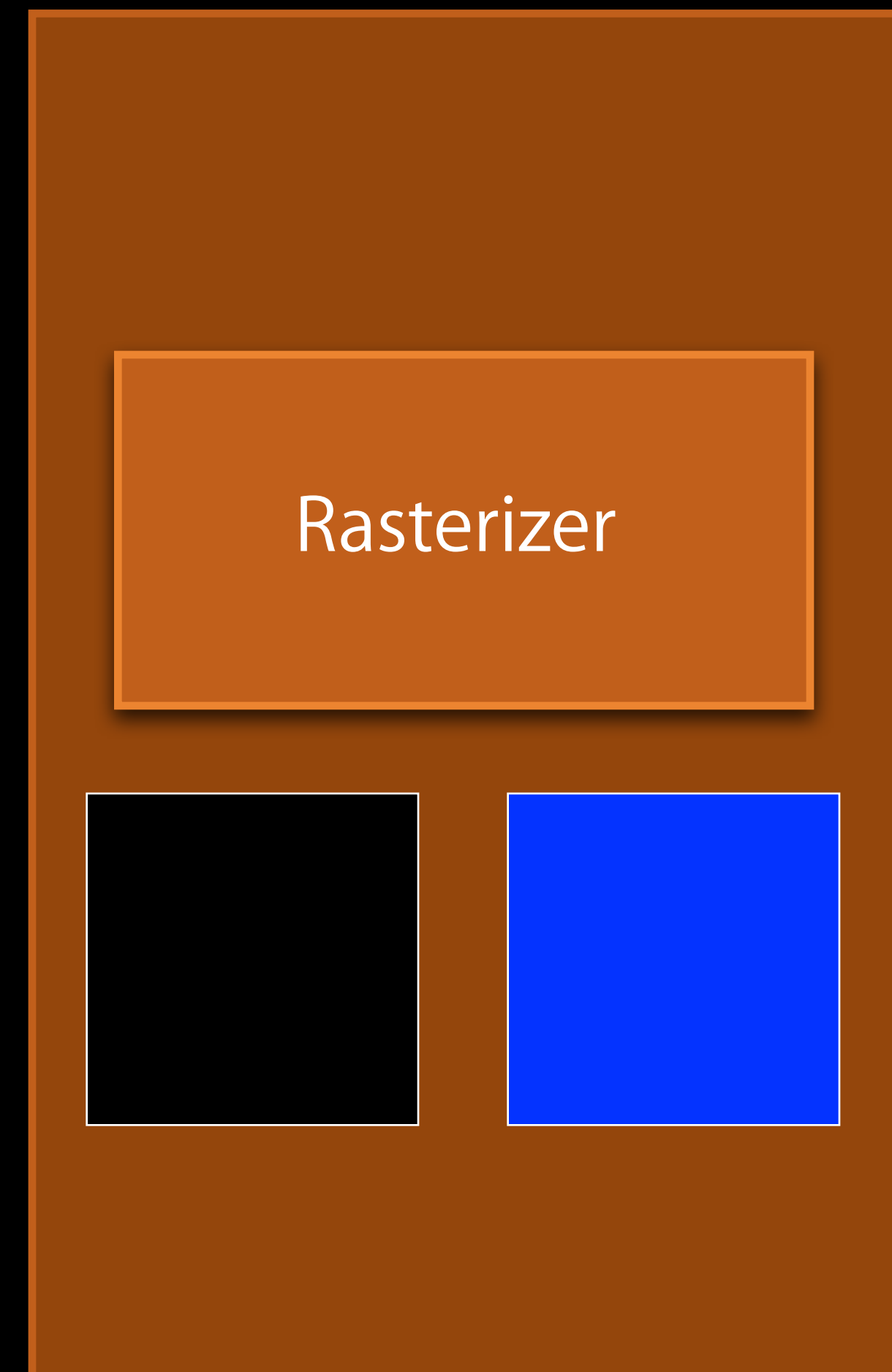


- GPU reads triangles assigned to tile
 - XY pixel coordinates and Z values generated

Unified Memory

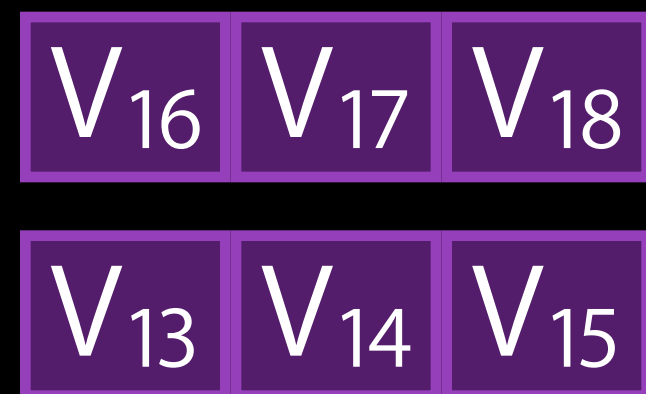


GPU

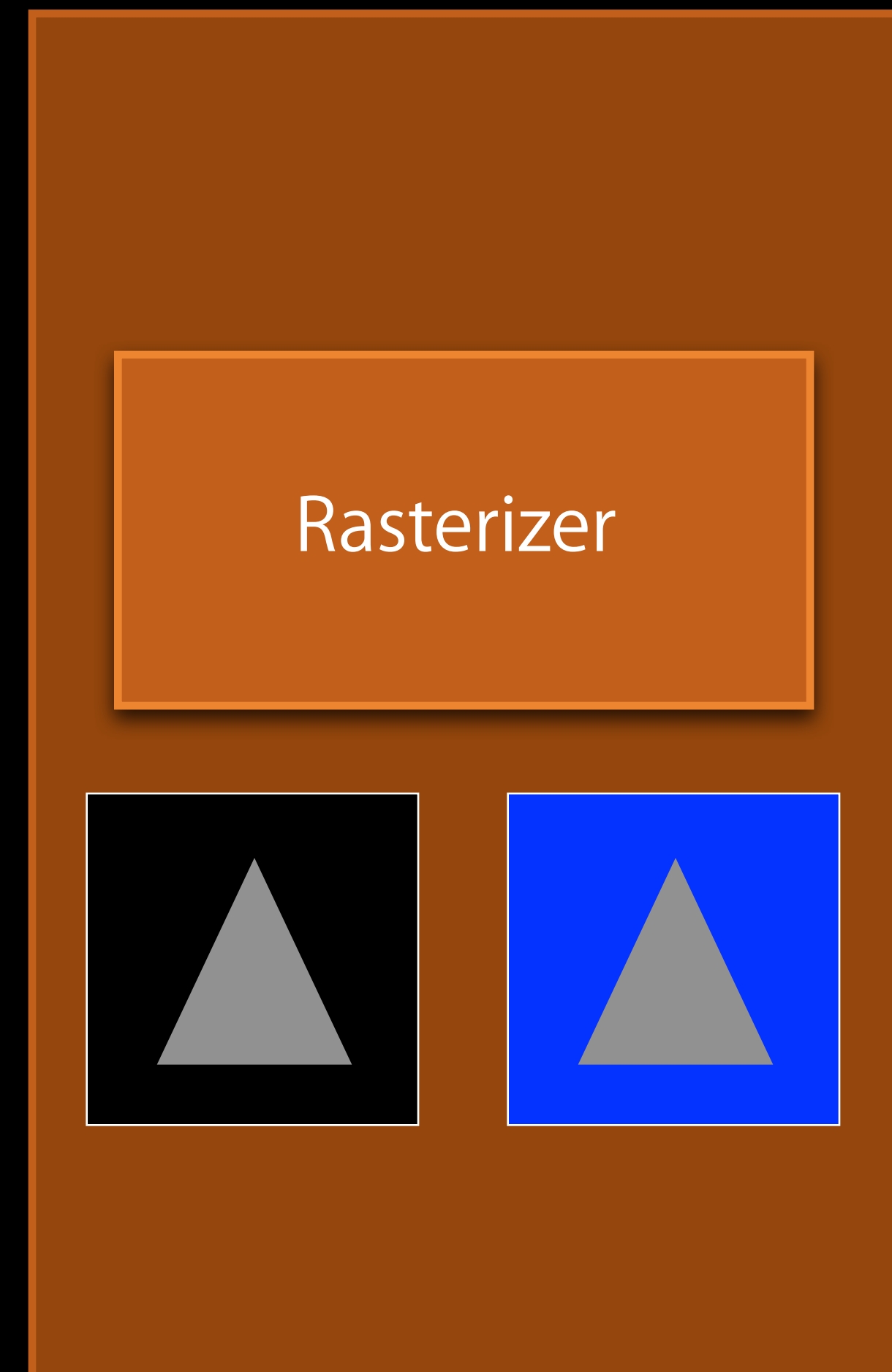


- GPU reads triangles assigned to tile
 - XY pixel coordinates and Z values generated

Unified Memory

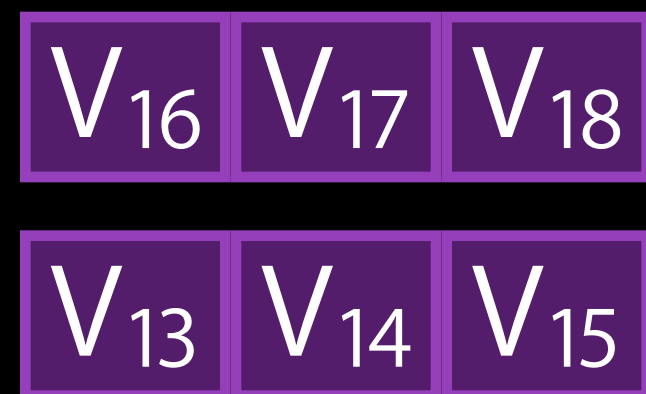


GPU

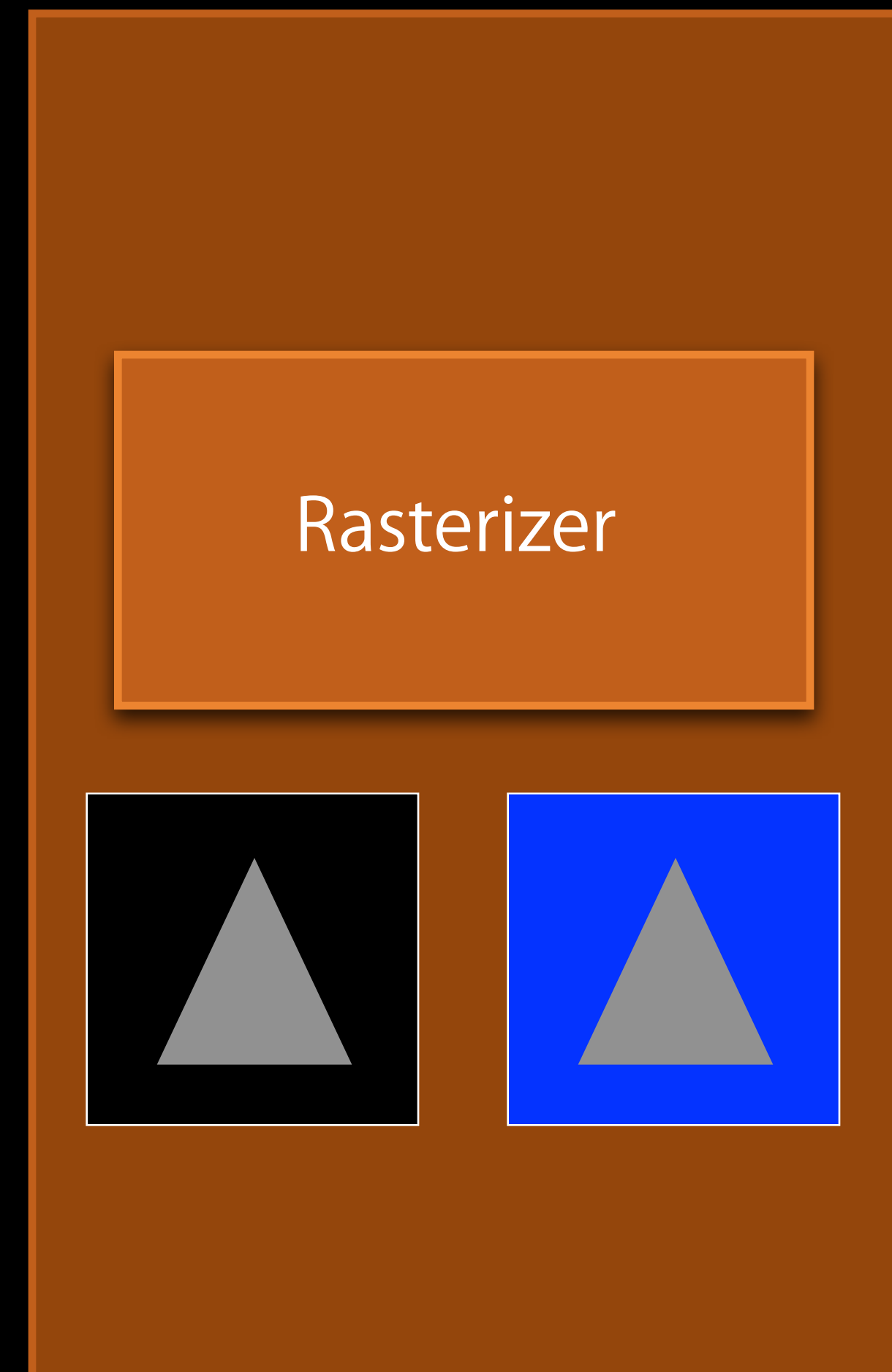


- Fragment shader not run yet
 - Positions and depth calculated only

Unified Memory

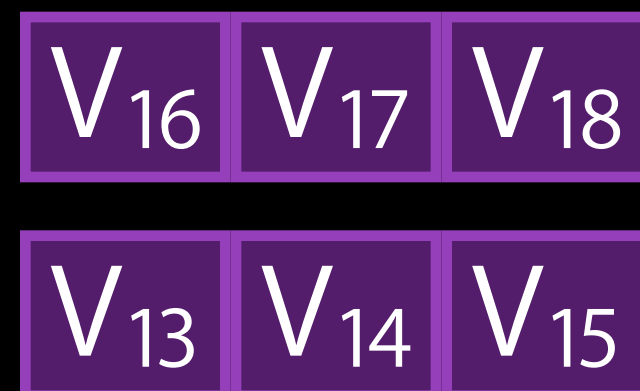


GPU

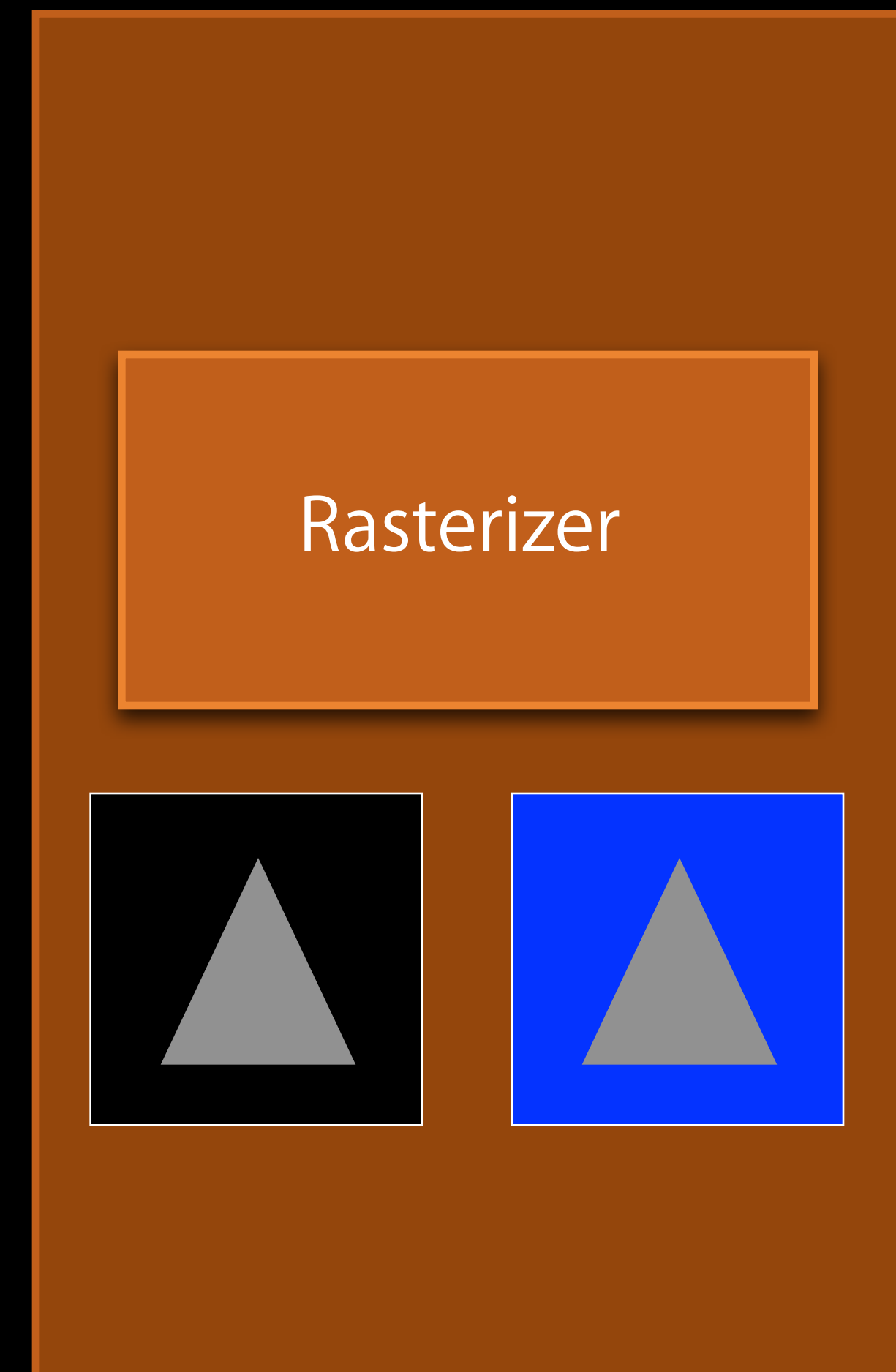


- Hidden Surface Removal performed
 - GPU can reject fragments before shading them

Unified Memory



GPU

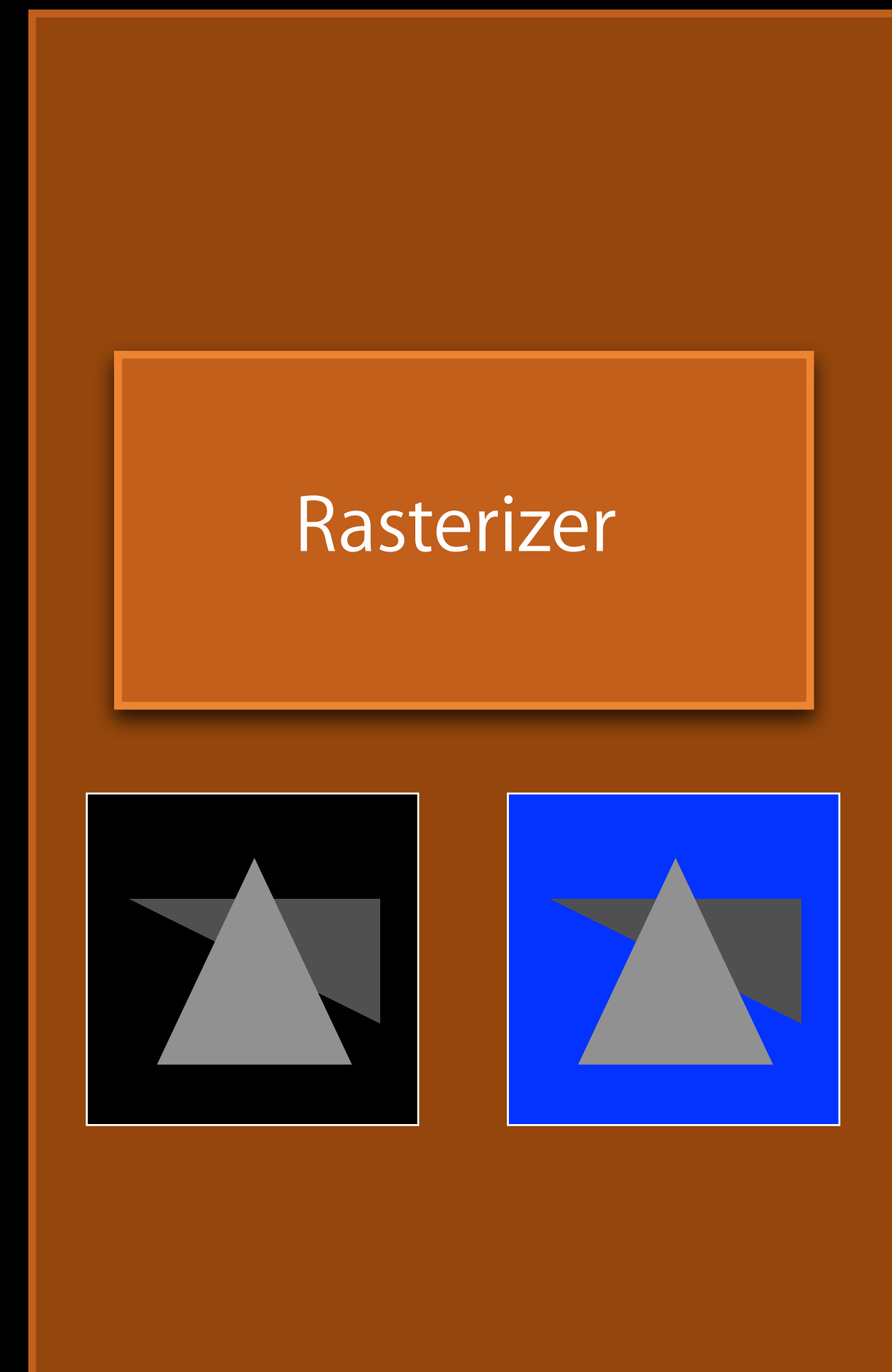


- Hidden Surface Removal performed
 - GPU can reject fragments before shading them

Unified Memory



GPU

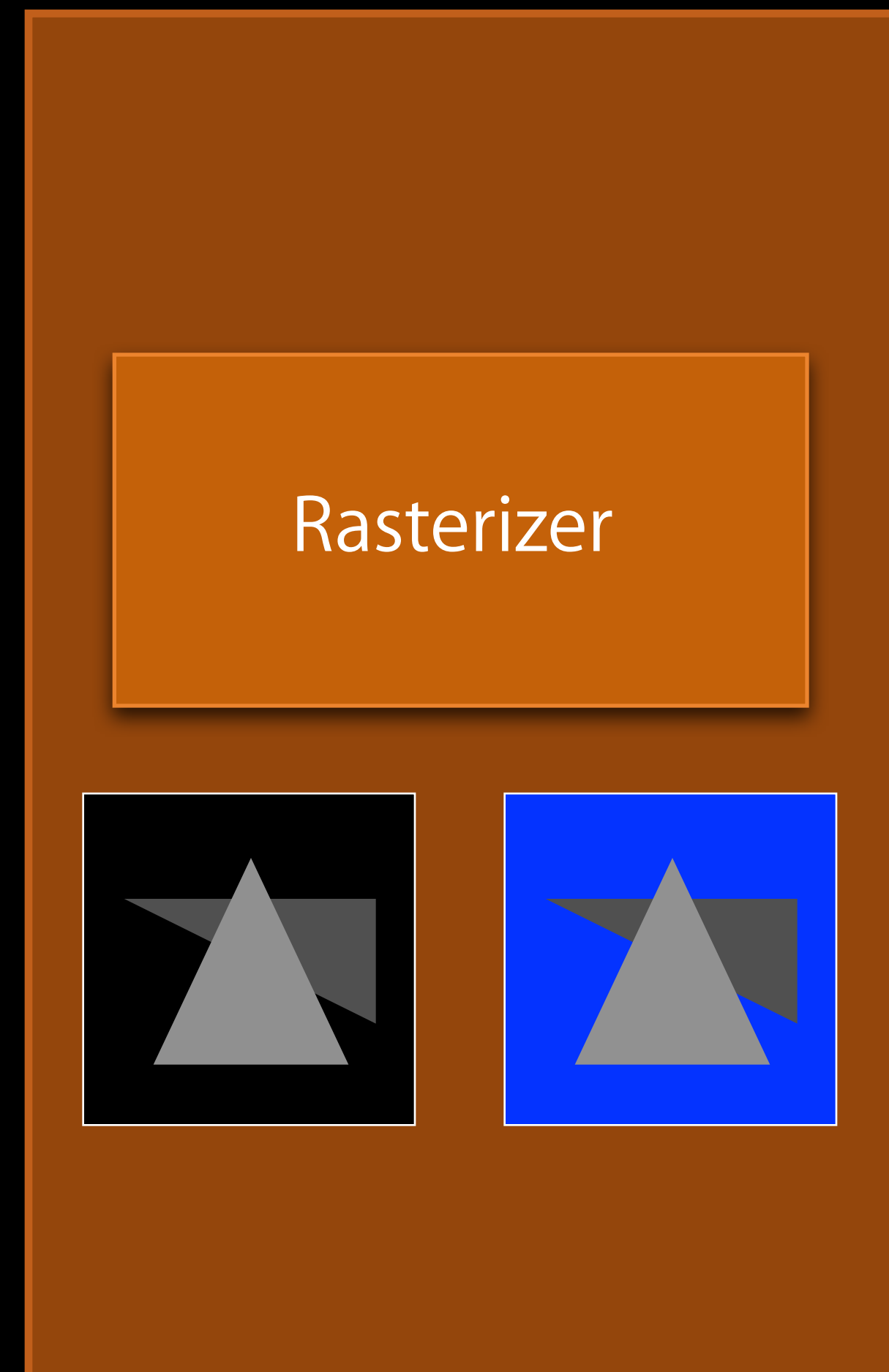


- Triangles of **entire** frame are present
 - Allows **many** fragments to be rejected

Unified Memory



GPU



- Triangles of **entire** frame are present
 - Allows **many** fragments to be rejected

Unified Memory

GPU

⚠ Loss Of Depth Test Hardware Optimizations

The fragment shader in Program #1 disabled depth test hardware optimizations on the GPU. Your application used the following feature: `discard()` in a fragment shader. If possible, rework your rendering pipeline to avoid using this feature. If you must draw using this feature,

V₁₃ V₁₄ V₁₅

Rasterizer

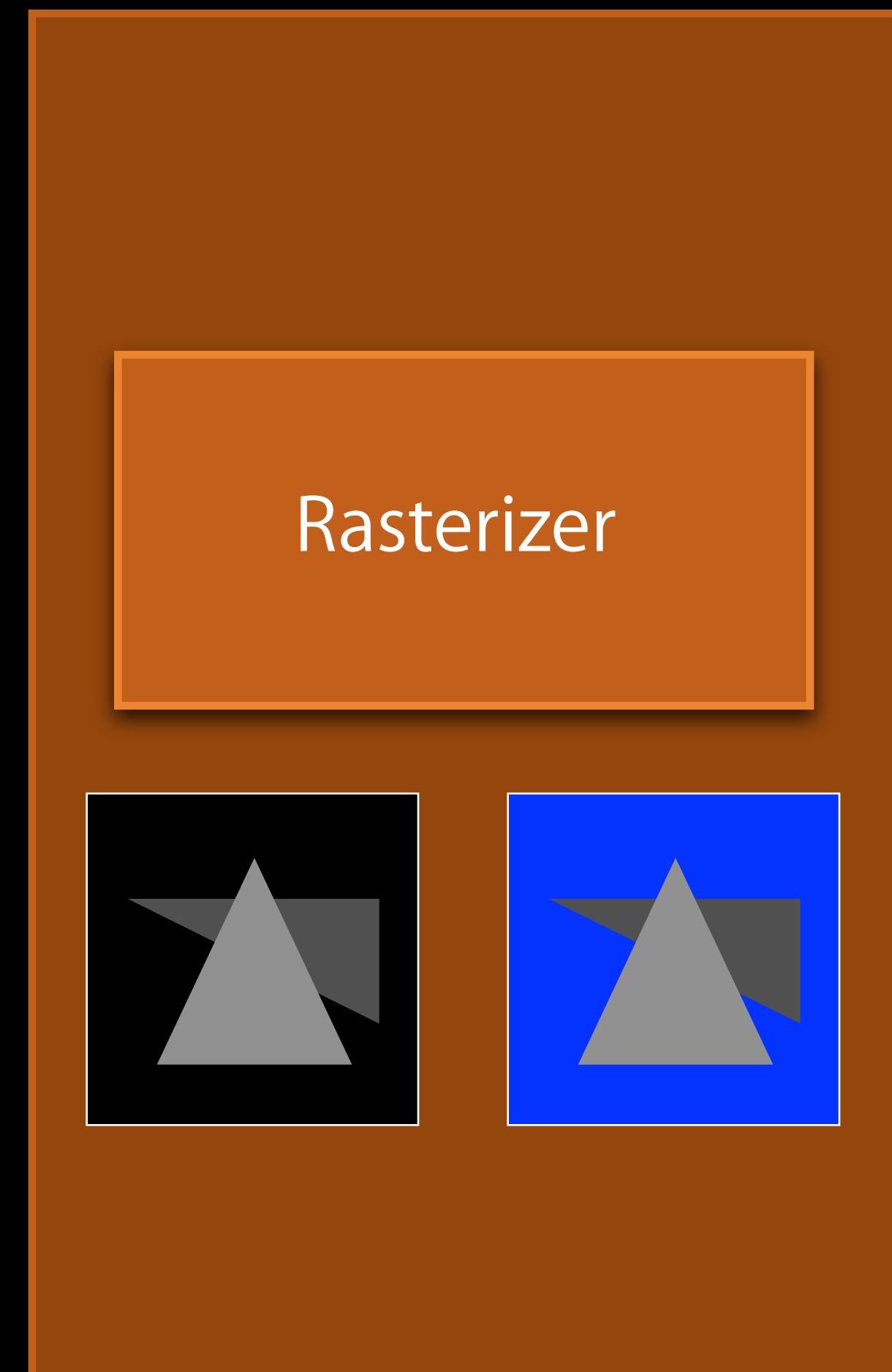


- Costly to enable blending or use discard in shader
 - Defeats the Hidden Surface Removal optimization

Unified Memory



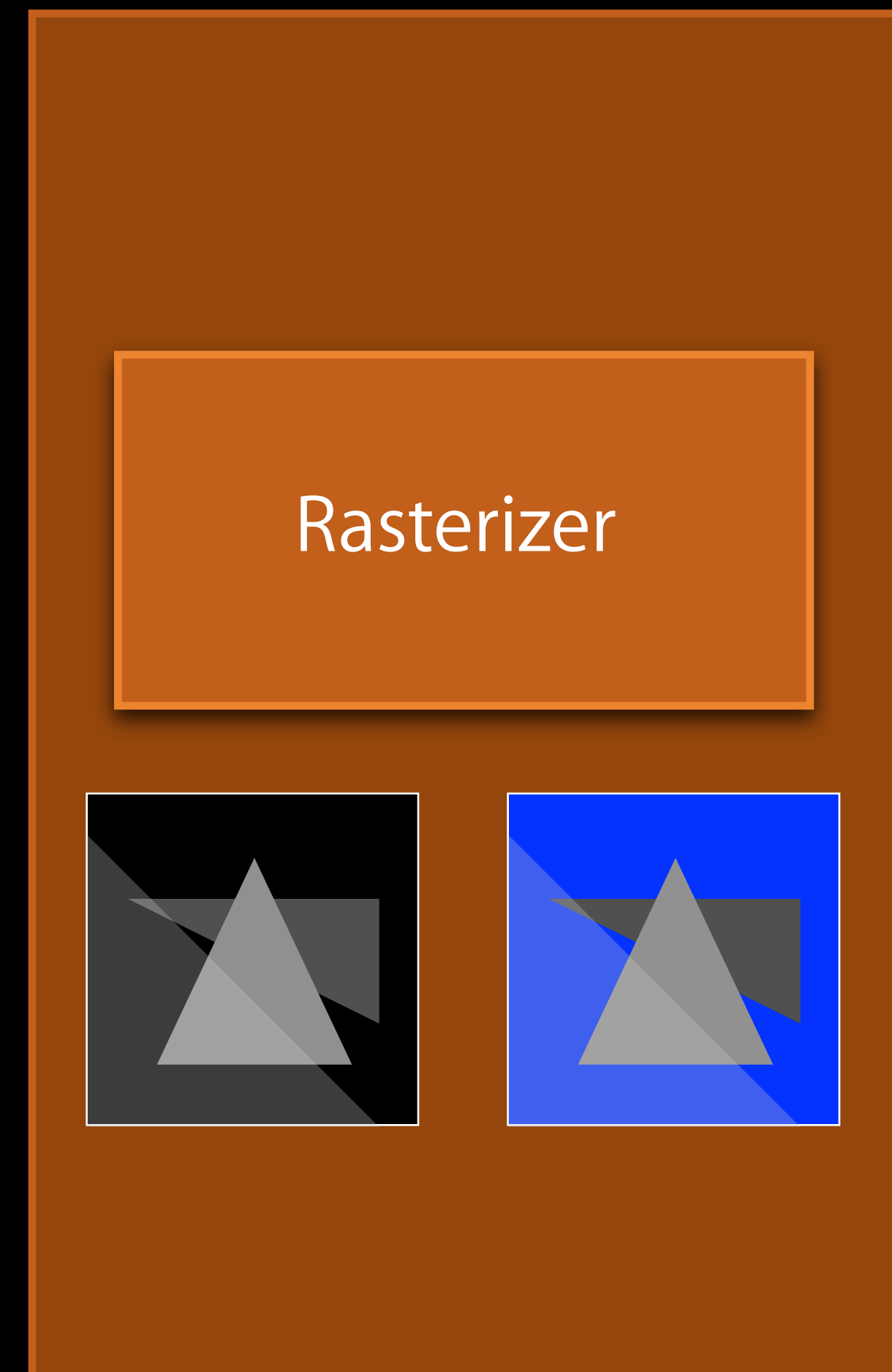
GPU



- Costly to enable blending or use discard in shader
 - Defeats the Hidden Surface Removal optimization

Unified Memory

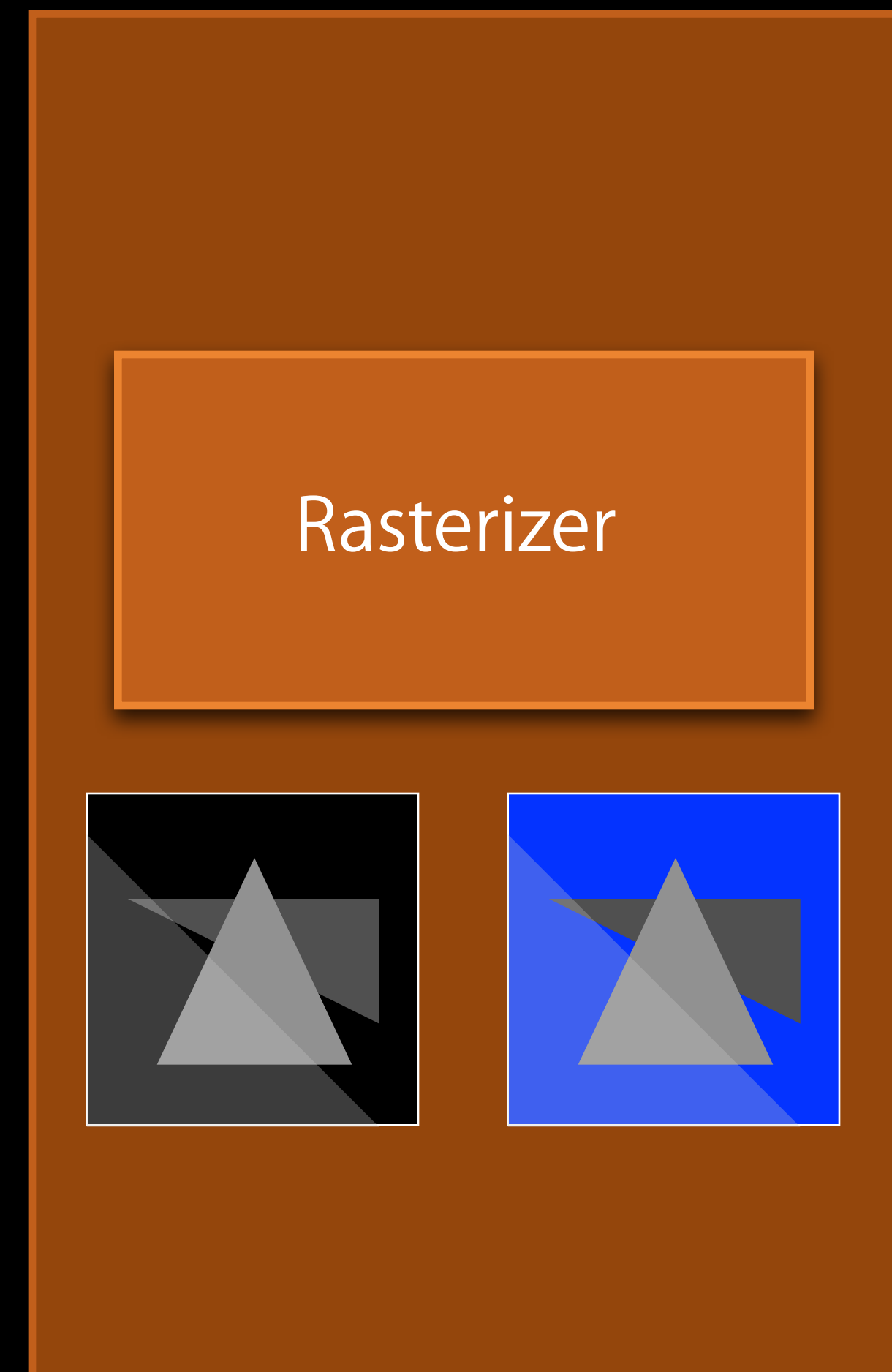
GPU



- Fragments behind other triangle could be visible
 - Shader must run for **all** triangles

Unified Memory

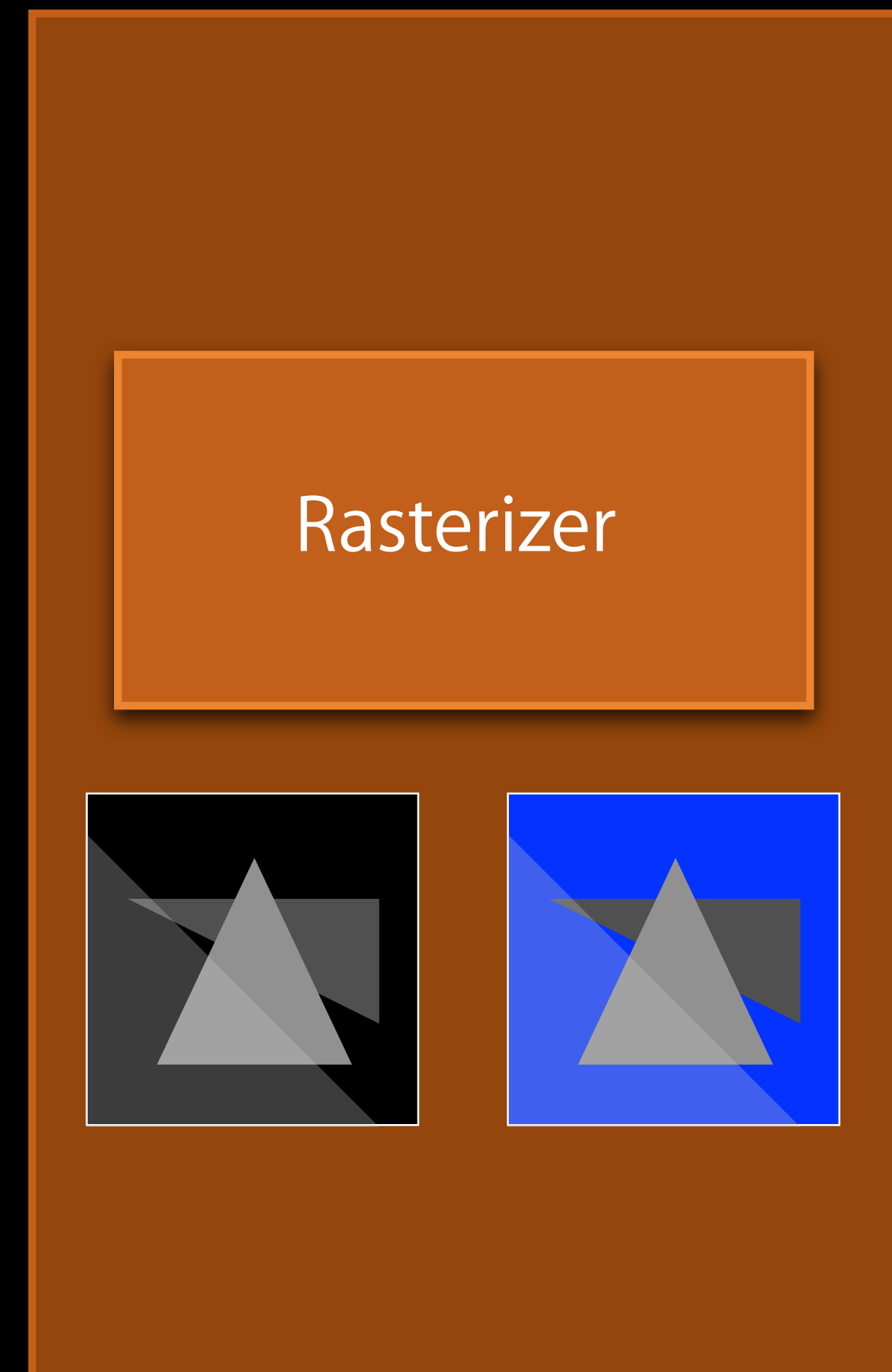
GPU



- No fragment shader savings without Hidden Surface Removal
 - Cost to performance and power

Unified Memory

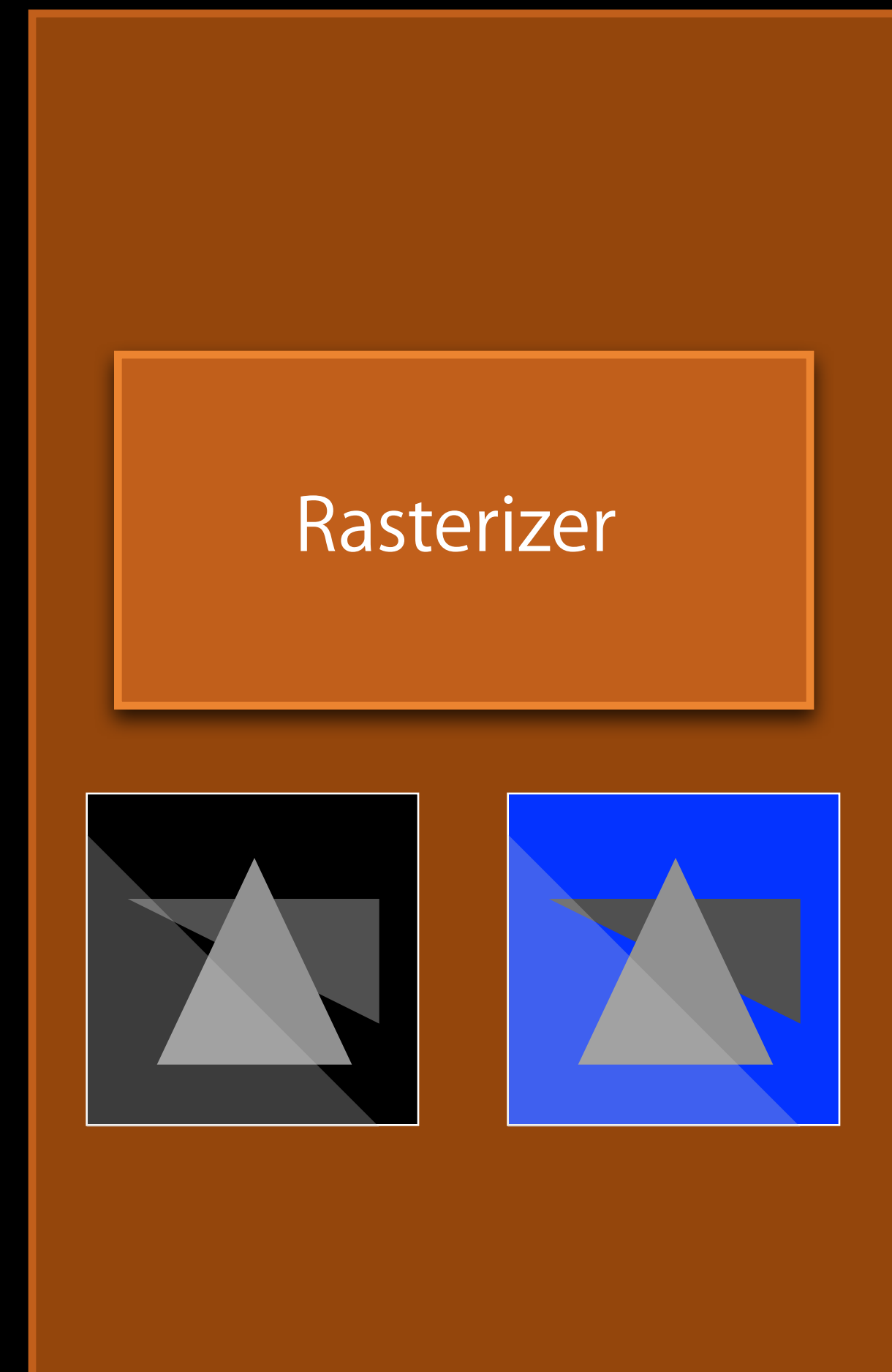
GPU



- Developers must be judicious of their use of discard and blending
 - Allow GPU to reject as many fragments as possible

Unified Memory

GPU

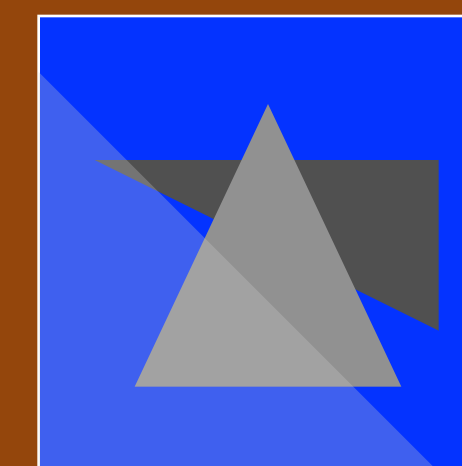
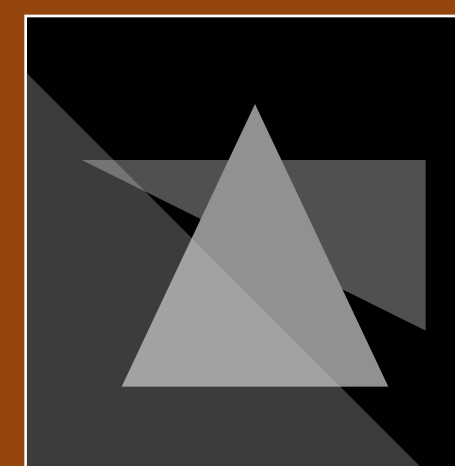


Fragment Shading

Unified Memory

GPU

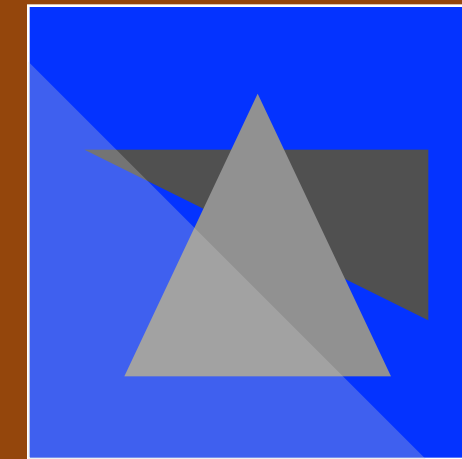
Rasterizer



Fragment Shading

Unified Memory

GPU

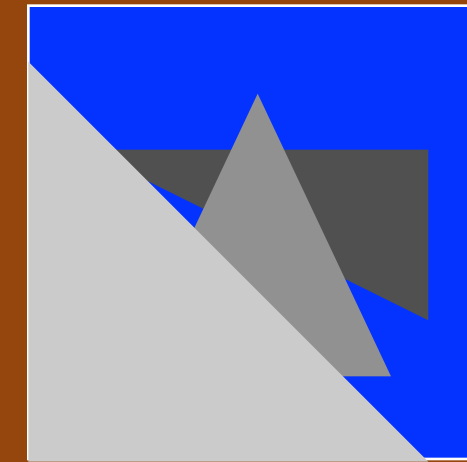


Fragment
Processor

- With Hidden Surface Removal algorithm
 - Only need to run fragment shader on each pixel once

Unified Memory

GPU

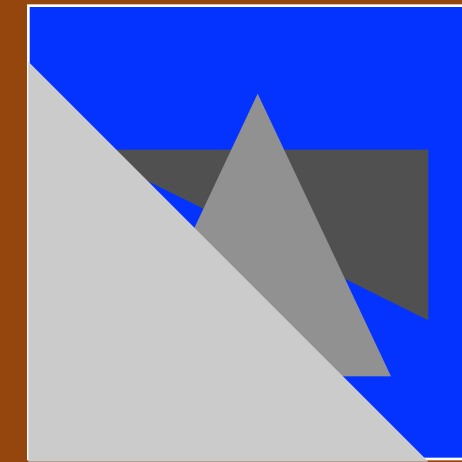


Fragment
Processor

- Fragment processor shades and produces colored pixels
- Colors written to embedded tile memory on GPU

Unified Memory

GPU



Fragment
Processor

- Fragment processor shades and produces colored pixels
- Colors written to embedded tile memory on GPU

Unified Memory

GPU

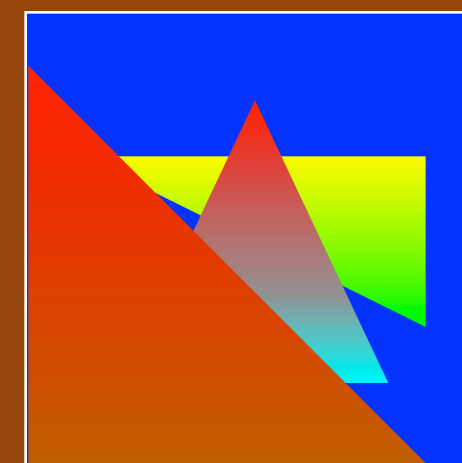


Tile Storage

Unified Memory

GPU

Fragment
Processor

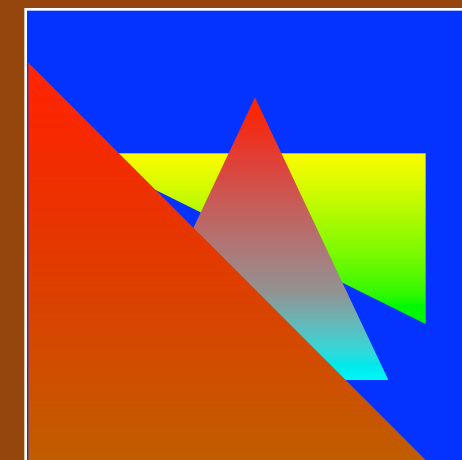


Tile Storage

Unified Memory

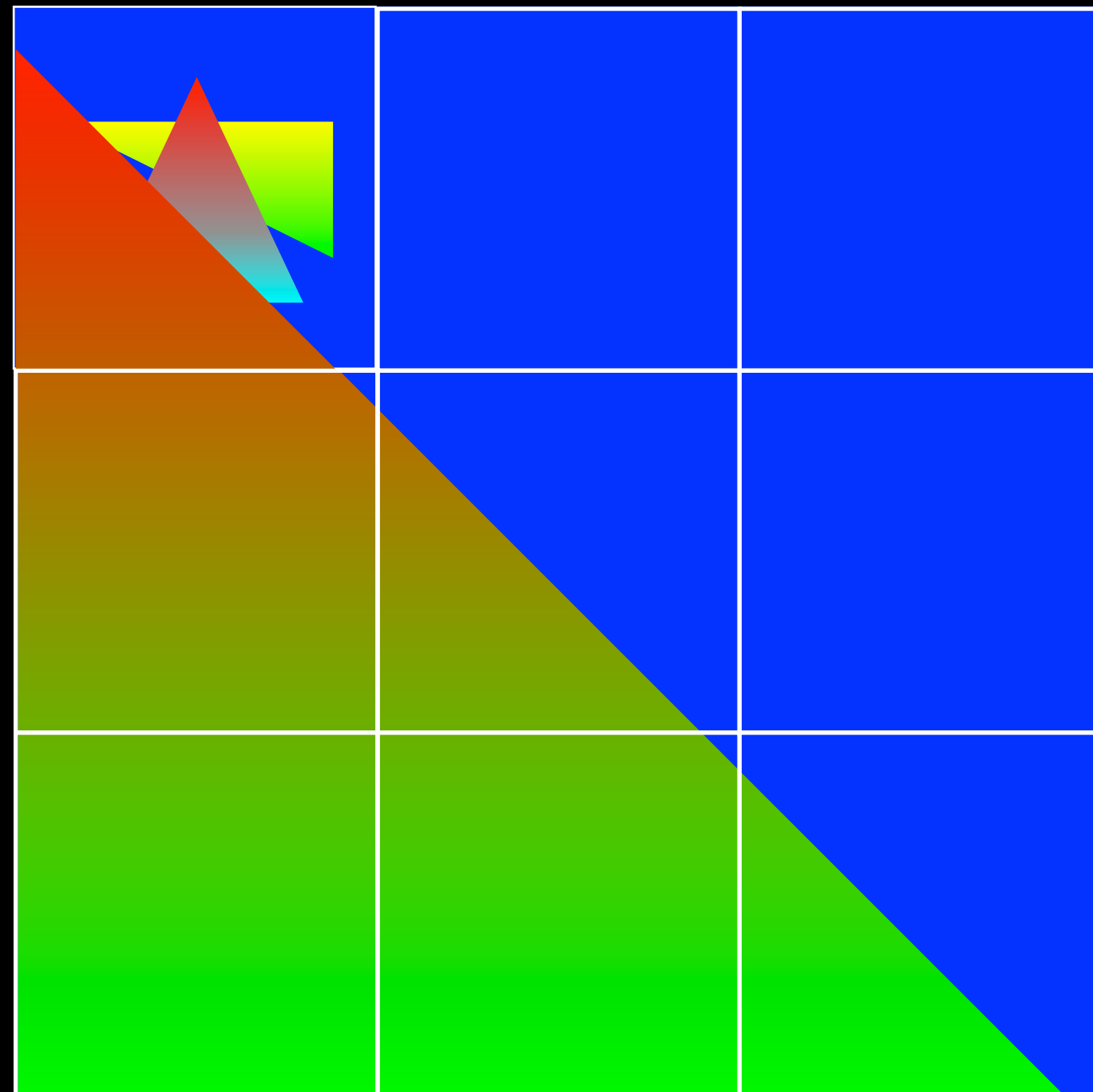
GPU

Tile Store

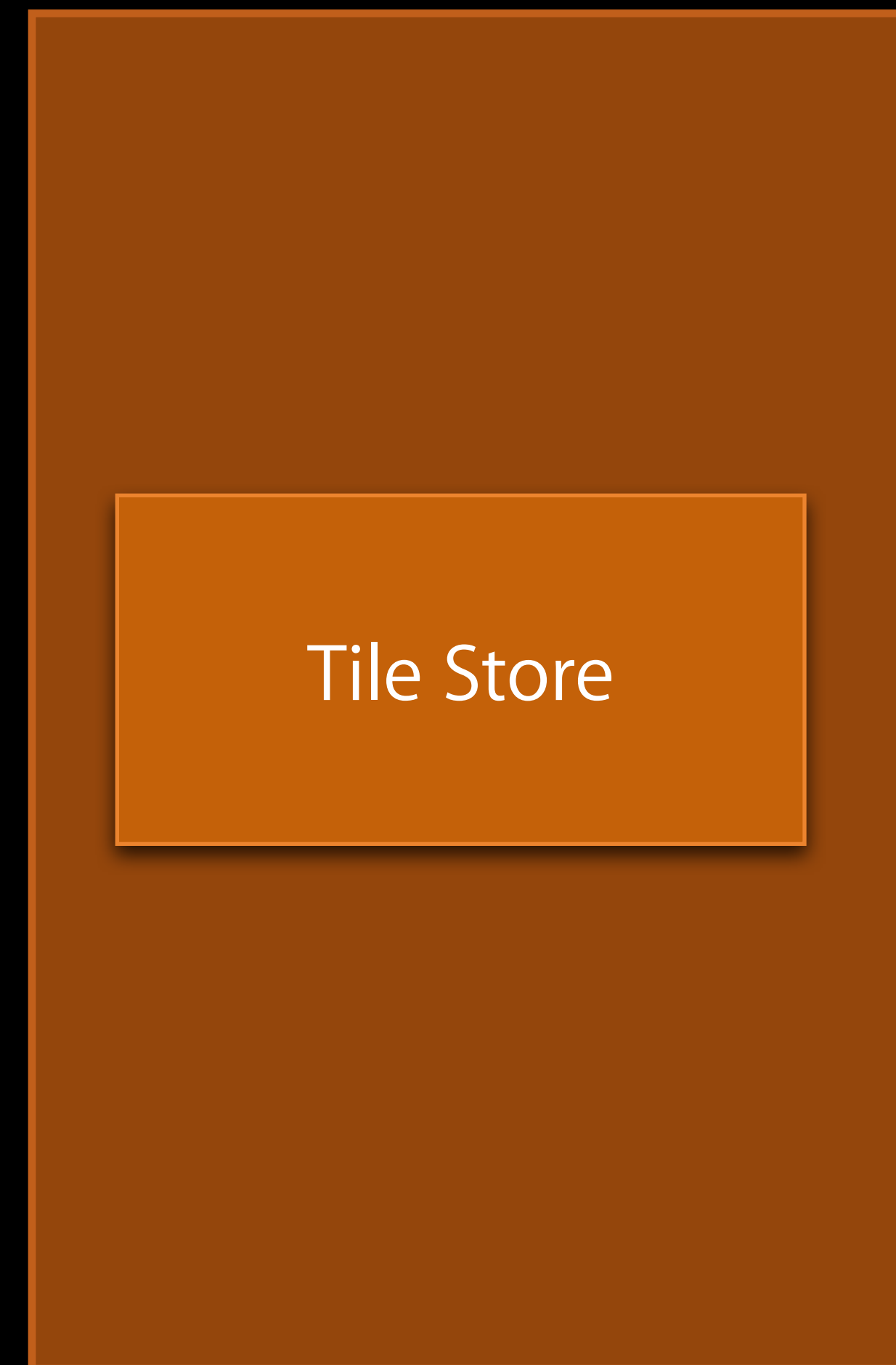


Tile Storage

Unified Memory

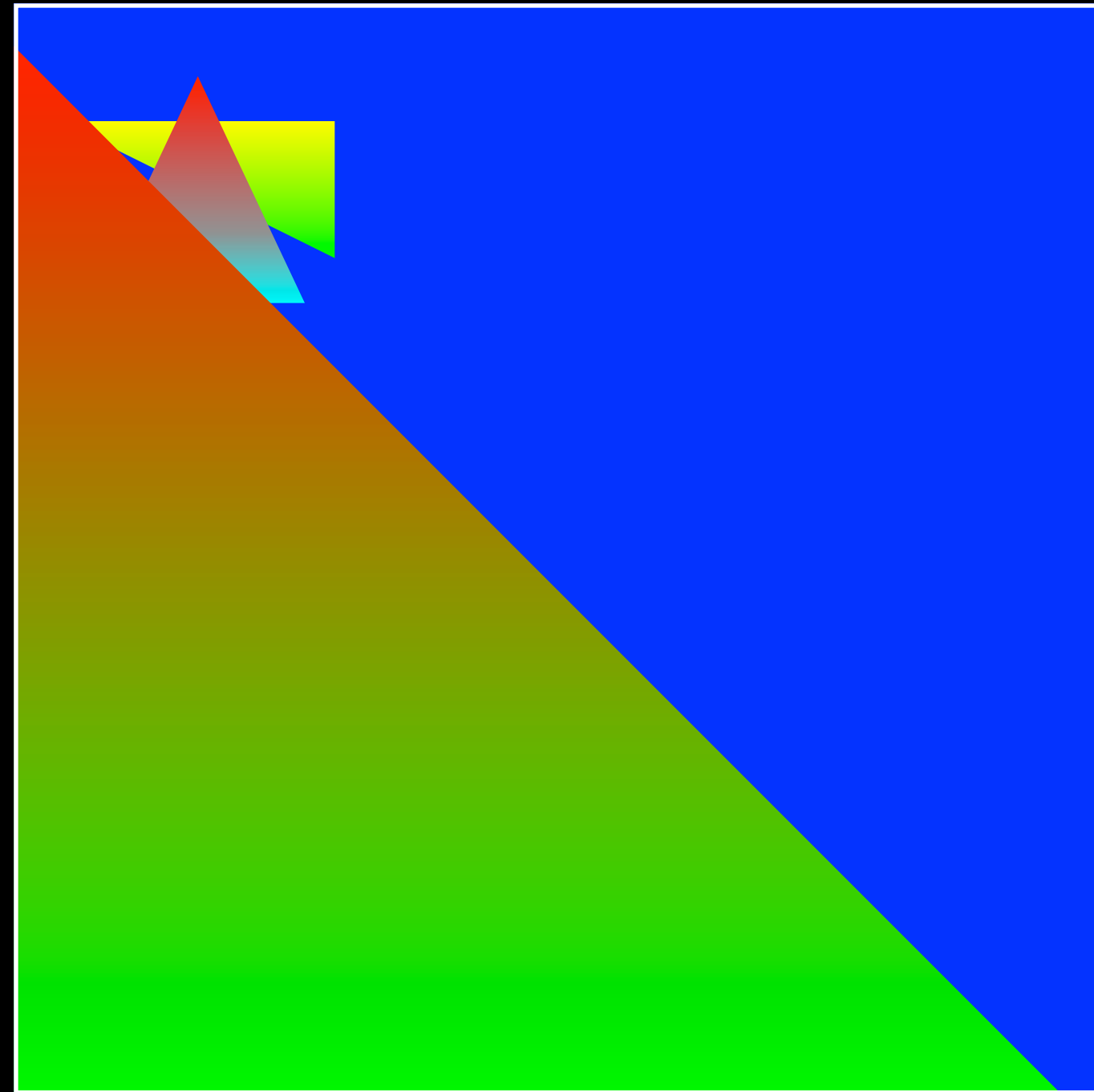


GPU

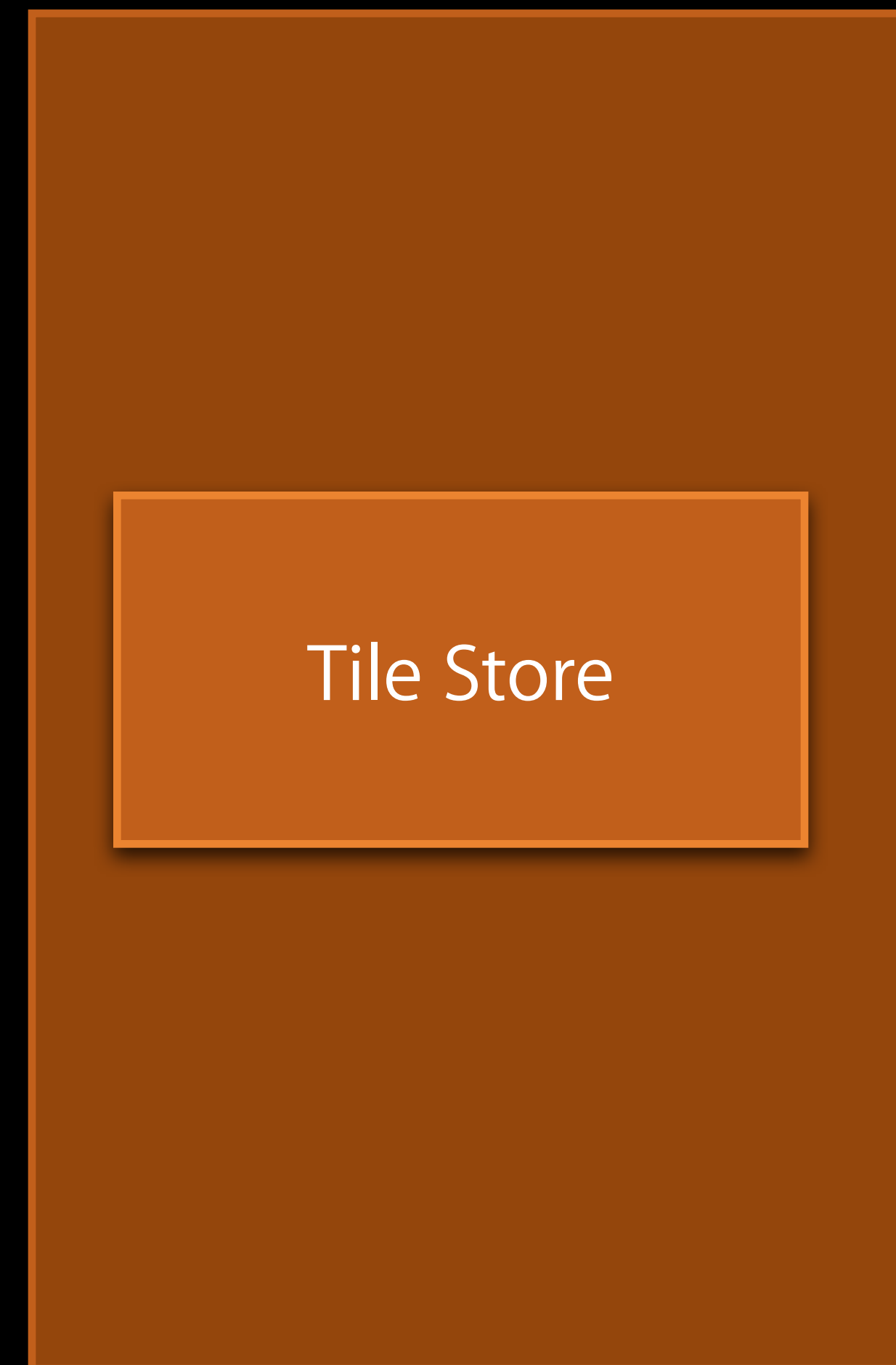


- Tile stored in unified memory
- Once all tiles processed, renderbuffer is ready for use

Unified Memory

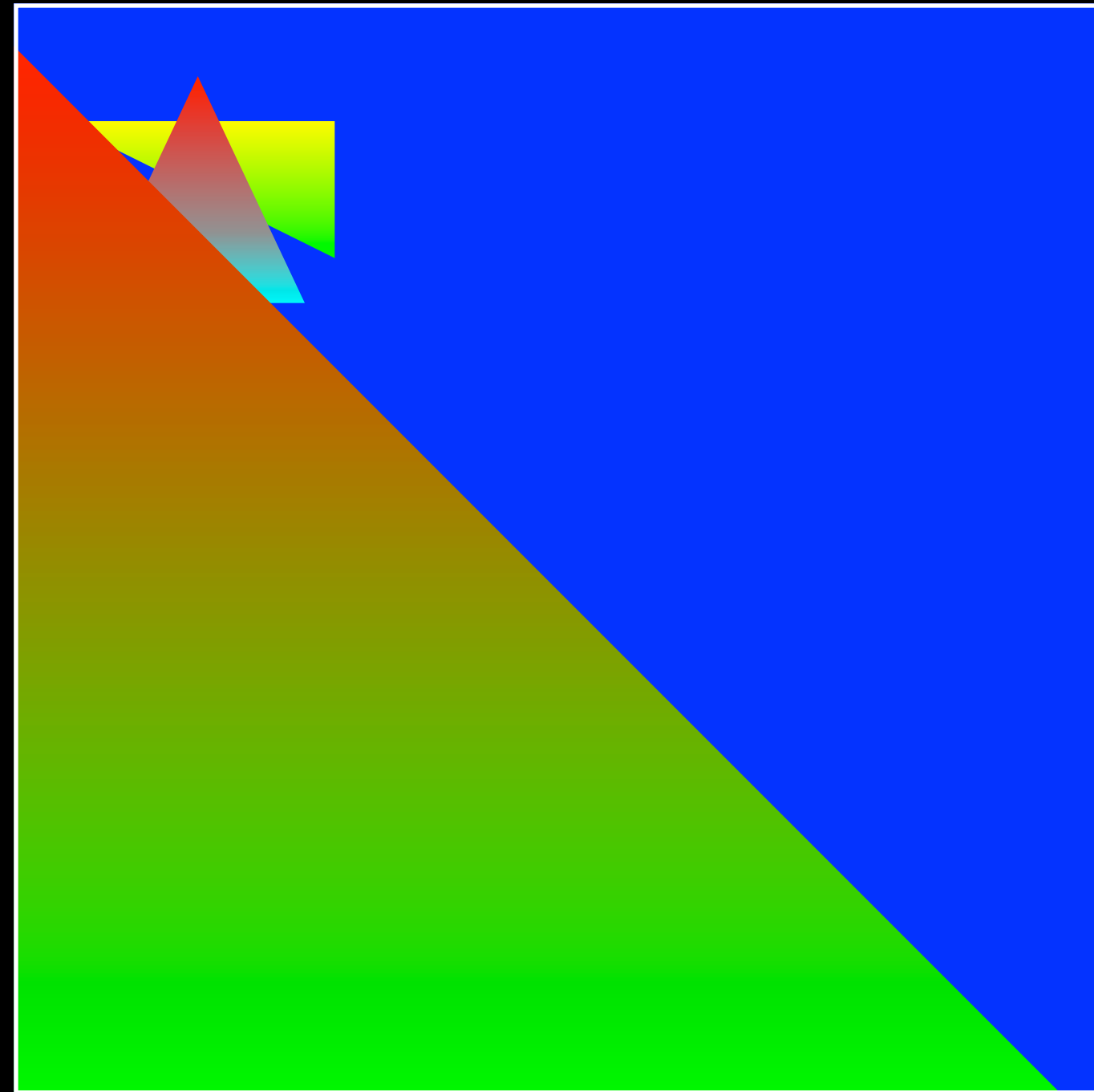


GPU

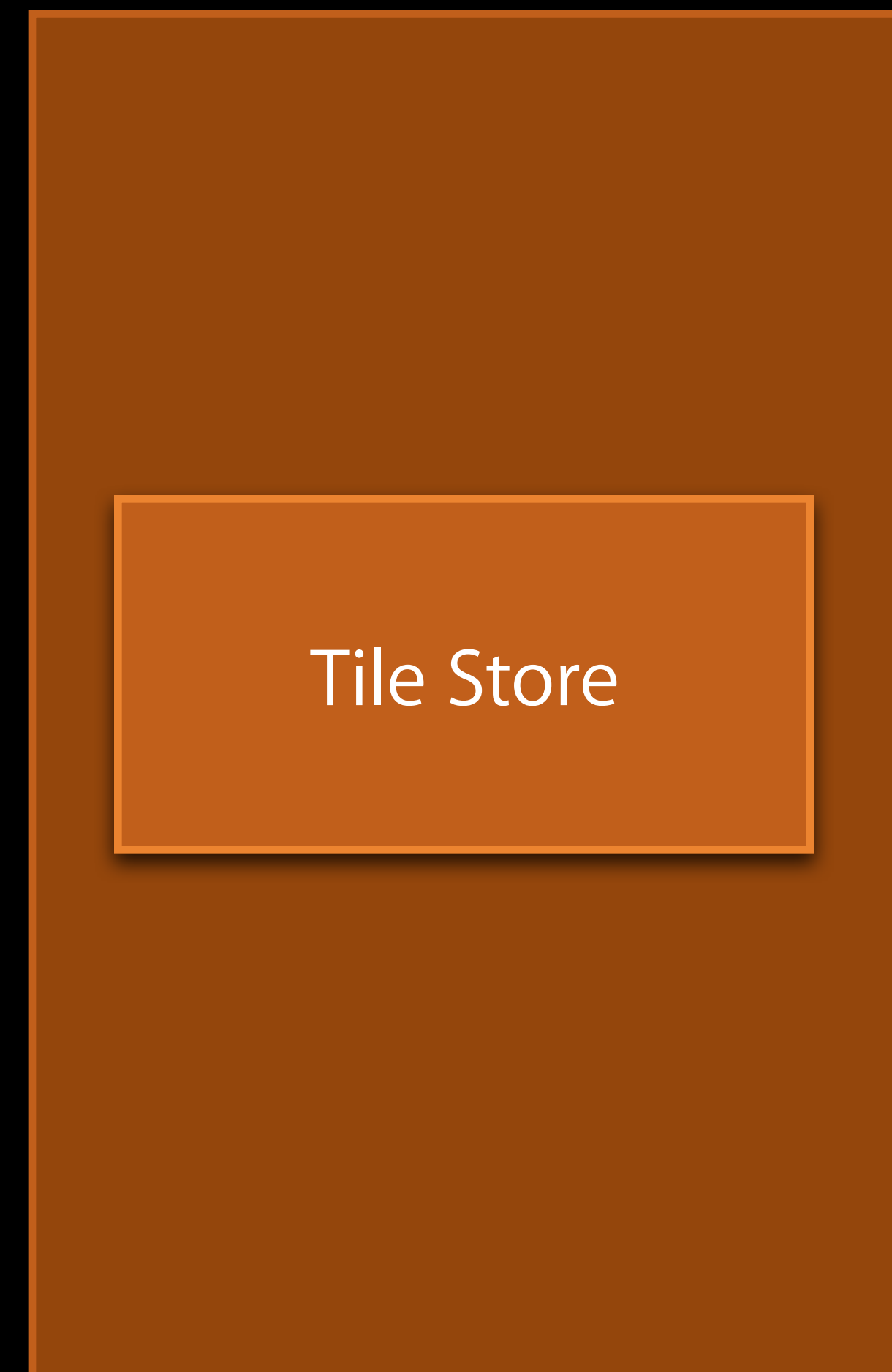


- Storing tile to unified memory called Logical Buffer Store
- Each frame needs at least one

Unified Memory

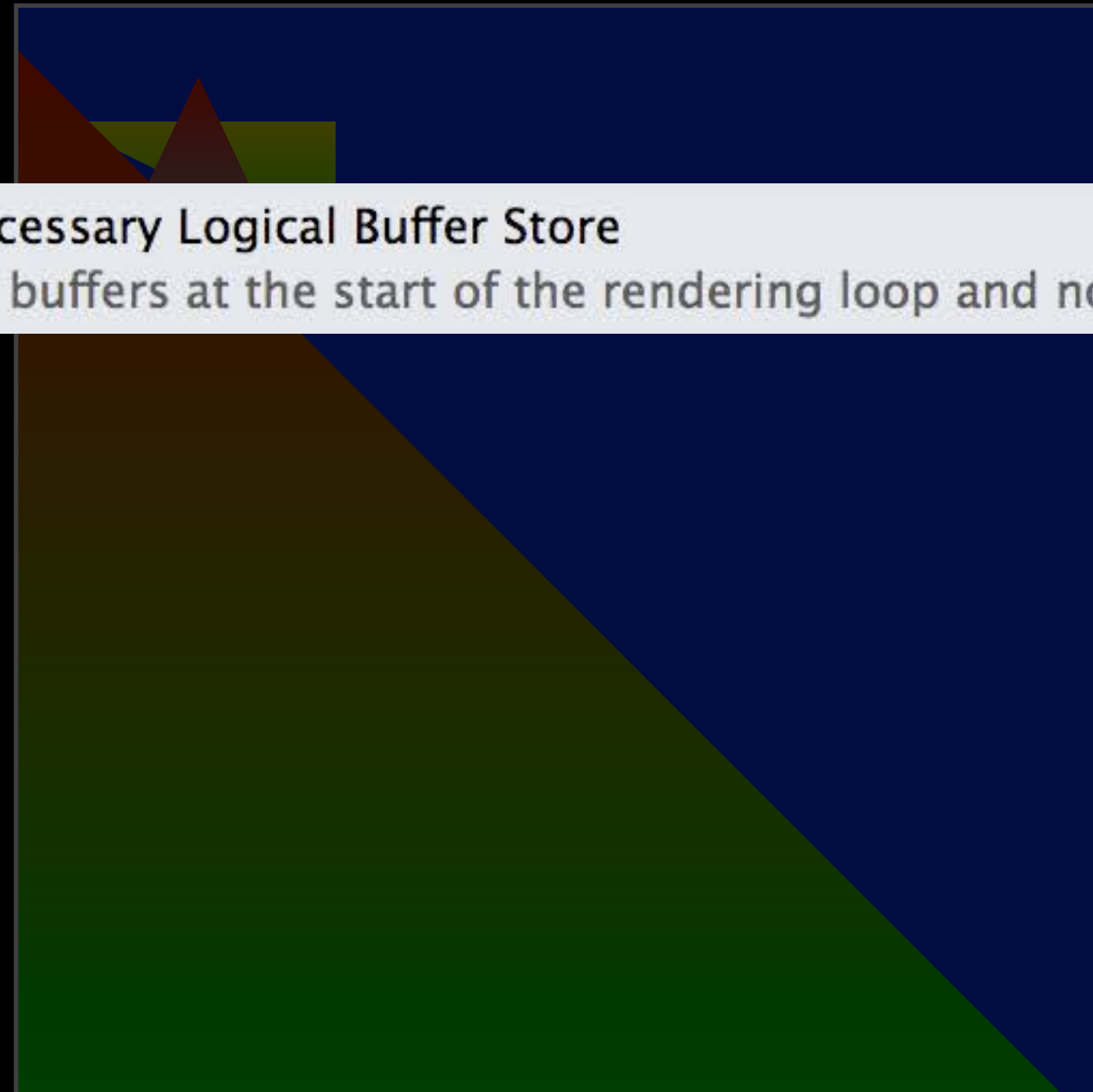



GPU



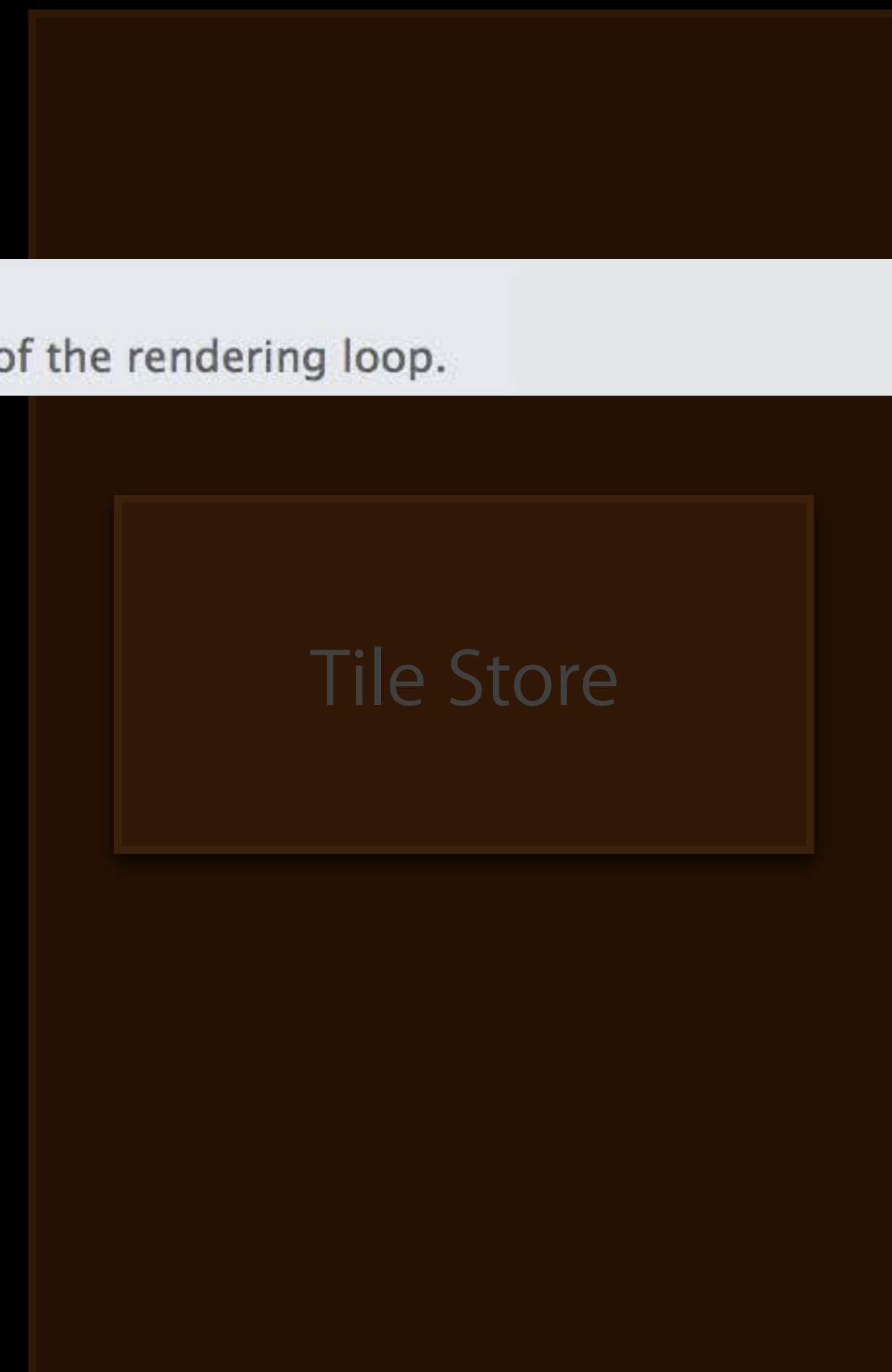
- Storing tile to unified memory called Logical Buffer Store
- Each frame needs at least one

Unified Memory



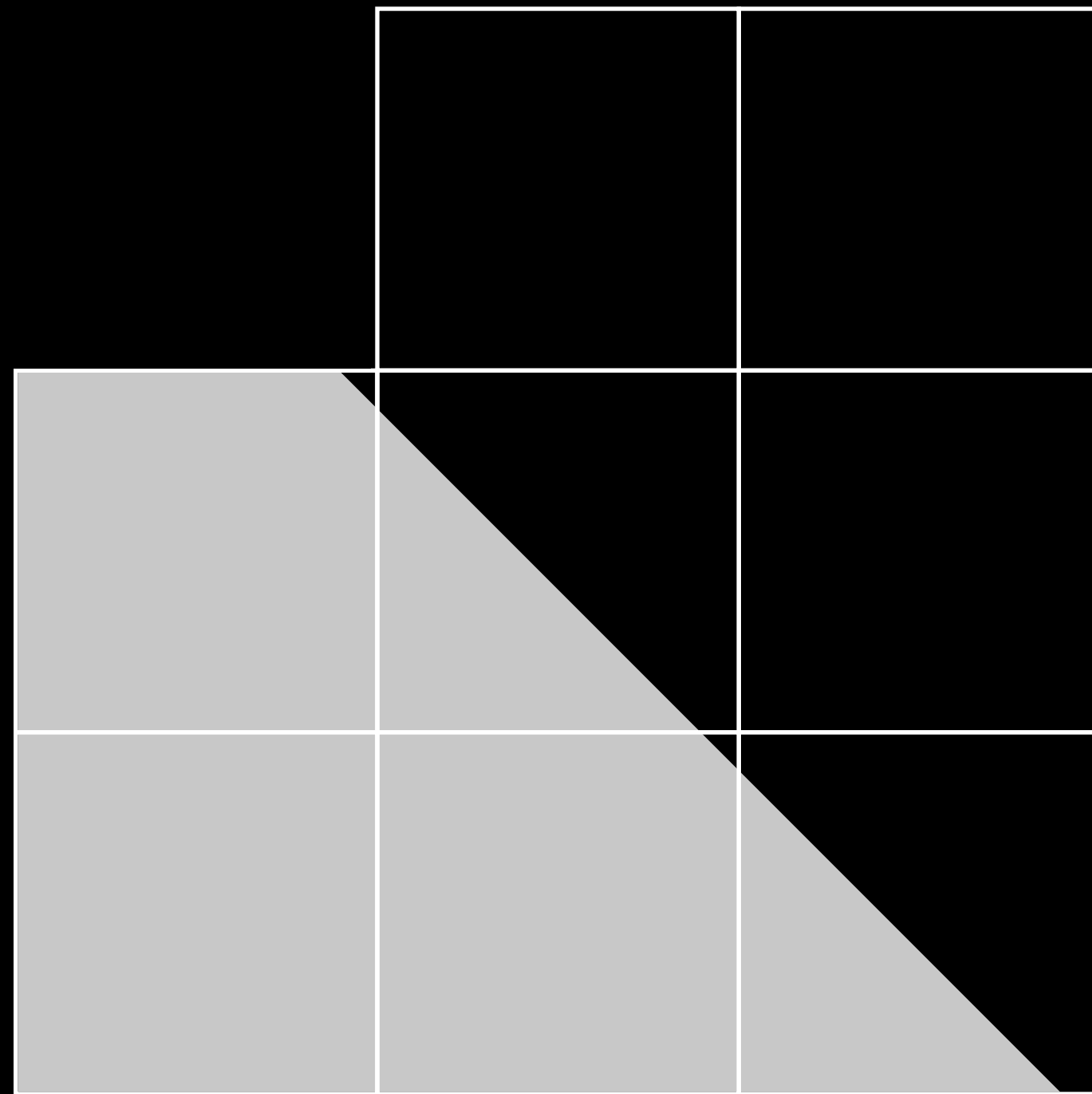
 **Unnecessary Logical Buffer Store**
Clear buffers at the start of the rendering loop and not discard them at the end of the rendering loop.

GPU

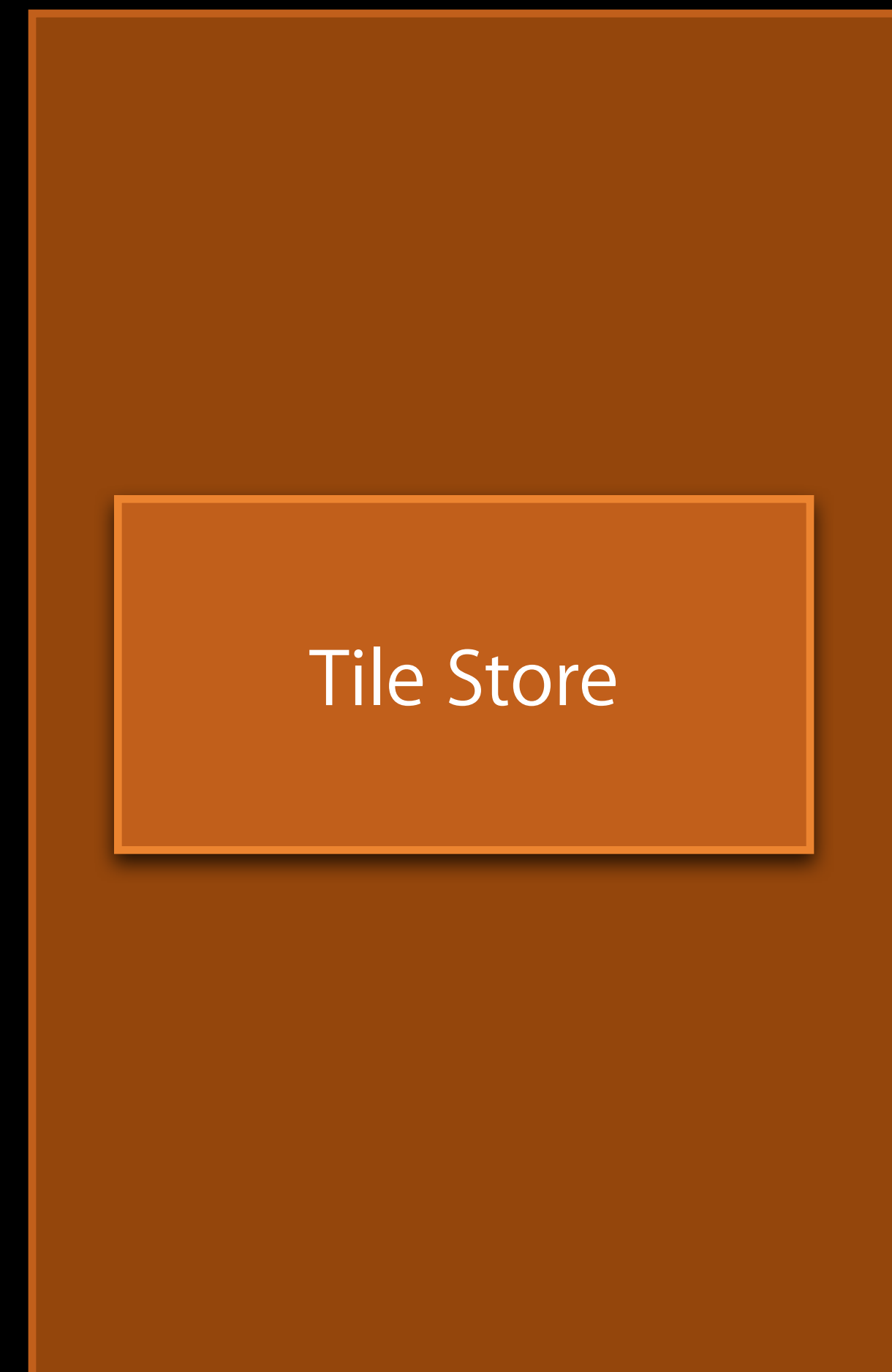


- Depth only needs to be stored for depth texture effects
 - Such as shadowing or Screen Space Ambient Occlusion

Unified Memory

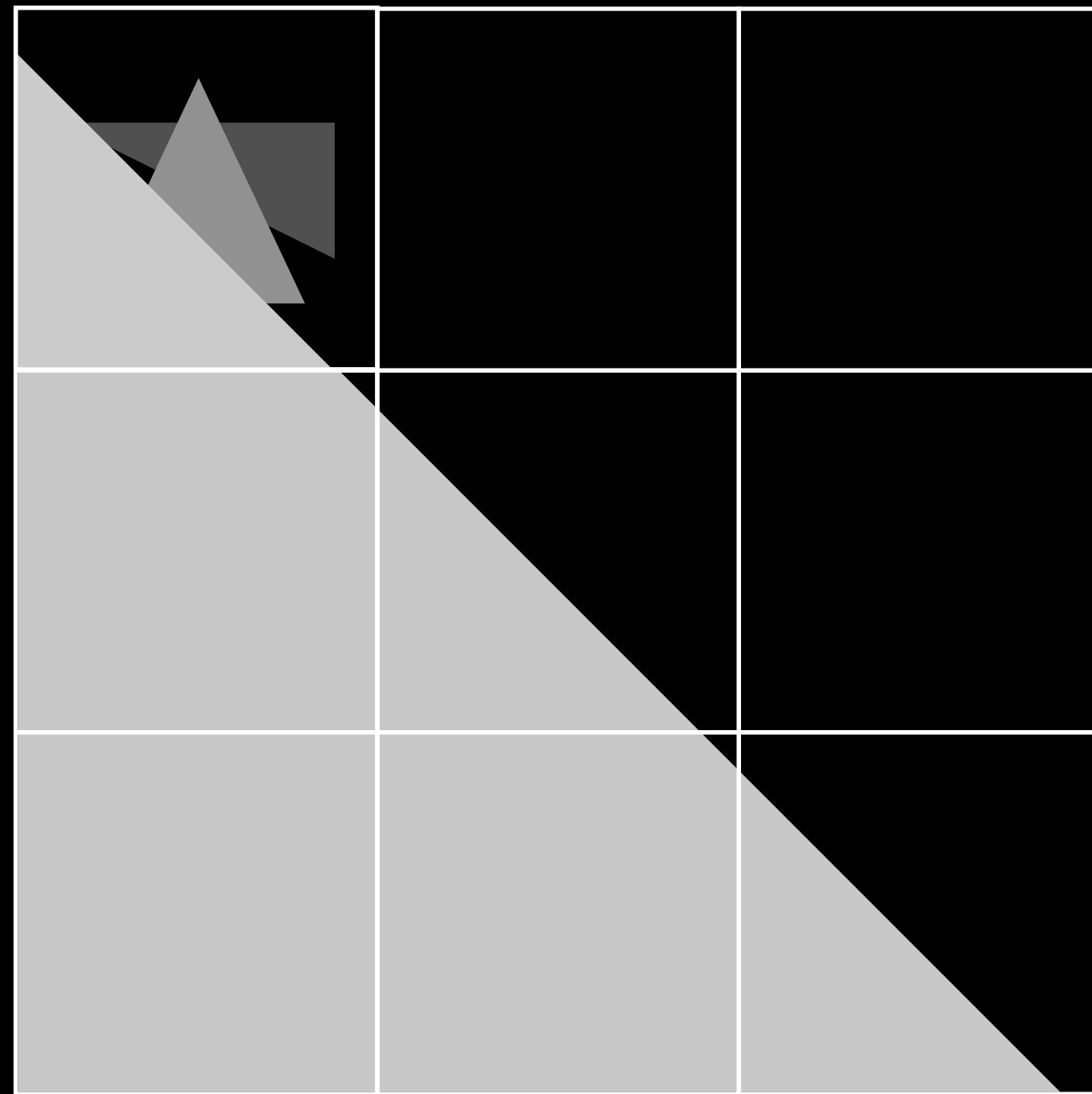


GPU

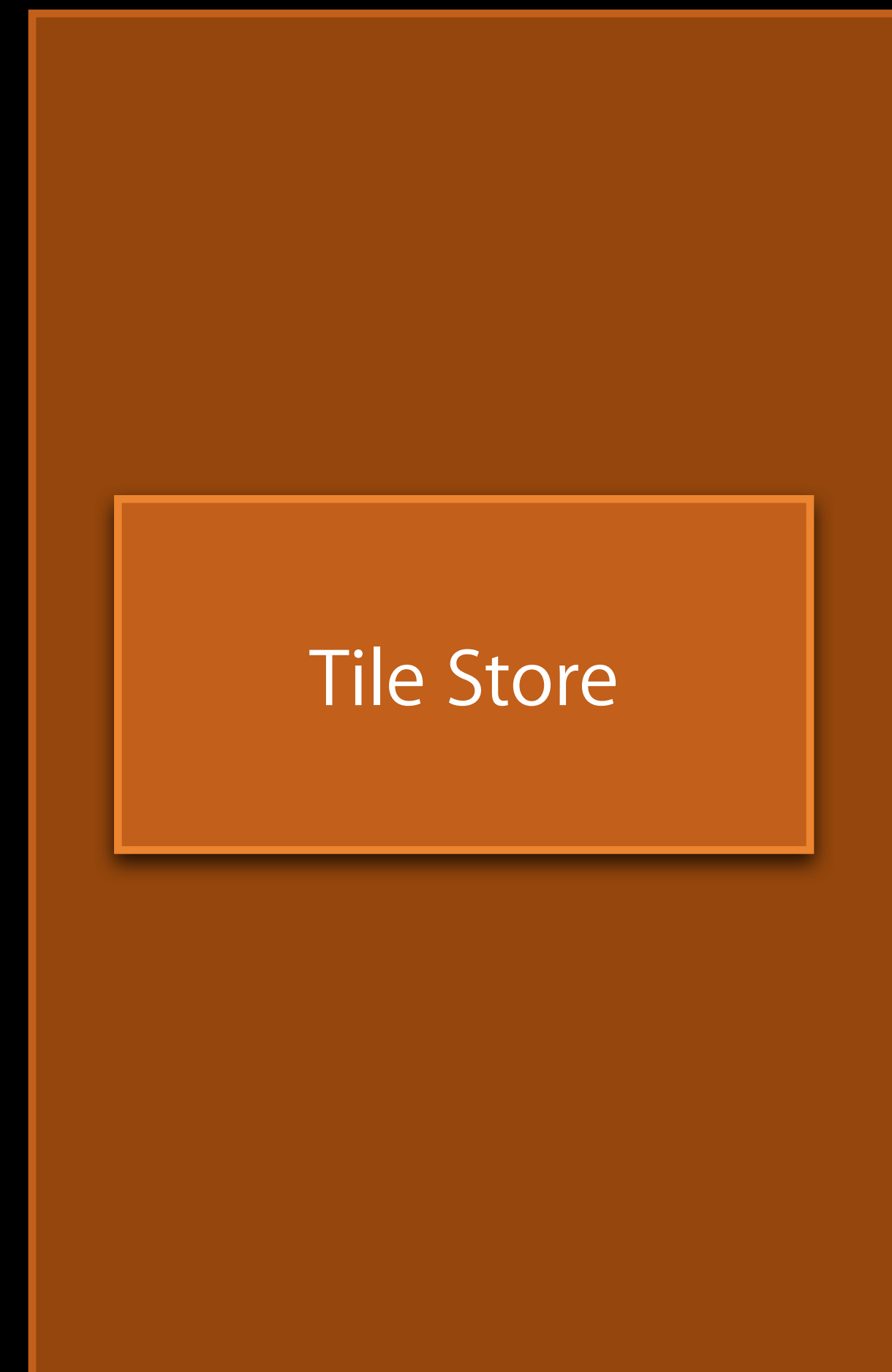


- Depth only needs to be stored for depth texture effects
 - Such as shadowing or Screen Space Ambient Occlusion

Unified Memory

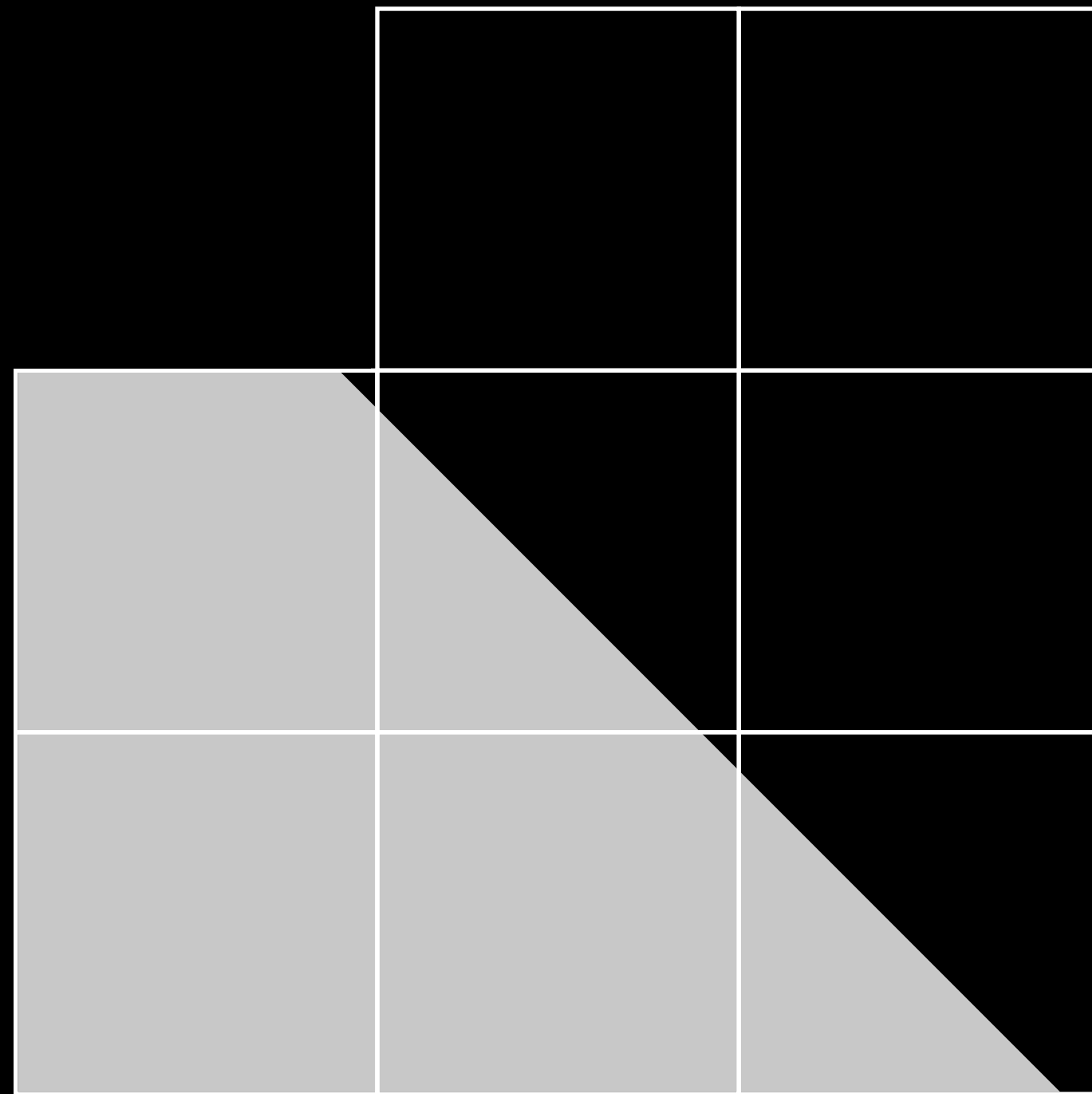


GPU

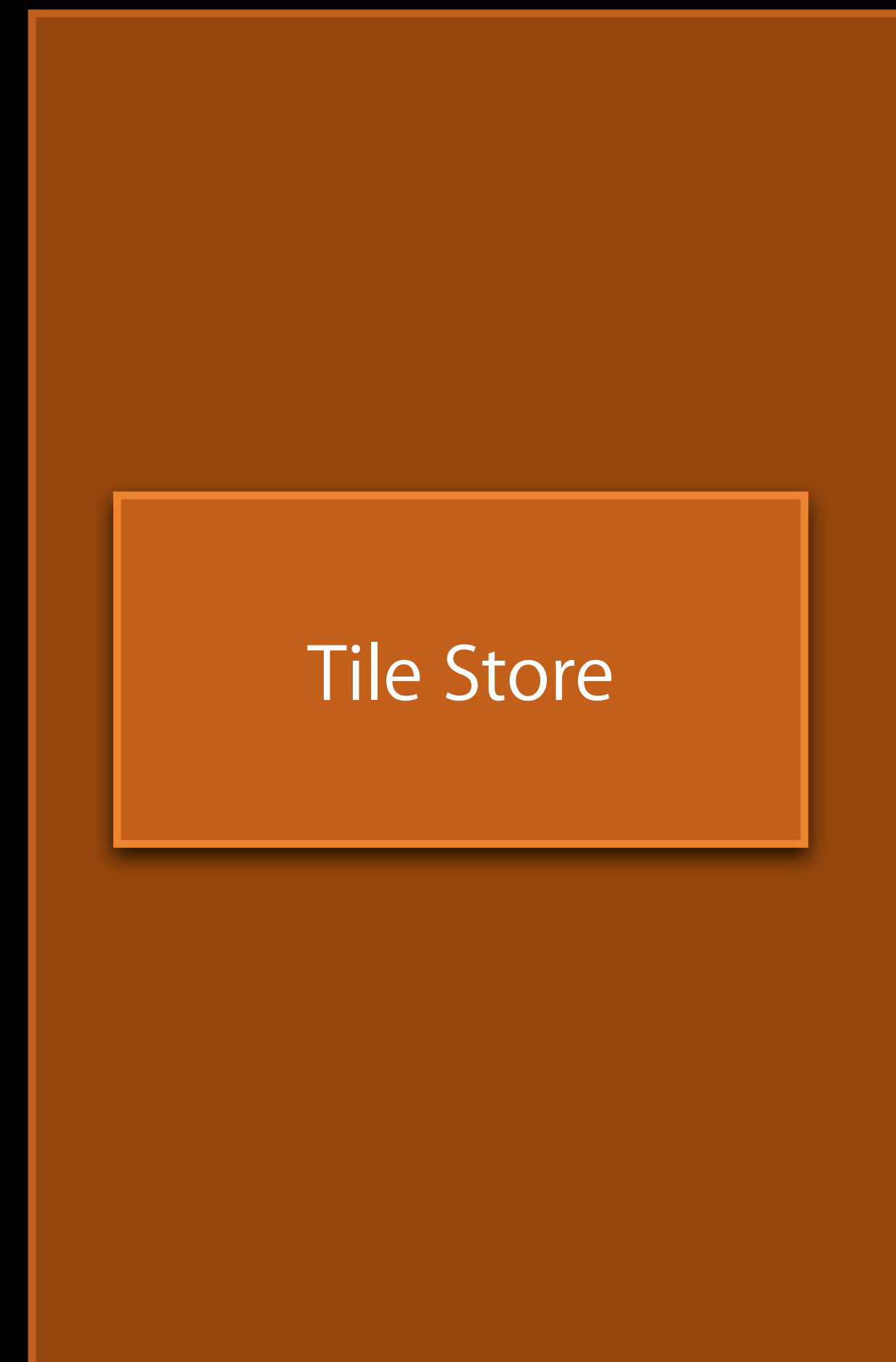


- Unless using such effect
 - Depth buffer storage unnecessary

Unified Memory



GPU

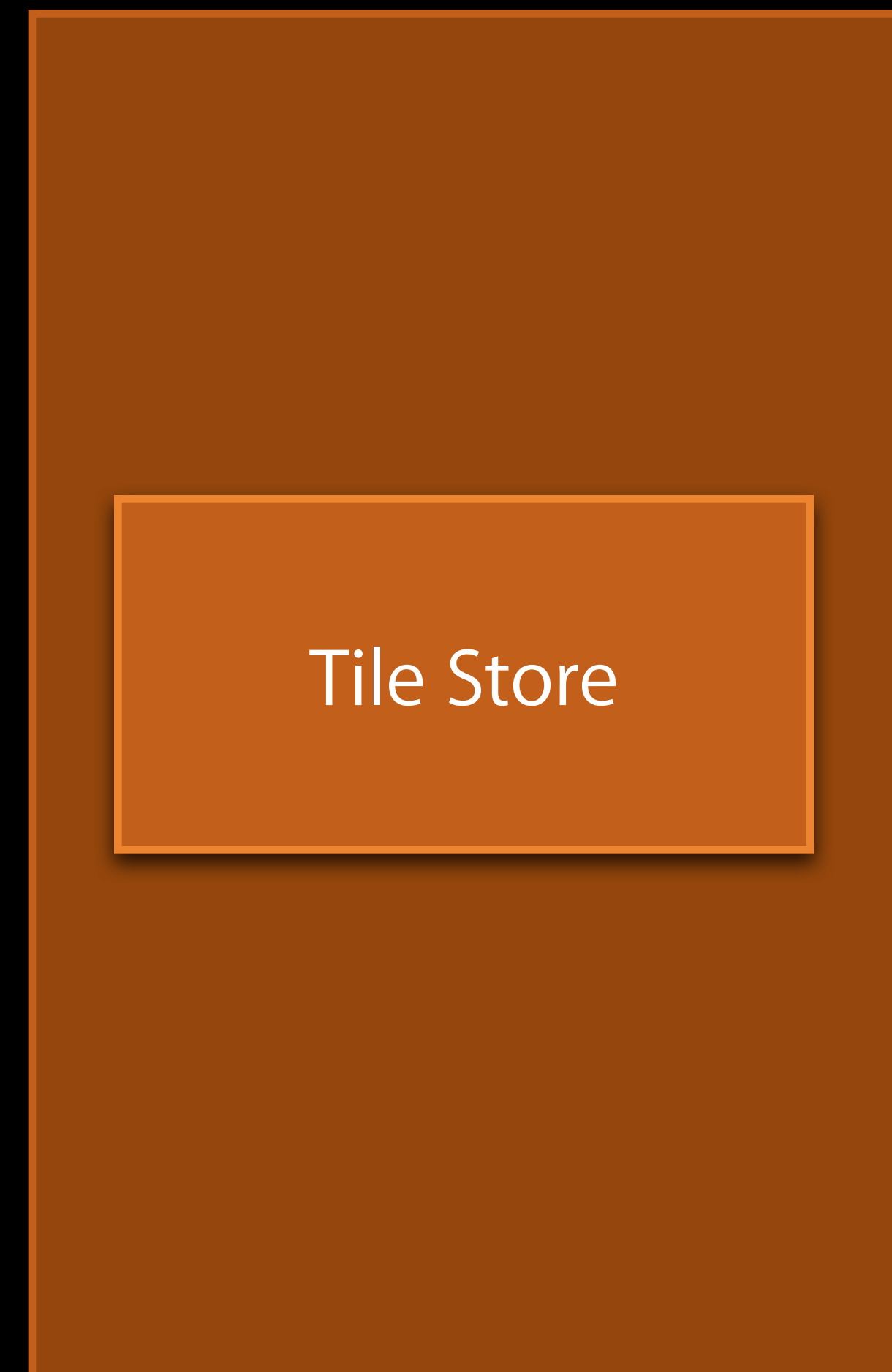


- Unless using such effect
 - Depth buffer storage unnecessary

Unified Memory



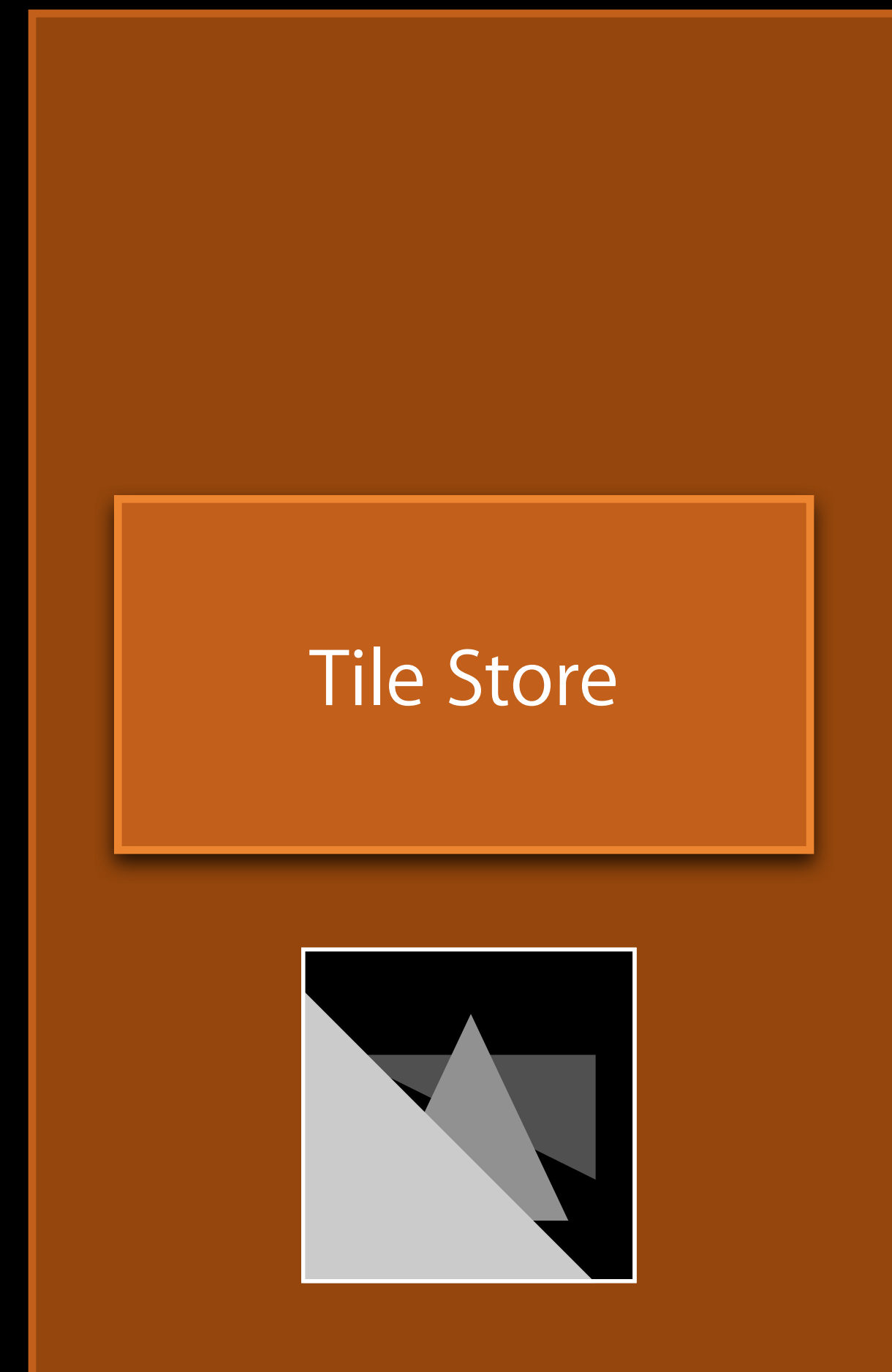
GPU



- Developers can call `glDiscardFramebufferEXT` to skip Logical Buffer Store
- Depth buffer tile discarded after rendering complete

Unified Memory

GPU



- Developers can call `glDiscardFramebufferEXT` to skip Logical Buffer Store
- Depth buffer tile discarded after rendering complete

Unified Memory

GPU

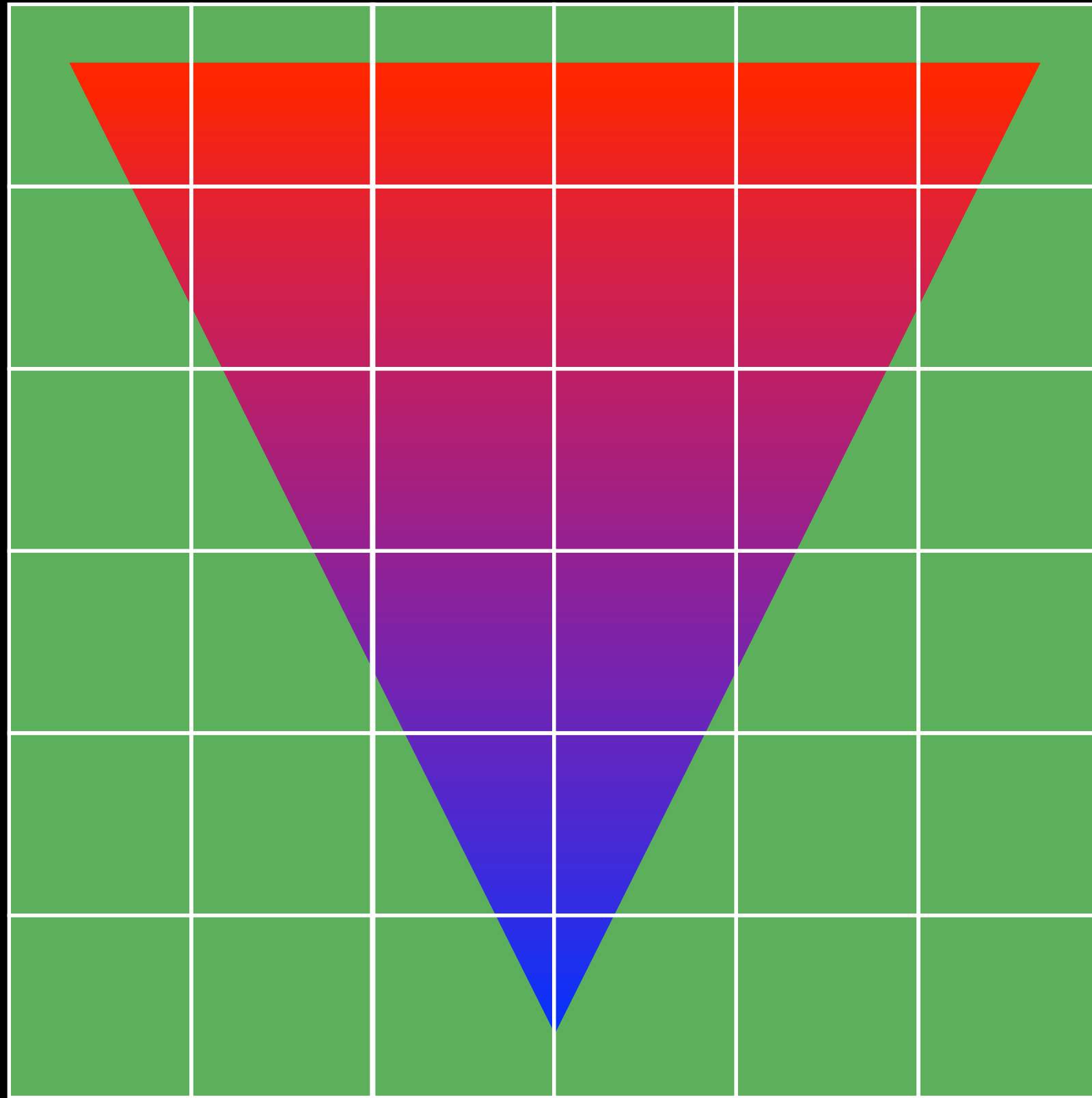


The diagram illustrates the GPU memory structure. It features a large orange rectangle representing the GPU memory. Inside this rectangle, there is a smaller orange rectangle labeled "Tile Store". The "Unified Memory" label is positioned to the left of the GPU memory area.

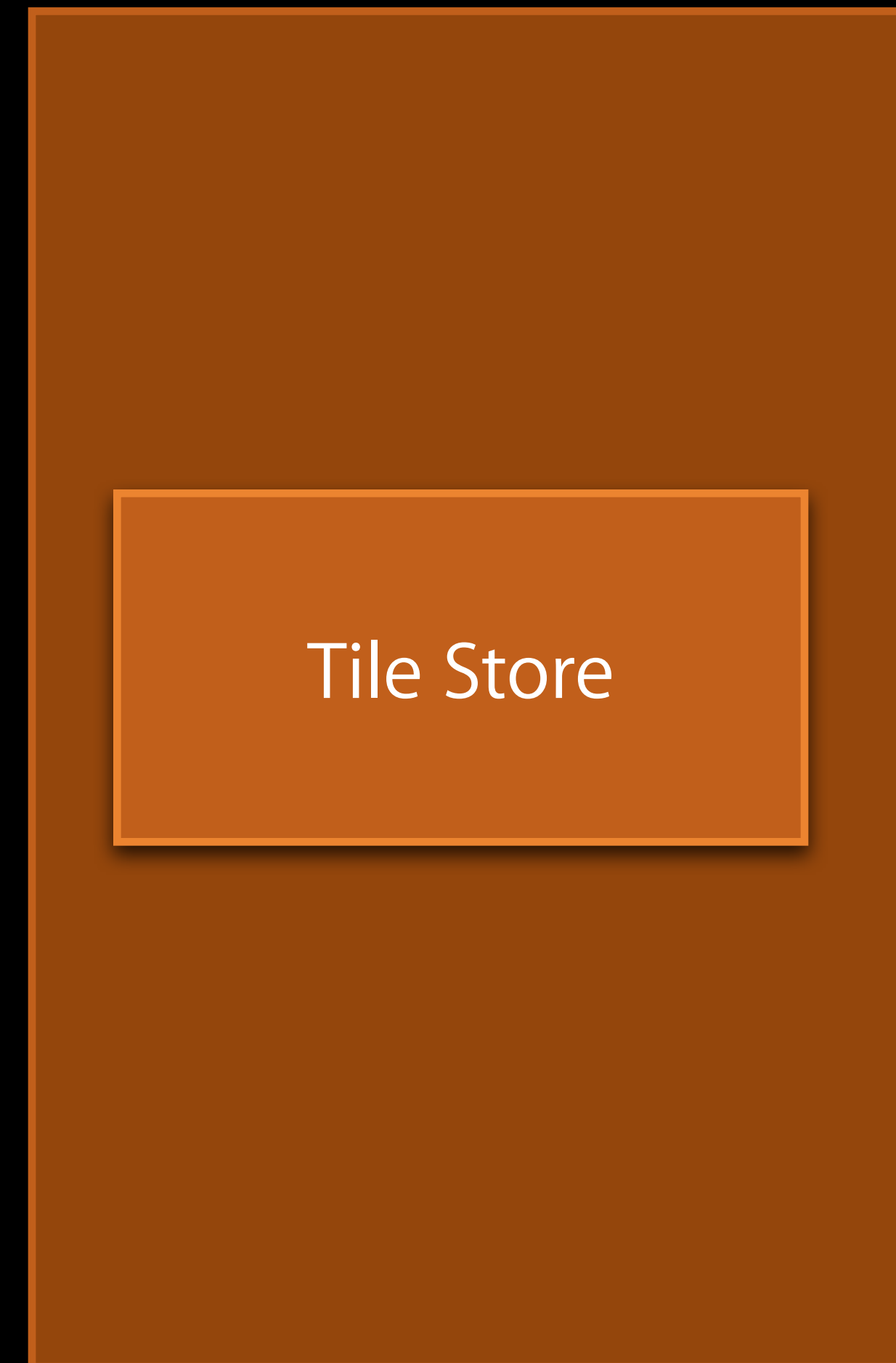
Tile Store

- Multisample anti-aliased renderbuffers have many more tiles
 - Developers do not need to store prerresolved MSAA renderbuffer

Unified Memory

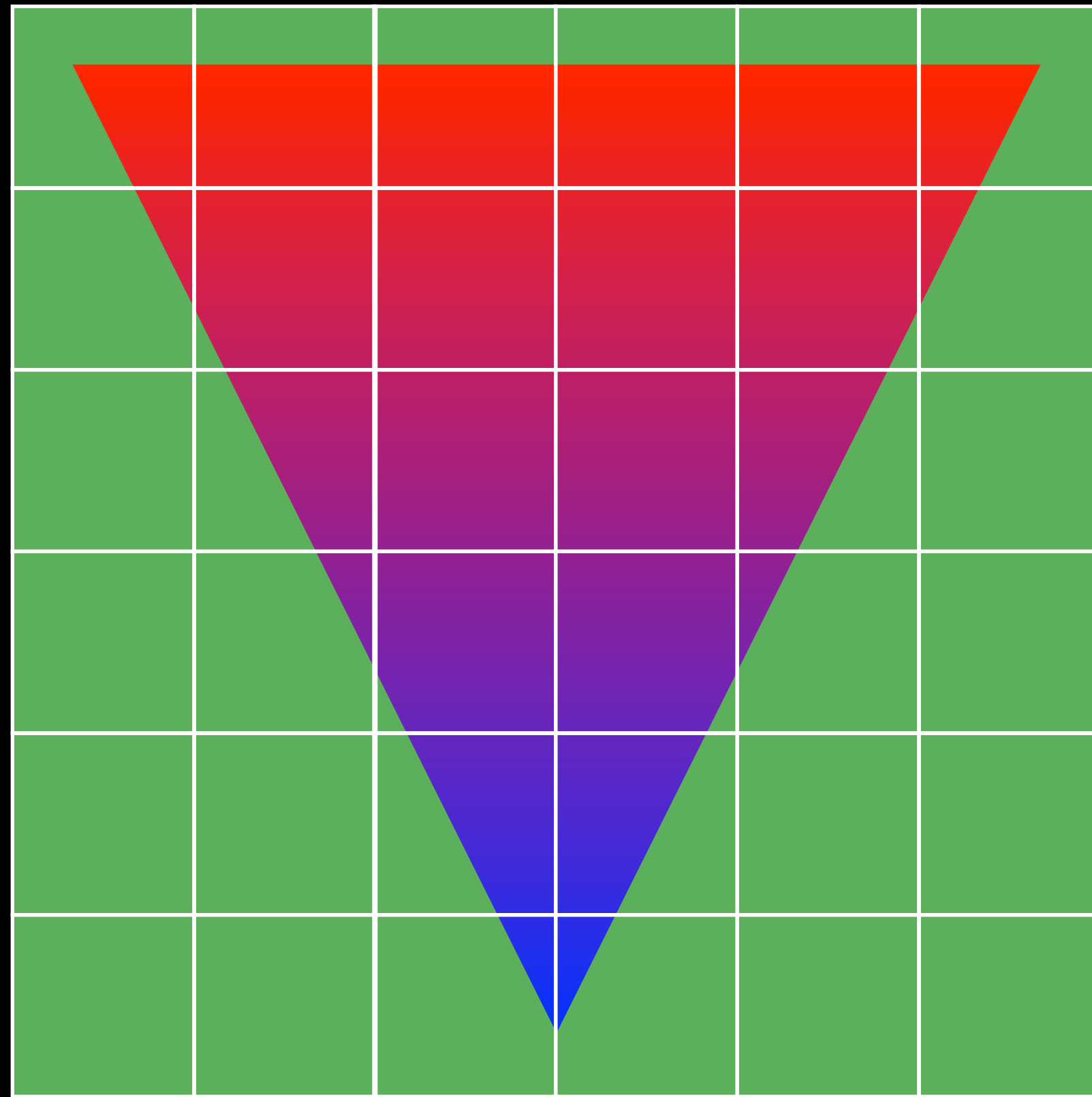


GPU

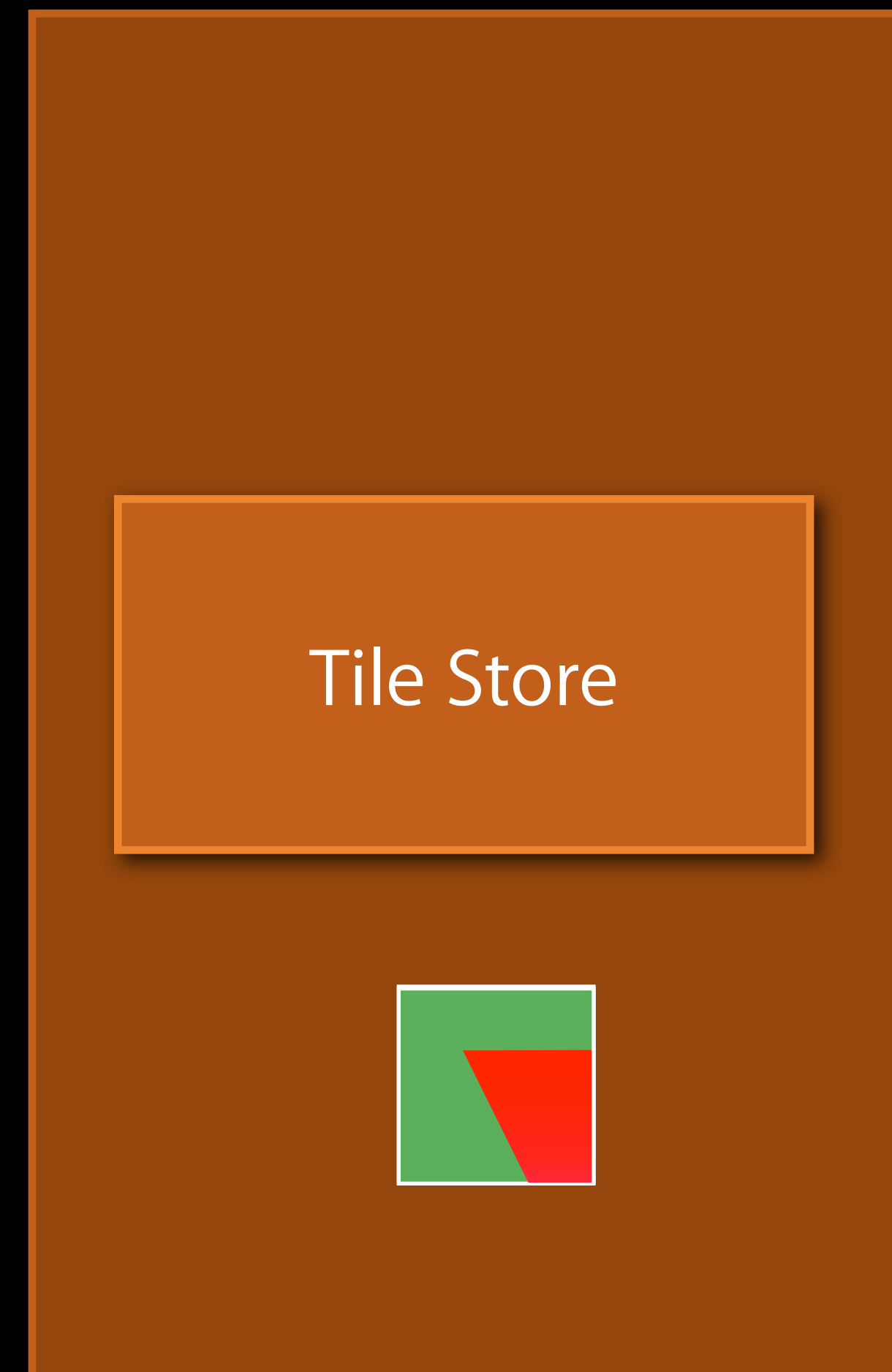


- Only need color buffer MSAA buffer has been resolved to

Unified Memory

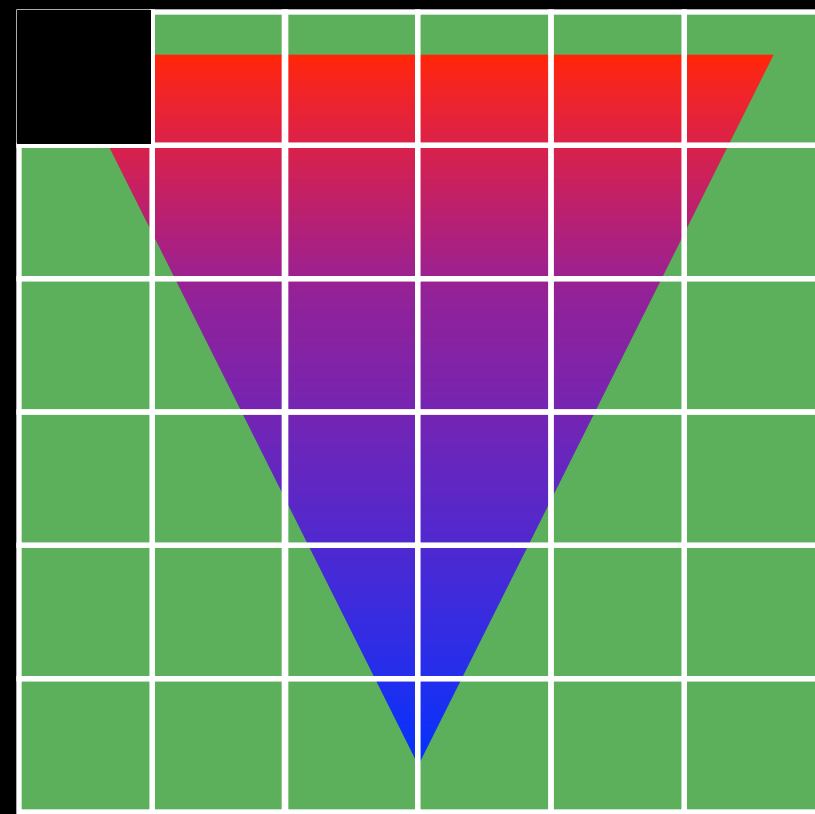


GPU

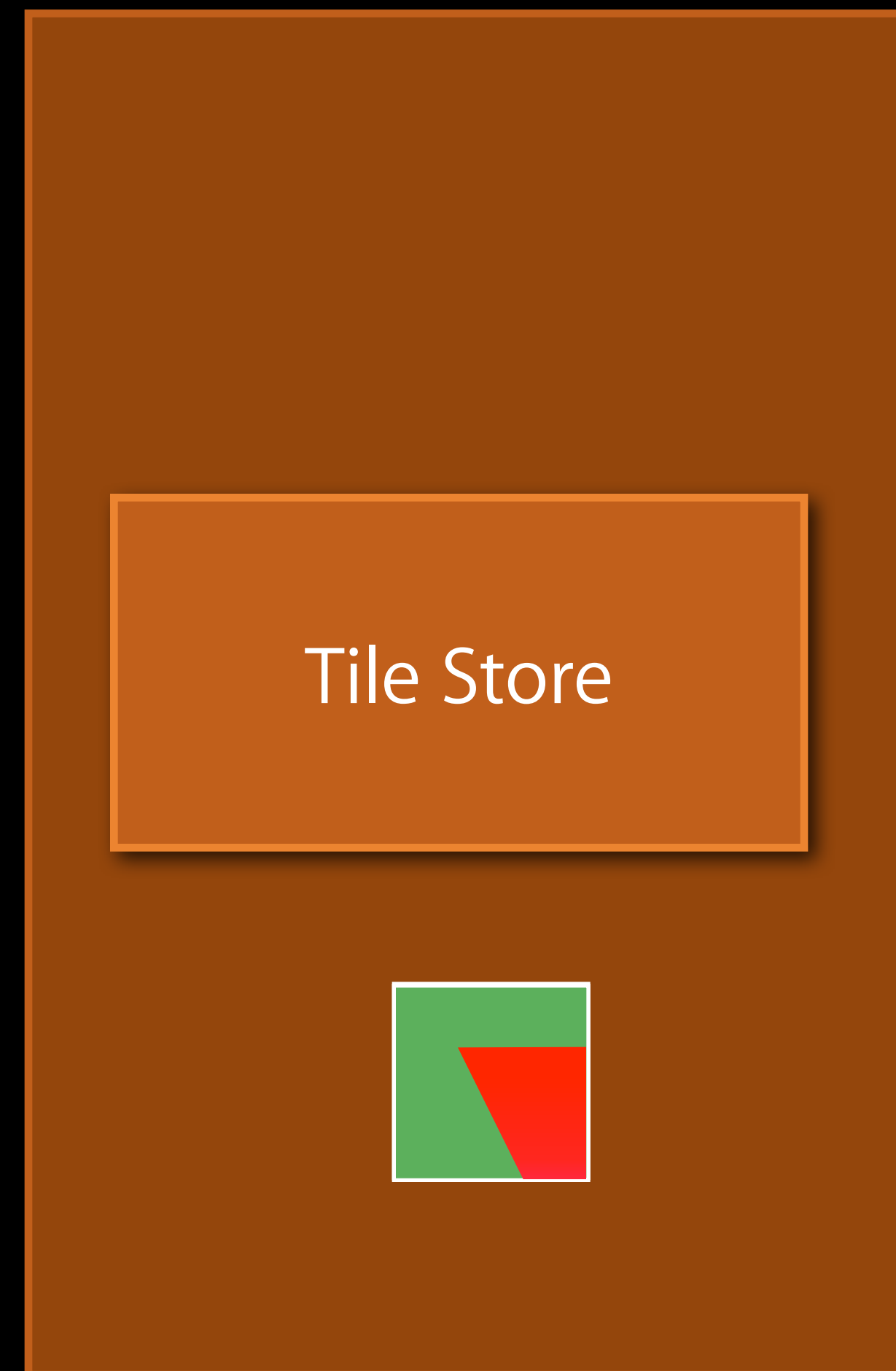


- Only need color buffer MSAA buffer has been resolved to

Unified Memory

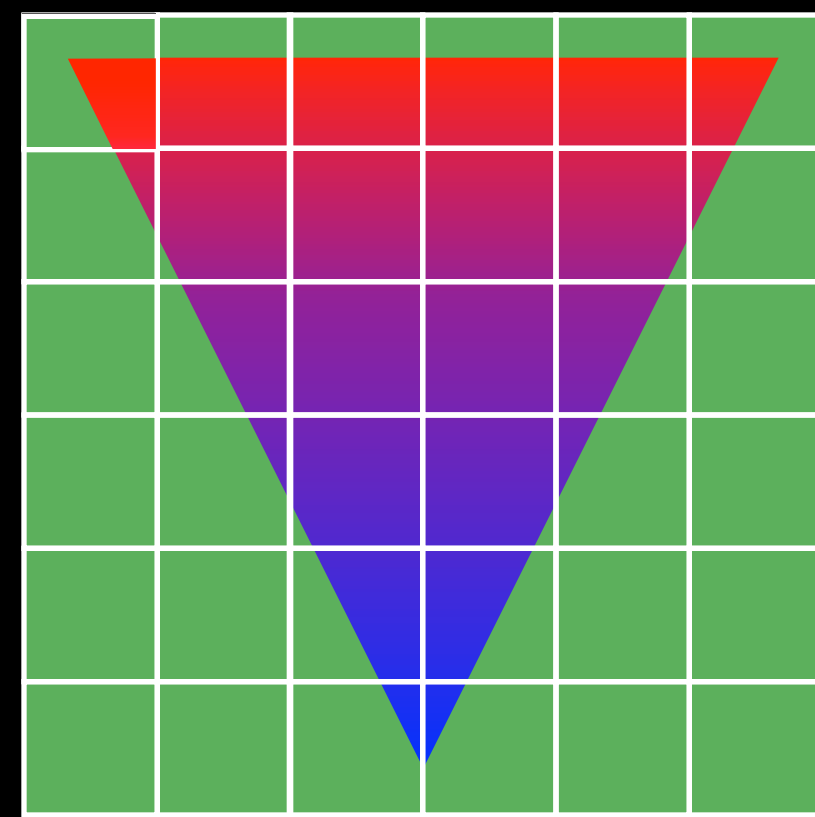


GPU

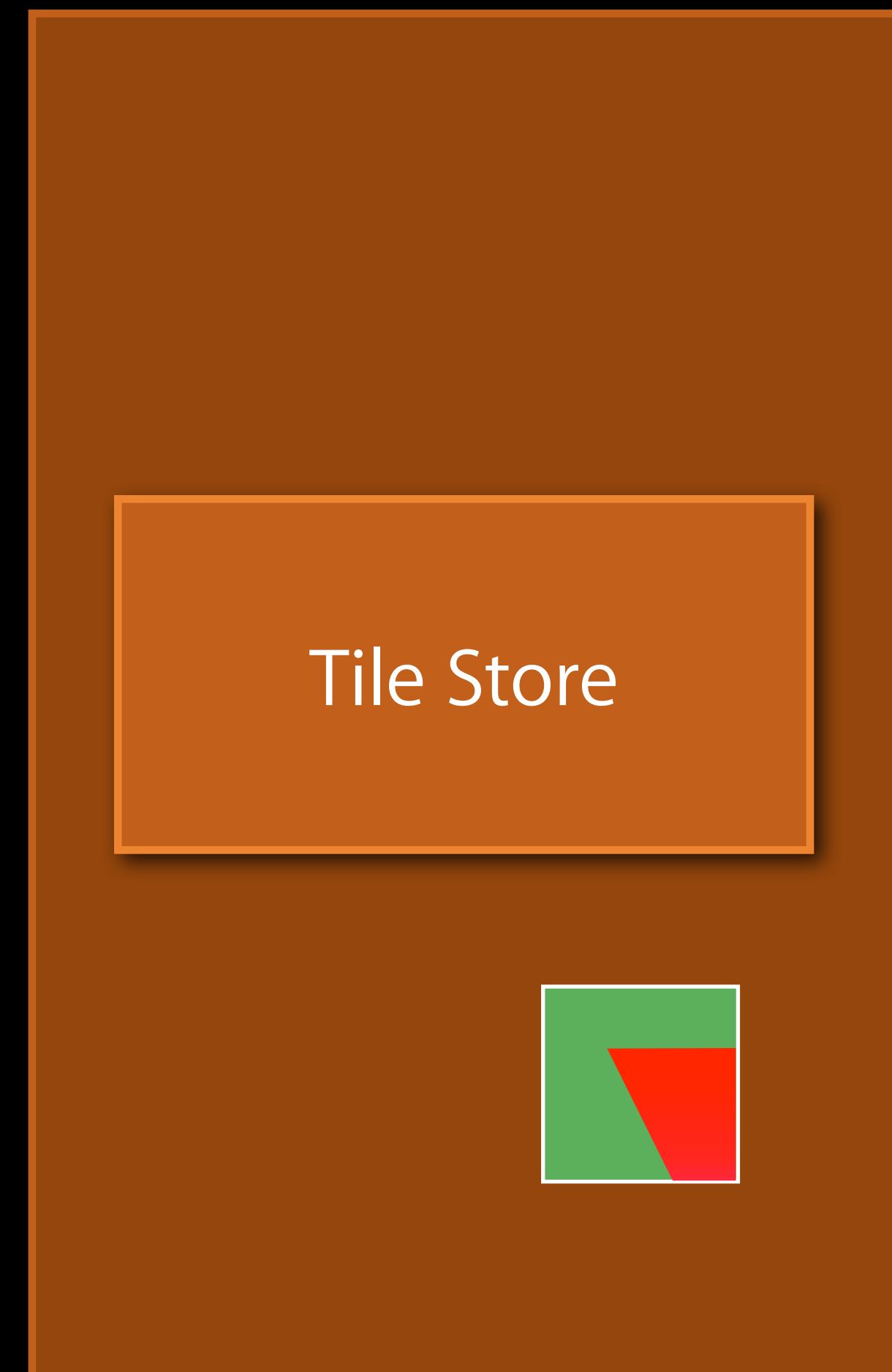


- Only need color buffer MSAA buffer has been resolved to

Unified Memory

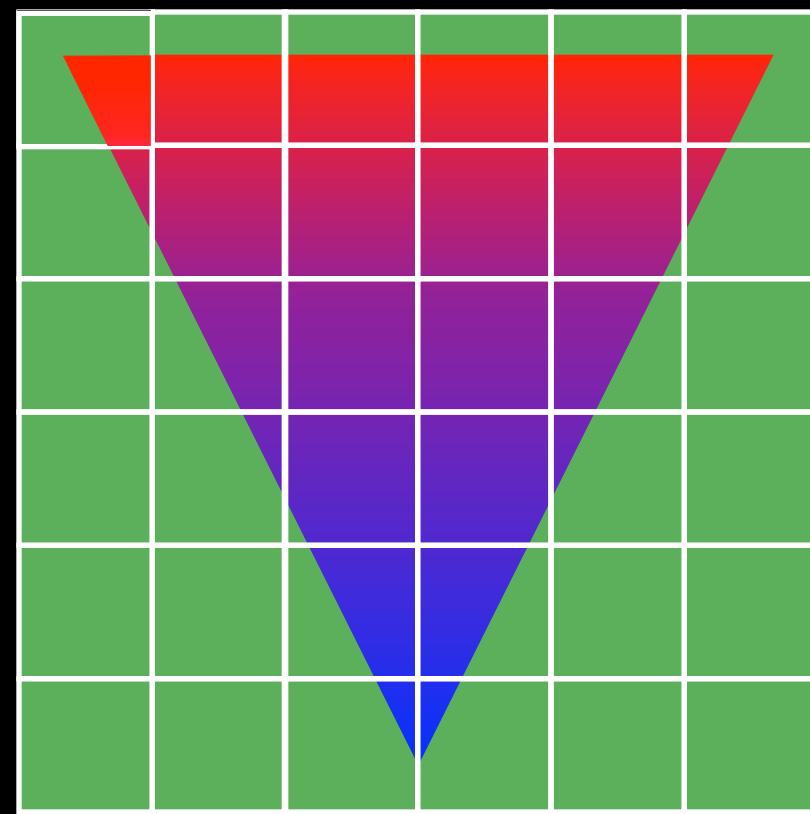


GPU

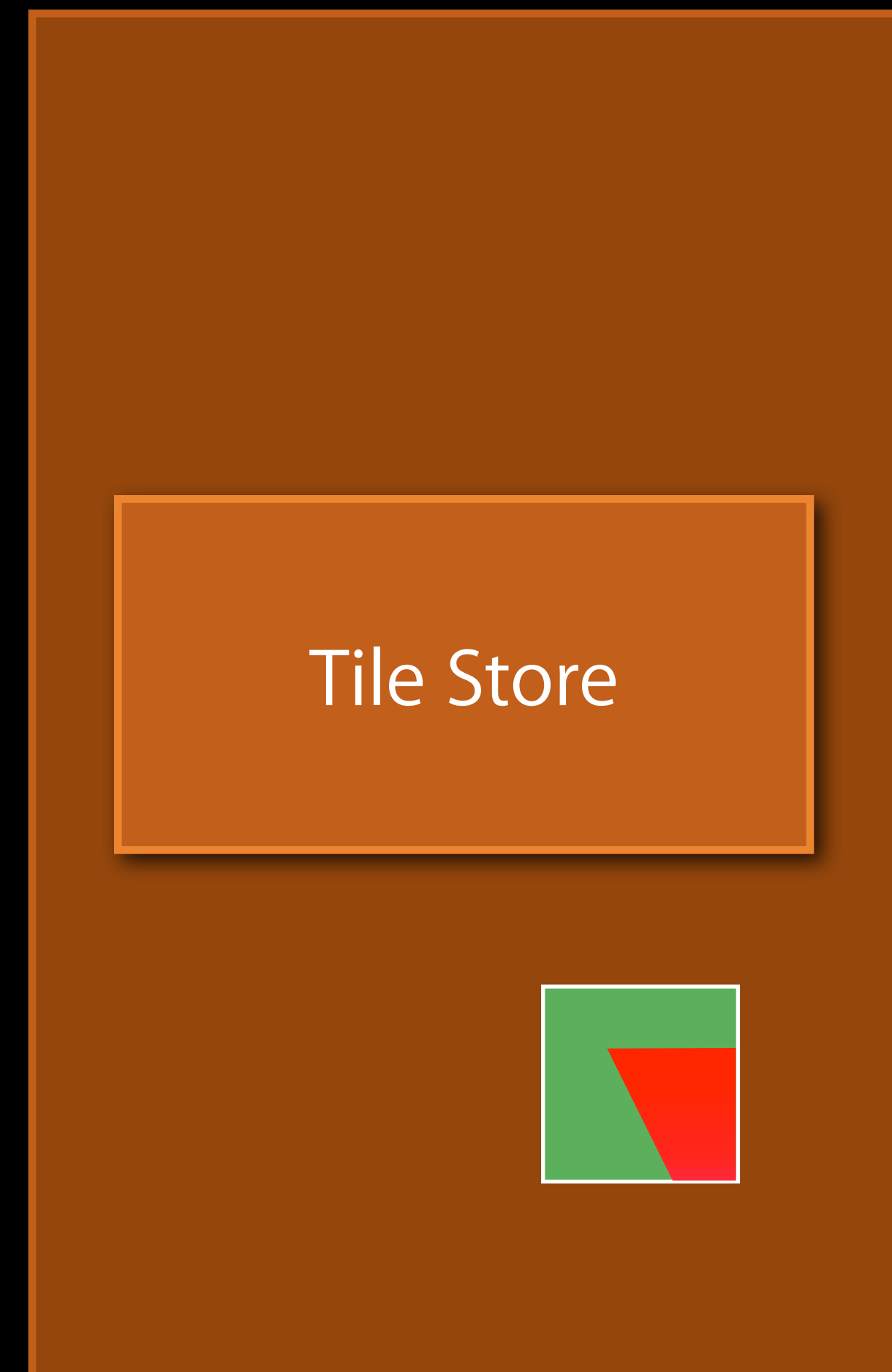


- Only need color buffer MSAA buffer has been resolved to
 - Call `glDiscardFramebufferEXT` on MSAA color buffer

Unified Memory

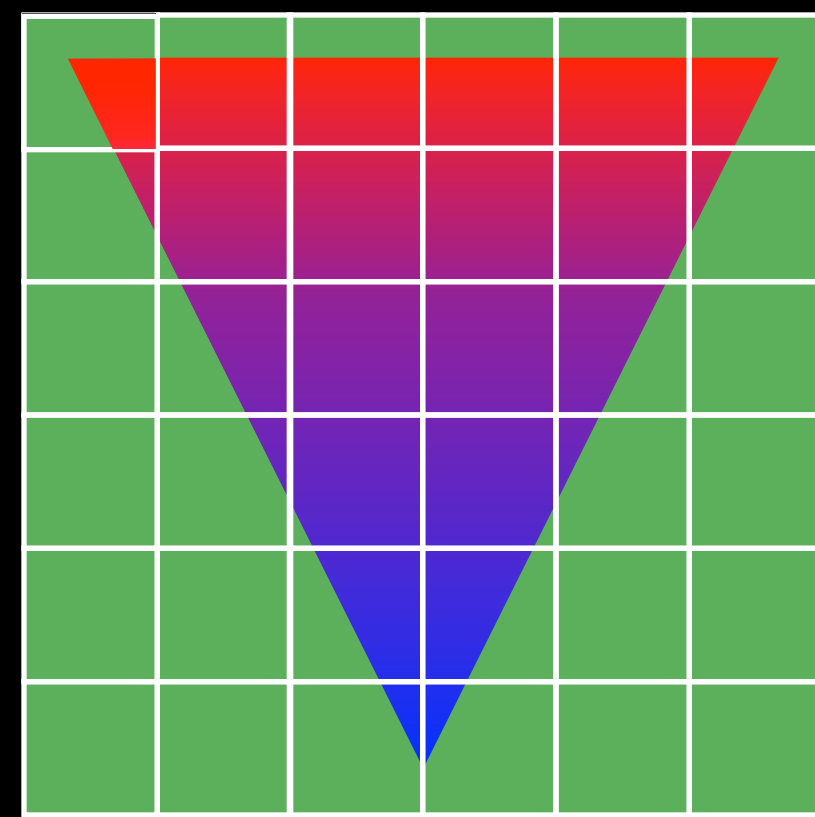


GPU

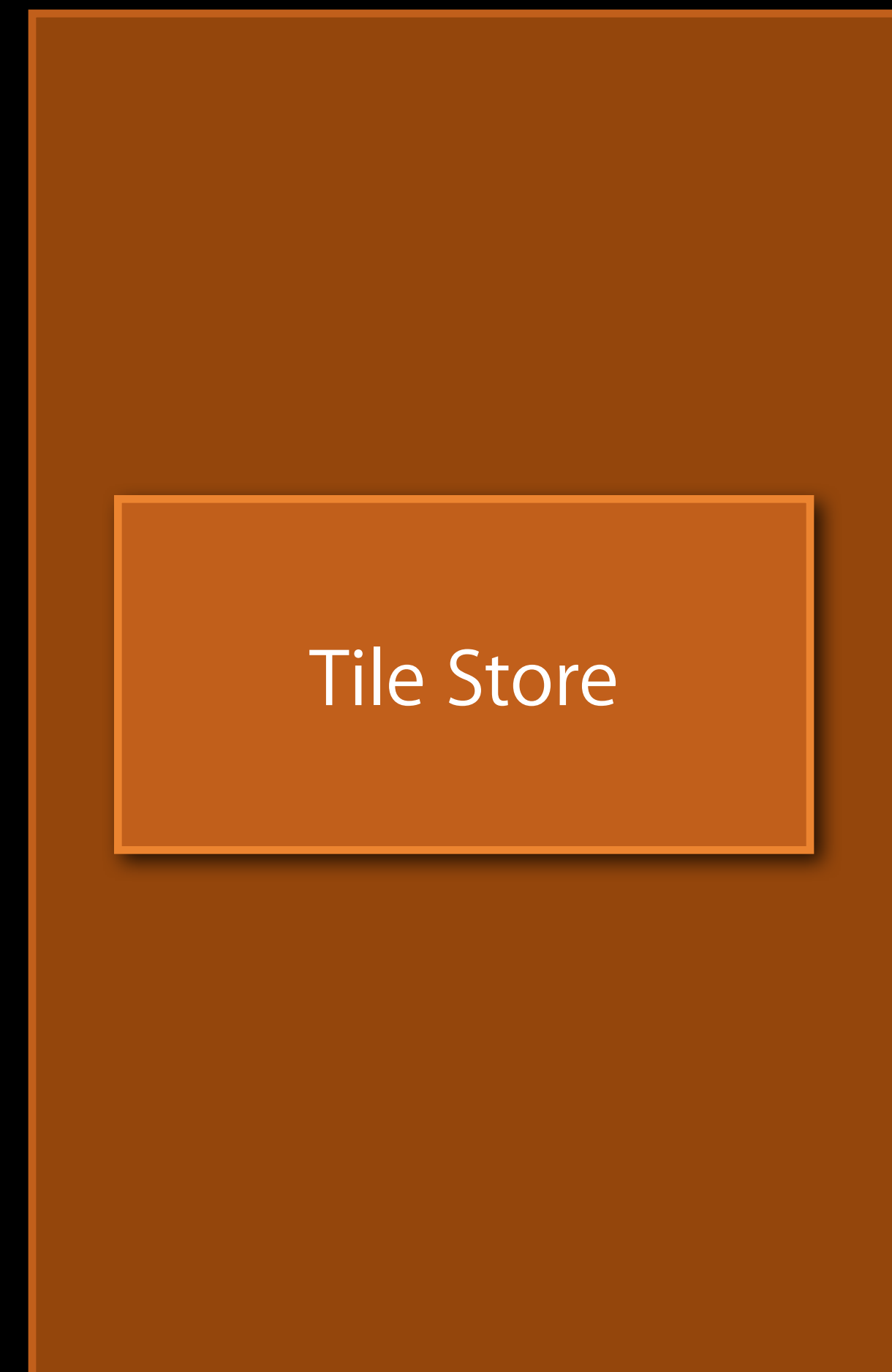


- Only need color buffer MSAA buffer has been resolved to
 - Call `glDiscardFramebufferEXT` on MSAA color buffer

Unified Memory



GPU



- Developers do not need MSAA depth buffer

Unified Memory

GPU



The diagram shows a large orange rectangle representing the GPU. Inside this rectangle, there is a smaller orange rectangle labeled "Tile Store". Below the "Tile Store" rectangle is a small square icon with a diagonal line, representing a GPU core or tile.

Tile Store

- Developers do not need MSAA depth buffer
 - Also call `glDiscardFramebufferEXT` on MSAA depth buffer

Unified Memory

GPU



The diagram illustrates the GPU architecture. It features a large orange rectangle representing the GPU. Inside this rectangle, there is a smaller orange rectangle labeled "Tile Store". Below the "Tile Store" rectangle, there is a small square icon with a dark gray background and a light gray triangle pointing towards the bottom-left corner.

Tile Store

- Developers do not need MSAA depth buffer
 - Also call `glDiscardFramebufferEXT` on MSAA depth buffer

Unified Memory

GPU



The diagram illustrates the GPU architecture. It features a large orange rectangle representing the GPU, with a smaller orange rectangle inside it labeled "Tile Store".

Tile Store

Taking TBDR into Consideration

Hidden Surface Removal

- Hidden Surface Removal a unique strength
 - Greatly reduces workload
 - Certain operation defeat HSR process
 - Enabling blending
 - Using `discard` in shader

Taking TBDR into Consideration

Using fragment discard and blending

- Draw all triangles using `discard` after drawing triangles that do not
 - Hidden Surface Removal used for triangles in opaque group

Taking TBDR into Consideration

Using fragment discard and blending

- Draw all triangles using `discard` after drawing triangles that do not
 - Hidden Surface Removal used for triangles in opaque group
- Trim geometry needing these operations
 - Worth adding more vertices to reduce fragments



Taking TBDR into Consideration

Logical Buffer Loads and Stores

- Transfers buffer between memory and GPU are expensive
- Avoid Logical Buffer Loads
 - Use `glClear` so GPU can skip them
 - Frequent framebuffer switches cause tile thrashing
- Avoid Logical Buffer Stores
 - Use `glDiscardFramebufferEXT`
 - Especially for multisample anti-aliased buffers

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec2 offset;
...

main()
{
    vec2 offsetCoord = texCoord + offset;
    vec4 color = texture(tex, offsetCoord);
    ...
}
```



Dependent Texture Sampling

The fragment shader in Program #12 performed dependent texture reads which are slower than non-dependent texture reads.

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec2 offset;
...

main()
{
    vec2 offsetCoord = texCoord + offset;
    vec4 color = texture(tex, offsetCoord);
    ...
}
```

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;  
varying vec2 texCoord;  
varying vec2 offset;
```

```
...
```

```
main()
```

```
{
```

```
    vec2 offsetCoord = texCoord + offset;
```

```
    vec4 color = texture(tex, offsetCoord);
```

```
    ...
```

```
}
```

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec2 offset;
...

main()
{
    vec2 offsetCoord = texCoord + offset;
    vec4 color = texture(tex, offsetCoord);
    ...
}
```

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec2 offset;
...

main()
{
    vec2 offsetCoord = texCoord + offset;
    vec4 color = texture(tex, offsetCoord);
    ...
}
```

Dependent Texture Sampling

- Calculate texture coordinate in shader
 - Sample with `texture*` function

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec2 offset;
...

main()
{
    vec2 offsetCoord = texCoord + offset;
    vec4 color = texture(tex, offsetCoord);
    ...
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;
varying vec4 packedTexCoords;

...

main()
{
    vec4 color1 = texture2D(tex, packedTexCoords.xy);
    vec4 color2 = texture2D(tex, packedTexCoords.zw);
    ...
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;  
varying vec4 packedTexCoords;
```

```
...
```

```
main()
```

```
{
```

```
    vec4 color1 = texture2D(tex, packedTexCoords.xy);
```

```
    vec4 color2 = texture2D(tex, packedTexCoords.zw);
```

```
    ...
```

```
}
```


Dependent Texture Sampling

```
uniform sampler2D tex;  
varying vec4 packedTexCoords;
```

```
...
```

```
main()  
{
```

```
    vec4 color1 = texture2D(tex, packedTexCoords.xy);
```

```
    vec4 color2 = texture2D(tex, packedTexCoords.zw);
```

```
    ...
```

```
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;
varying vec4 packedTexCoords;

...

main()
{
    vec4 color1 = texture2D(tex, packedTexCoords.xy);
    vec4 color2 = texture2D(tex, packedTexCoords.zw);
    ...
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;
varying vec4 packedTexCoords;

...

main()
{
    vec4 color1 = texture2D(tex, packedTexCoords.xy);
    vec4 color2 = texture2D(tex, packedTexCoords.zw);
    ...
}
```

Dependent Texture Sampling

- High latency to sample texture in unified memory
 - GPU prefetches texture data for nondependent reads
 - Done in rasterization before shader executed
 - Cannot prefetch if coordinate calculated in shader
 - Shader stalls, waiting for texture data
- Minimize dependent texture samples
 - Hoist calculation
 - Perform coordinate calculation in vertex shader
 - Calculate in app and put in uniform

Dependent Texture Sampling

```
uniform sampler2D tex;
varying vec2 texCoord0;
varying vec2 texCoord1;

...

main()
{
    vec4 color1 = texture2D(tex, texCoord0);
    vec4 color2 = texture2D(tex, texCoord1);

    ...

}
```

Dependent Texture Sampling

```
uniform sampler2D tex;  
varying vec2 texCoord0;  
varying vec2 texCoord1;
```

...

```
main()  
{  
    vec4 color1 = texture2D(tex, texCoord0);  
    vec4 color2 = texture2D(tex, texCoord1);  
    ...  
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;  
varying vec2 texCoord0;  
varying vec2 texCoord1;
```

```
...
```

```
main()
```

```
{
```

```
    vec4 color1 = texture2D(tex, texCoord0);  
    vec4 color2 = texture2D(tex, texCoord1);
```

```
    ...
```

```
}
```

Dependent Texture Sampling

```
uniform sampler2D tex;
varying vec2 texCoord0;
varying vec2 texCoord1;

...

main()
{
    vec4 color1 = texture2D(tex, texCoord0);
    vec4 color2 = texture2D(tex, texCoord1);

    ...

}
```



Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

```
...
```

```
main()  
{
```

 **Fragment Shader Dynamic Branching**
The fragment shader in Program #14 "Light" contains dynamic branch instructions, which reduces the performance of the shader.

```
{  
    outVal = 1.0;  
}
```

```
else
```

```
{  
    outVal = 0.0;  
}
```

```
...
```

```
}
```

Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

```
...
```

```
main()
```

```
{
```

```
    if(testVal > 1.0)
```

```
    {
```

```
        outVal = 1.0;
```

```
    }
```

```
    else
```

```
    {
```

```
        outVal = 0.0;
```

```
    }
```

```
    ...
```

```
}
```

Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

...

```
main()  
{
```

```
    if(testVal > 1.0)
```

```
    {
```

```
        outVal = 1.0;
```

```
    }
```

```
    else
```

```
    {
```

```
        outVal = 0.0;
```

```
    }
```

```
    ...
```

```
}
```

Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

```
...
```

```
main()
```

```
{
```

```
    if(testVal > 1.0)
```

```
    {
```

```
        outVal = 1.0;
```

```
    }
```

```
    else
```

```
    {
```

```
        outVal = 0.0;
```

```
    }
```

```
    ...
```

```
}
```

Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

...

```
main()  
{
```

```
    if(testVal > 1.0)
```

```
    {
```

```
        outVal = 1.0;
```

```
    }
```

```
    else
```

```
    {
```

```
        outVal = 0.0;
```

```
    }
```

...

```
}
```

Dynamic Branching

A simple example

```
attribute float testVal;  
varying float outVal;
```

```
...
```

```
main()
```

```
{
```

```
    if(testVal > 1.0)
```

```
    {
```

```
        outVal = 1.0;
```

```
    }
```

```
    else
```

```
    {
```

```
        outVal = 0.0;
```

```
    }
```

```
    ...
```

```
}
```

Dynamic Branching

- GPUs are highly parallel devices
 - Process multiple vertices and fragments simultaneously
- Special branch mode for execution
 - More latency to stay in sync
- If possible, calculate predicate outside of shader
 - Branches on uniforms do not incur same overhead

Dynamic Branching

- GPUs are highly parallel devices
 - Process multiple vertices and fragments simultaneously
- Special branch mode for execution
 - More latency to stay in sync
- If possible, calculate predicate outside of shader
 - Branches on uniforms do not incur same overhead
- Shaders using **both** Dependent Texture Sampling **and** Dynamic Branching are particularly costly

Minimizing CPU Overhead

Managing draw calls

Optimizing Draw Call Performance

- Most CPU overhead in draw calls
- More state set for draw, more expensive draw
 - State setting looks inexpensive
 - Work for state set deferred until draw

Optimizing Draw Call Performance

- Most CPU overhead in draw calls
- More state set for draw, more expensive draw
 - State setting looks inexpensive
 - Work for state set deferred until draw
- Maximize efficiency of each draw

Optimizing Draw Call Performance

- Most CPU overhead in draw calls
- More state set for draw, more expensive draw
 - State setting looks inexpensive
 - Work for state set deferred until draw
- Maximize efficiency of each draw

Optimizing Draw Call Performance

- Most CPU overhead in draw calls
- More state set for draw, more expensive draw
 - State setting looks inexpensive
 - Work for state set deferred until draw
- Maximize efficiency of each draw

⚠ Redundant Call

`glBindTexture(GL_TEXTURE_2D, 1u)` set a piece of GL state to its current value.

⚠ Inefficient State Update

`glViewport(0, 0, 1024, 1024)` sets a piece of GL state more than once before it is used by a clear or a draw call.

Optimizing Draw Call Performance

- Most CPU overhead in draw calls
- More state set for draw, more expensive draw
 - State setting looks inexpensive
 - Work for state set deferred until draw
- Maximize efficiency of each draw
- Reduce inefficient state setting
 - Shadow state
 - Sort state

Optimizing Draw Call Performance

- Some fixed overhead for draw
 - State validation
 - Call to driver

Optimizing Draw Call Performance

- Some fixed overhead for draw
 - State validation
 - Call to driver
- Minimize number of draw calls made
 - Object culling
 - Coalesce calls
 - Instancing
 - Vertex batching
 - Texture atlases

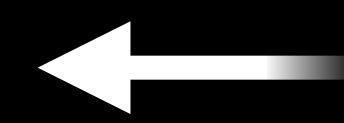
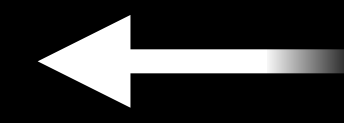
Texture Atlases

Bind and draw



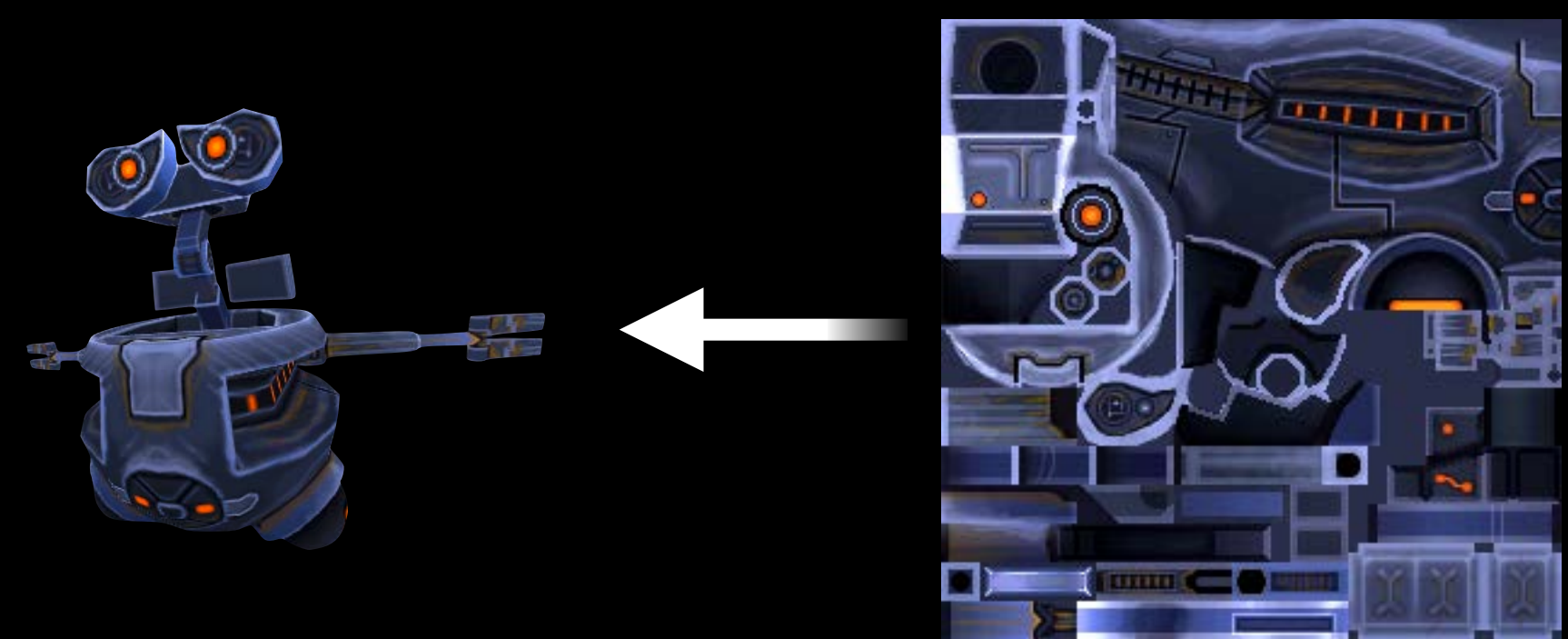
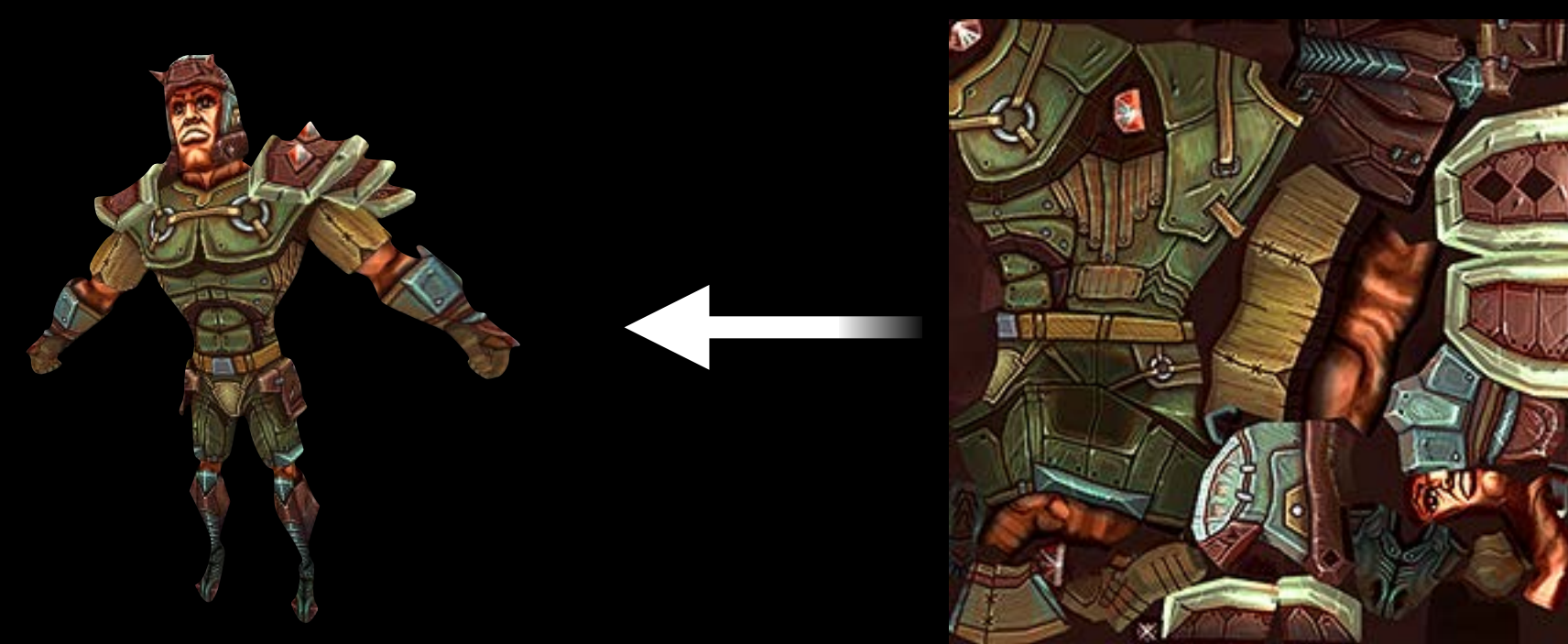
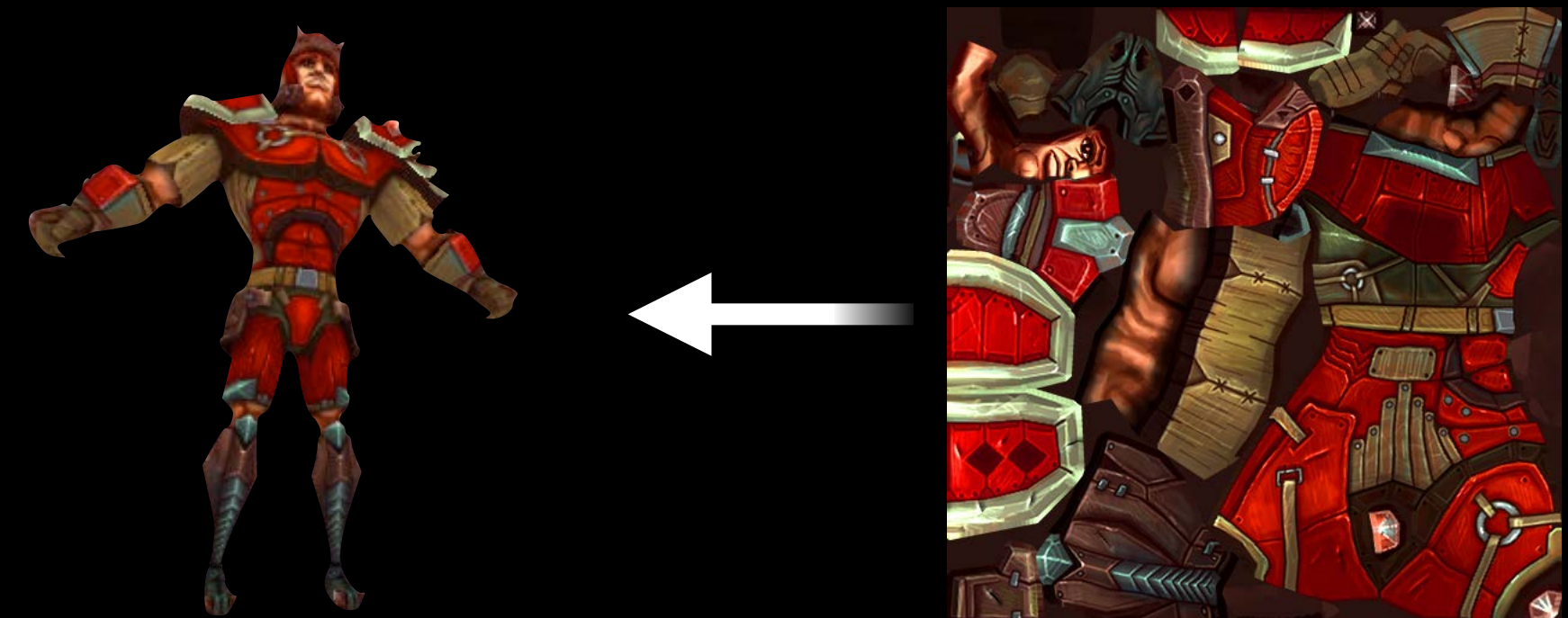
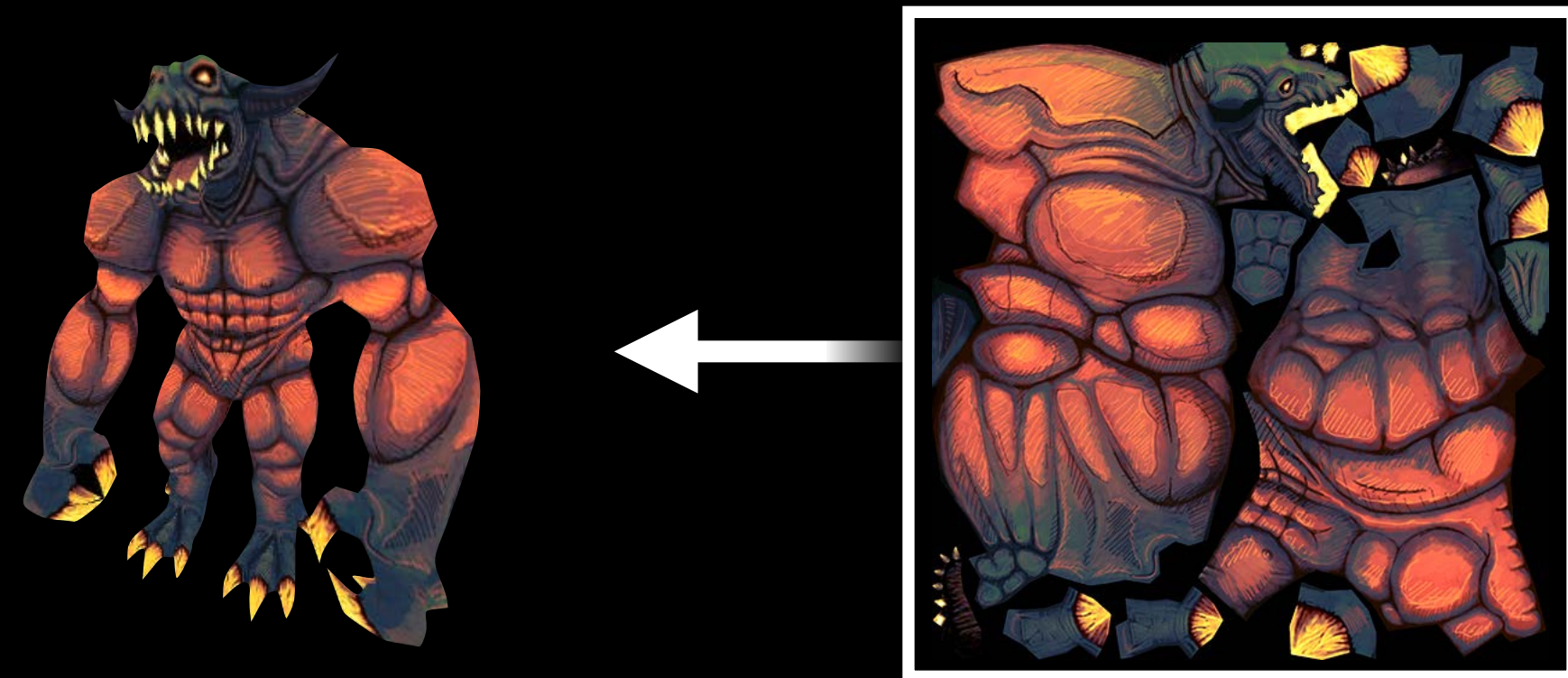
Texture Atlases

Bind and draw



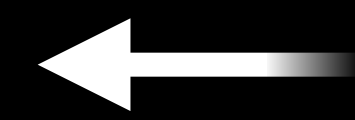
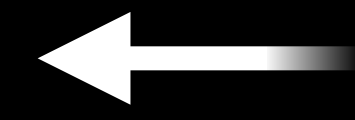
Texture Atlases

Bind and draw



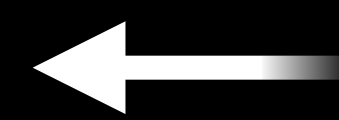
Texture Atlases

Bind and draw



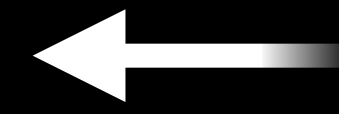
Texture Atlases

Bind and draw



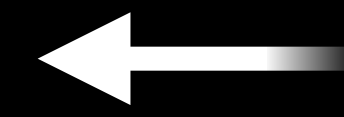
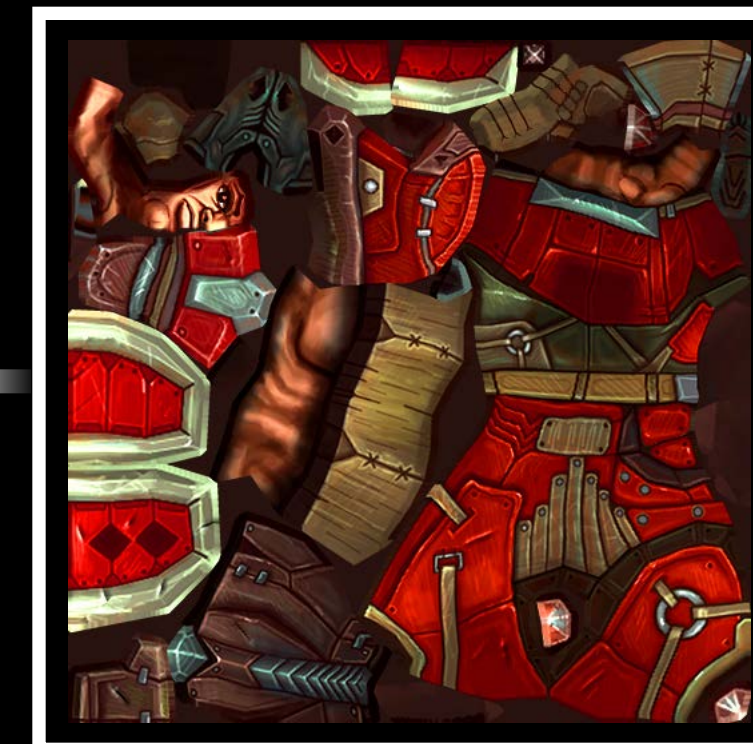
Texture Atlases

Bind and draw



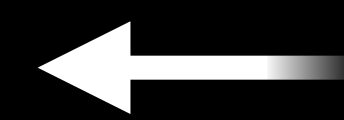
Texture Atlases

Bind and draw



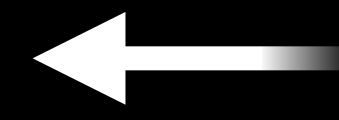
Texture Atlases

Bind and draw



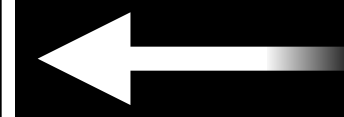
Texture Atlases

Bind and draw



Texture Atlases

Bind and draw



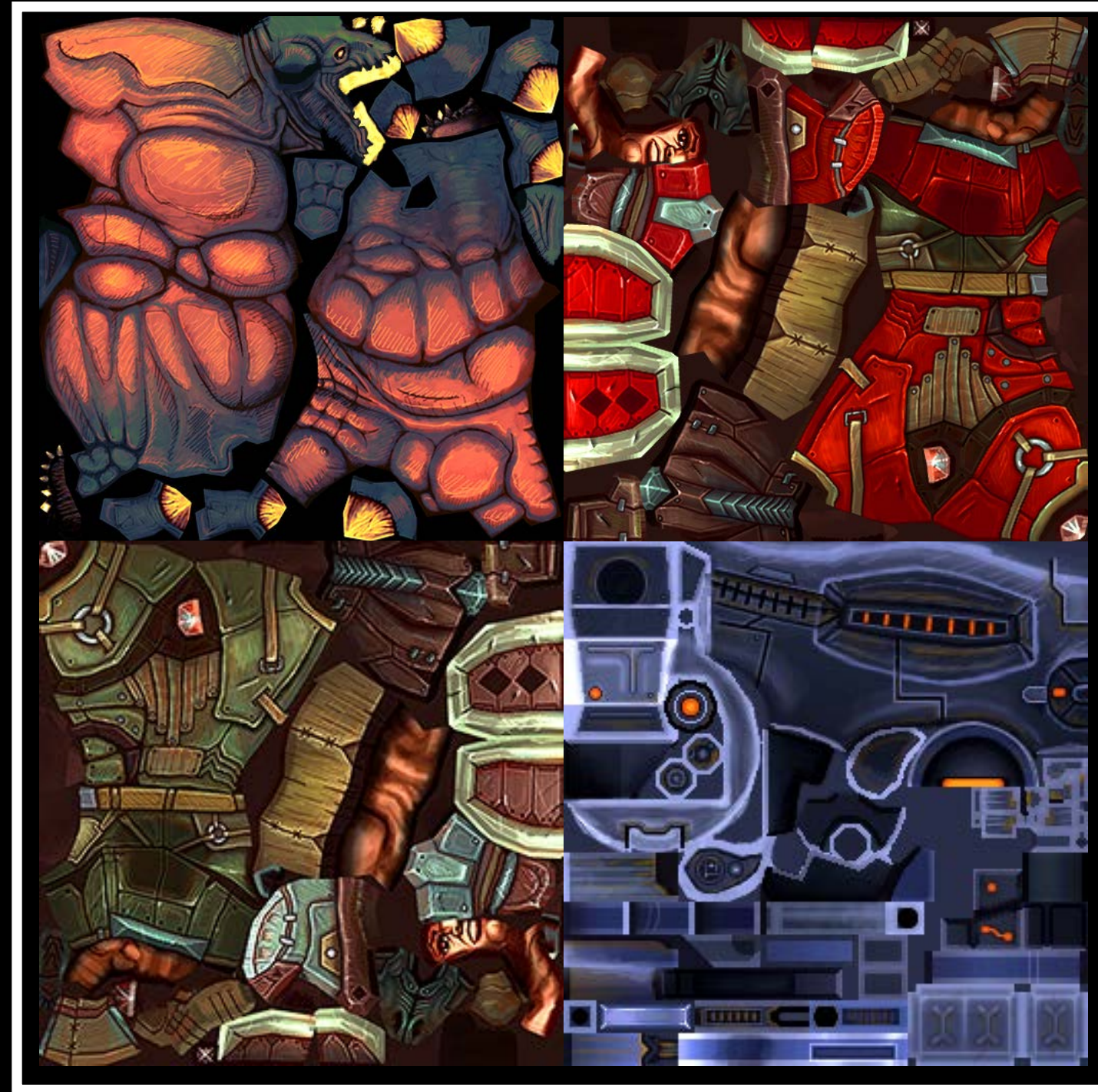
Texture Atlases

Reducing binds



Texture Atlases

Reducing binds



Texture Atlases

Reducing binds



Texture Atlases

Reducing binds



Texture Atlases

Reducing binds



Texture Atlases

Reducing binds



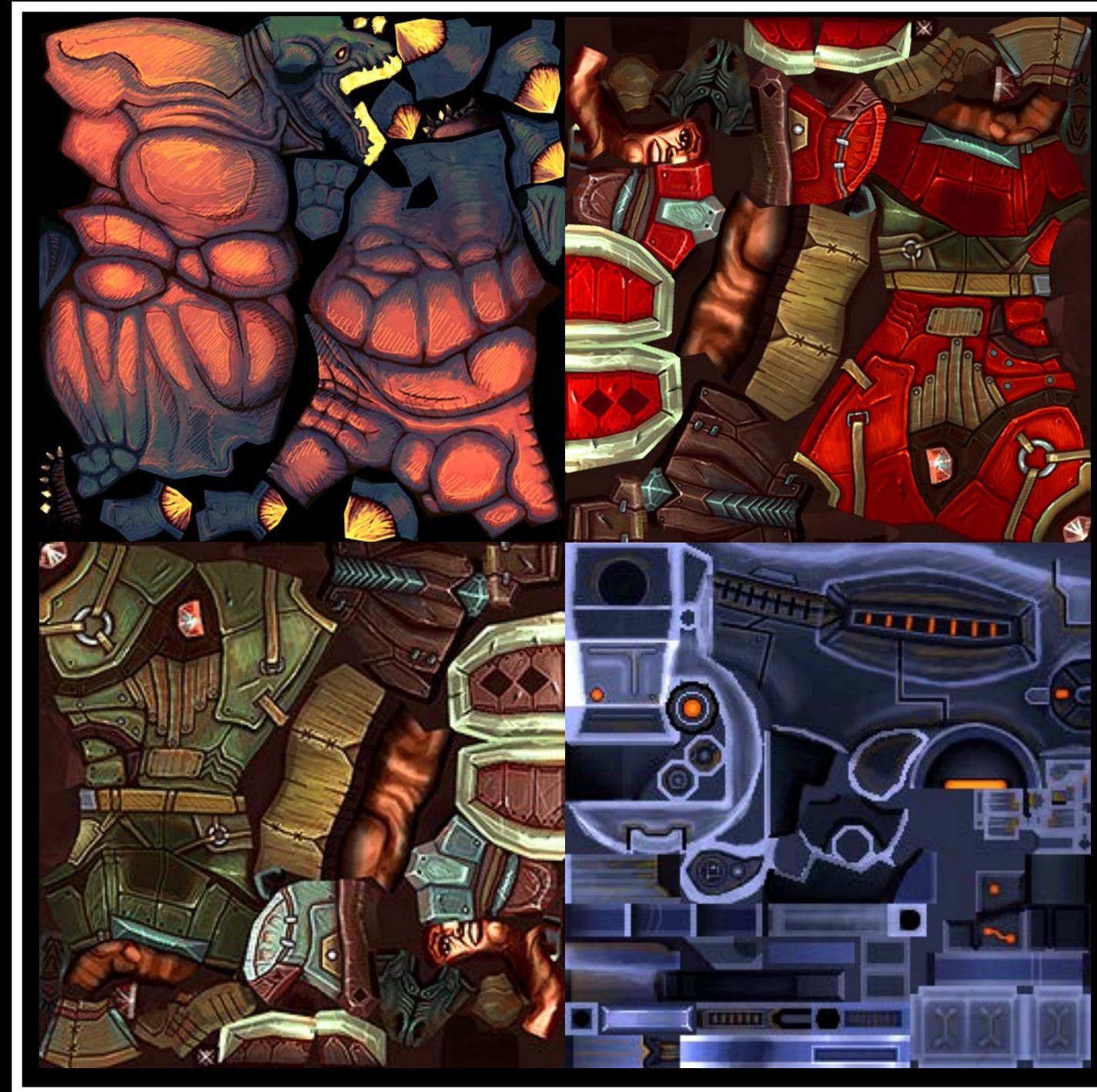
Texture Atlases

Combining draws



Texture Atlases

Combining draws



Texture Atlases

Combining draws



Texture Atlases

Sprite Kit texture atlas tool

- Combines images efficiently
- Produces property list denoting sub-images
 - Scale texture coordinates based on plist
- 'TextureAtlas' command line tool in Xcode



More Information

Allan Schaffer

Graphics and Game Technologies Evangelist
aschaffer@apple.com

Documentation

OpenGL ES Programming Guide for iOS
<https://developer.apple.com/opengles>

Apple Developer Forums

<http://devforums.apple.com>

Related Sessions

Introduction to Sprite Kit

Presidio
Wednesday 11:30AM

What's New in OpenGL for OS X

Marina
Thursday 2:00PM

Labs

Sprite Kit Lab

Graphics and Games Lab B
Thursday 9:00AM

OpenGL and OpenGL ES Lab

Graphics and Games Lab A
Thursday 10:15AM

Summary

- Reduce draw call overhead
 - Use techniques including instancing and texture atlases
- Consider GPU's operation when architecting your renderer and in performance investigations
 - GPU tools help greatly
 - Tile-Based Deferred Rendering has some special consideration

 WWDC2013