Graphics Programming

Lecturer: **Rachel McDonnell**

Assistant Professor in Creative Technologies

Rachel.McDonnell@cs.tcd.ie

Course www: https://www.scss.tcd.ie/Rachel.McDonnell/

Credits: Some notes taken from Prof. Jeff Chastine

Lab Today

- Online now
- Build a simple GLUT/OpenGL program and alter it
- Visual C++ project
  - Install GLUT library
- Play around with it to get an understanding of how OpenGL works
  - Add a new response key

Unity 3D


testscene.unity – Bootcamp – PC and Mac Standalone
Pivot
Local
Layers
Layout
Game
Free Aspect
Maximize on Play
Gizmos
Stats
Inspector
Camera
Static
Tag
Untagged
Layer
Default
Transform
Position
X 17.74164
Y 3.618703
Z 17.97578
Rotation
X 8.649139
Y 330.9547
Z 0.0009765625
Scale
X 1
Y 1
Z 1
Camera
Clear Flags
Skybox
Background
Culling Mask
Mixed ...
Projection
Perspective
Field of View
36.84956
Clipping Planes
Near 0.3
Far 115
Normalized View Port Rect
X 0
Y 0
W 1
H 1
Depth
0
Rendering Path
Use Player Settings
Target Texture
None (Render Texture)
Flare Layer
GUILayer
Audio Listener
Animation
Color Correction Curves (Script)
Mode
Simple
Red
Green
Blue
Reset
Selective
Curve Resolution
256
Bloom And Flares (Script)
Mode
Advanced
Intensity
0.3
Blur iterations
3
Blur spread
1.188793
Hierarchy
Create
Camera
Soldier
SoldierFinalPos
EndTrigger
HudWeapons
MainMenuScreen
QualityManager
Radar
SargeManager
SoldierTargetHUD
SoundManager
SoundObjectManager
aquiris
beams_columns
Project
Create
CutsceneAnimations
Editor
Fonts
Robustik
Robustik-Bold
font material
font Texture
Robustik-Regular_9
Robustik-Regular_10
Robustik-Regular_12
Robustik-Regular_14
Robustik-Regular_14
Gizmos
Locomotion System
Inspector
Scene
Textured
RGB

Unreal Engine 4

Why OpenGL?

- No ready-to-use tools
- Graphics programming is hard
- Much, much longer to create a game
- 3D programming is very time consuming!

- Mastering OpenGL will lead you towards becoming a graphics programmer
- You will have a deeper understanding of how game engines have been built

Essential checklist

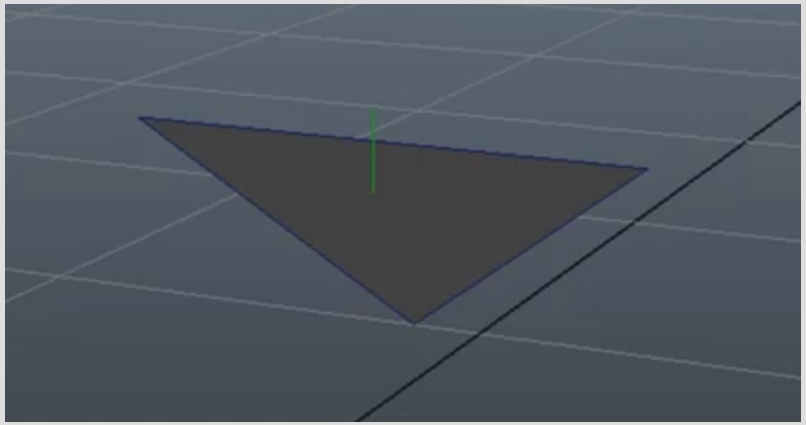
- ✔ always have a pencil and paper
- ✔ solve your problem before you start coding
- ✔ know how to compile and link against libraries
- ✔ know how to use memory, pointers, addresses
- ✔ understand the hardware pipeline
- ✔ make a 3d maths cheat sheet
- ✔ do debugging (visual and programmatic)
- ✔ print the Quick Reference Card for OpenGL
- ✔ start assignments ASAP

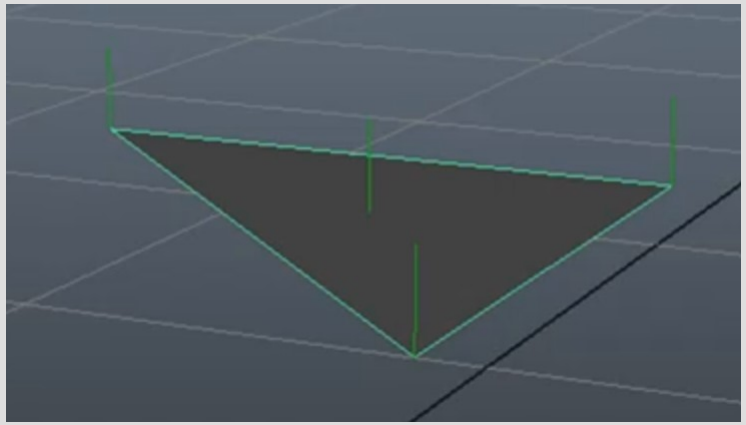
Overview

- OpenGL background
- OpenGL conventions,
- GLUT Event loop, callback registration
- OpenGL primitives, OpenGL objects
- Shaders
- Vertex Buffer Objects
- Books, resources, recommended reading

Quick Background

- A Vertex is a 3D point (x,y,z)
- A triangle
  - Is made from 3 vertices
  - Has a normal
  - Note: vertices can have normals too!

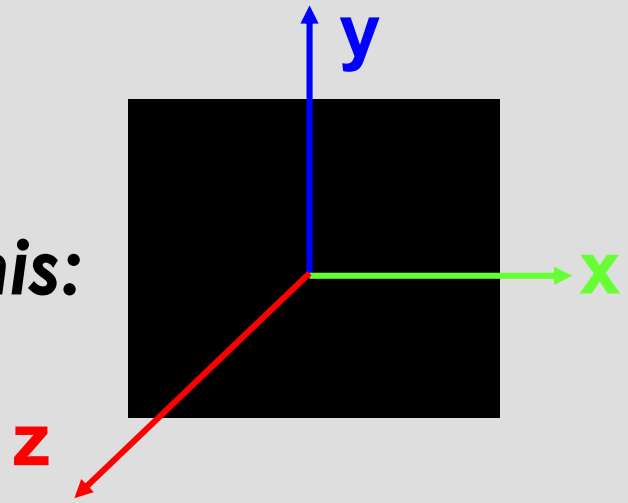




Sources of 3D data

Directly specify the Three-Dimensional data

Fine for this:



=

(-1, -1, -1)
(1, -1, -1)
(1, 1, -1)
(-1, 1, -1)
(-1, -1, 1)
(1, -1, 1)
(1, 1, 1)
(-1, 1, 1)

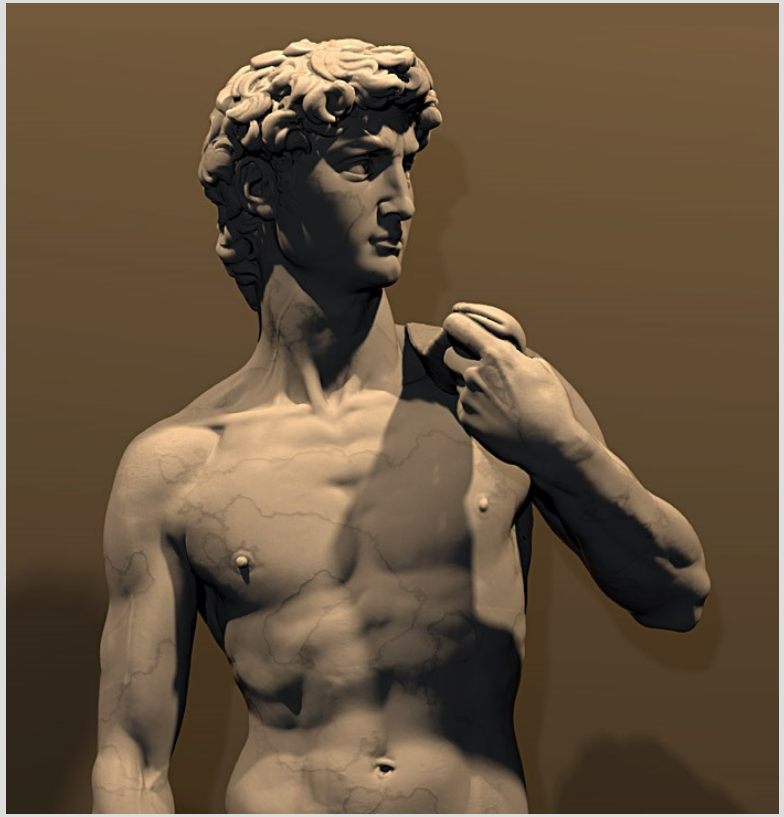
... But not for this!

Modelling Program

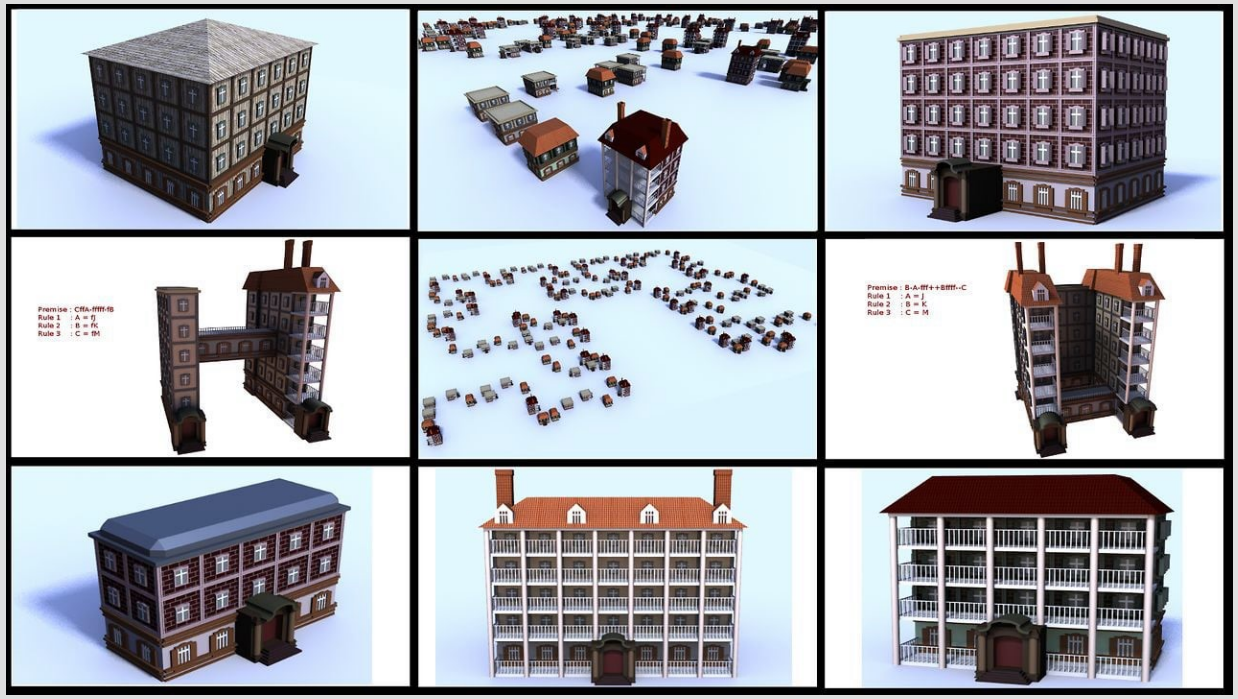
- 3ds Max, Maya, Softimage, Blender, Auto CAD etc.

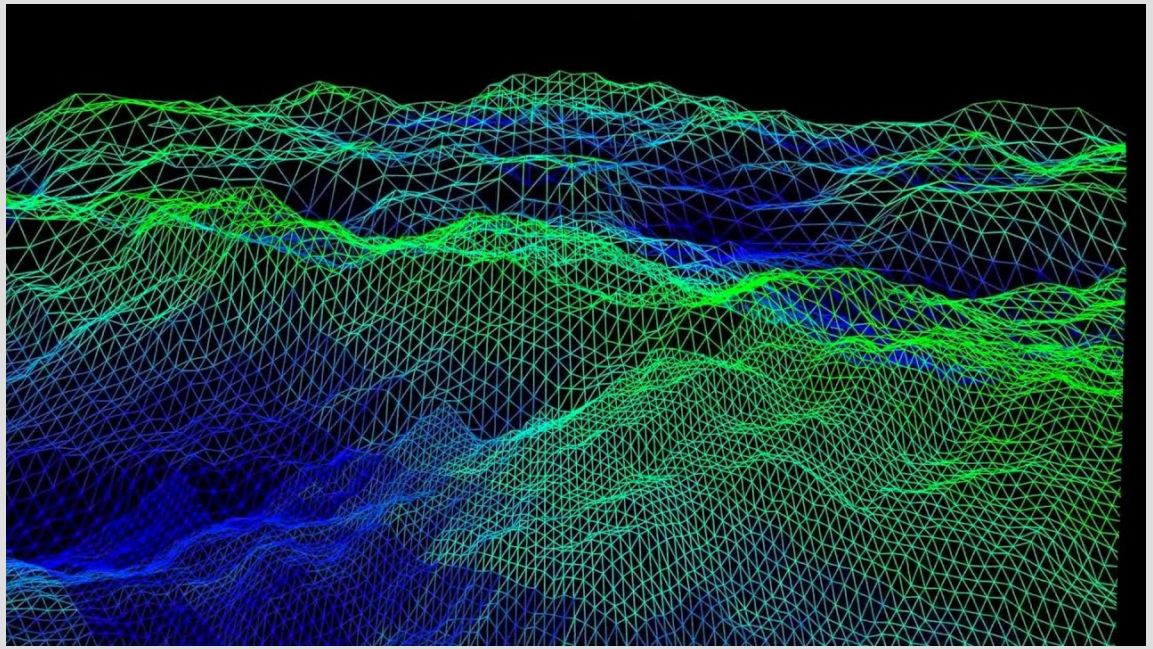
https://www.autodesk.com/education/free-software/all

Laser Scanning



Procedural Models

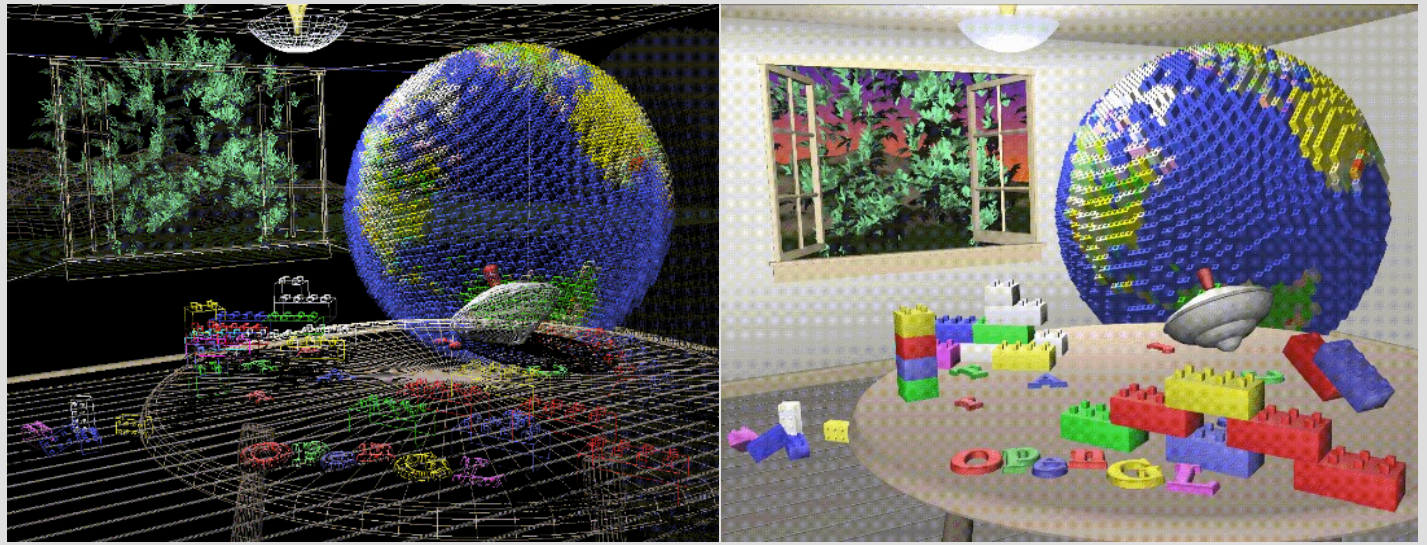




Algorithmic rules to generate complex models

Rendering

- *Rendering* is the process by which a computer creates images from models. These *models*, or objects, are constructed from geometric primitives - points, lines, and polygons - that are specified by their vertices
- The final rendered image consists of pixels drawn on the screen



A Graphics System

- Input devices
- Central Processing Unit
- Graphics Processing Unit
- Memory
- Frame buffer
- Output devices

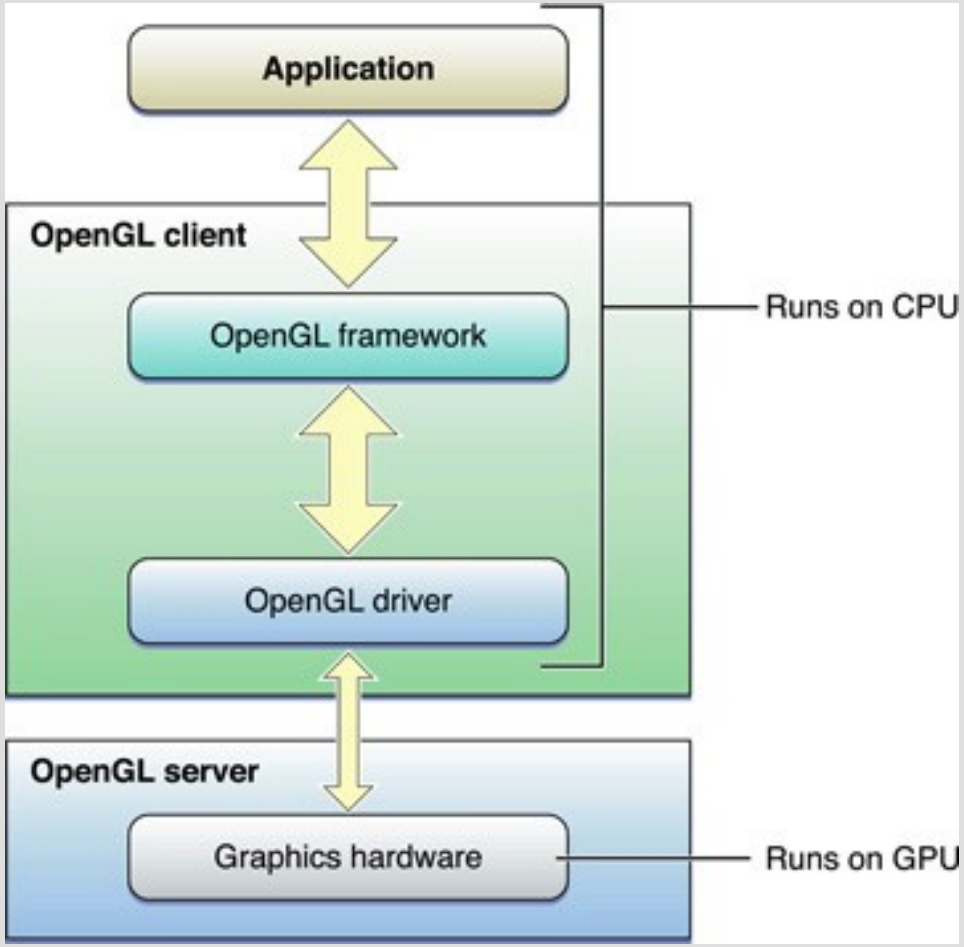
What is OpenGL?

- OpenGL = Open Graphics Library
- Application you can use to access and control the graphics subsystem of the device upon which it runs
- Developed at Silicon Graphics (SGI)
- It is *device independent*
- Cross Platform
  - (Win32, Mac OS X, Unix, Linux)
- Only does 3D Graphics. No Platform Specifics
  - (Windowing, Fonts, Input, GUI)

OpenGL

- OpenGL is a ***software library*** for accessing features in graphics hardware.
- About 500 distinct commands.
  - Not a single function relating to window, screen management, keyboard input, mouse input
- OpenGL uses a ***client-server*** model
  - Client is your application, server is OpenGL implementation on your graphics card/network graphics card
- Default language is **C/C++.**
- To the programmer OpenGL behaves like a **state machine**.
- The actual drawing operations are performed by the underlying *accelerated graphics hardware* (e.g. Nvidia, ATI, SGI etc).

Graphics API Architecture



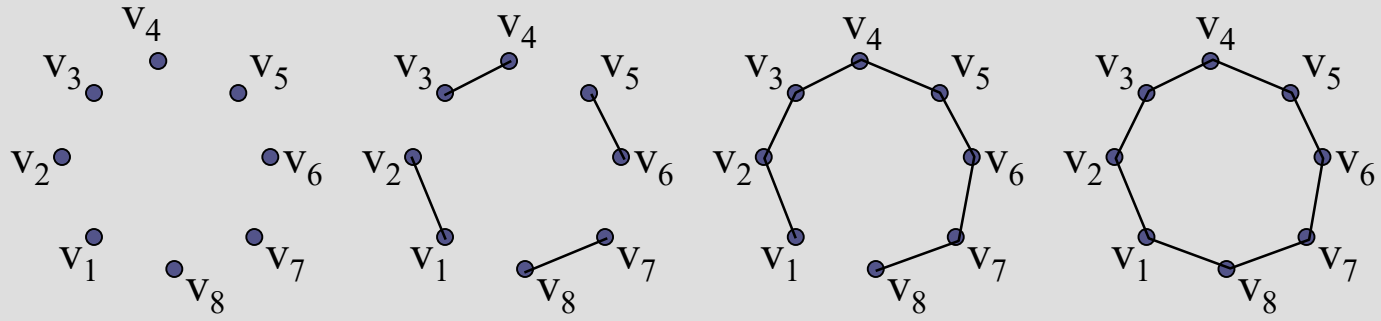
- Set-up & rendering loop run on CPU
- Copy mesh data to **buffers** in graphics hardware memory
- Write **shaders** to draw on the GPU
- CPU command queues drawing on GPU with *this* shader, and *that* mesh data
- CPU & GPU then run **asynchronously**

OpenGL Global State Machine

- Set various aspects of the state machine using the API
  - Colour, lighting, blending
- When rendering, everything drawn is affected by the current settings of the state machine
- Most **parameters are persistent**
  - Values remain unchanged until we explicitly change them through functions that alter the state
- Not uncommon to have **unexpected results** due to having one or more states set incorrectly

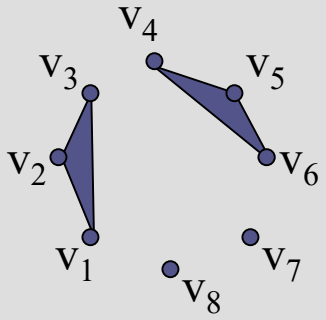
OpenGL Primitives

- All geometric objects in OpenGL are created from a set of basic *primitives*.
- Certain primitives are provided to allow optimisation of geometry for improved rendering speed.
- Line based primitives:

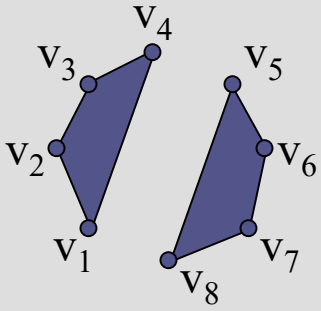


OpenGL® Primitives

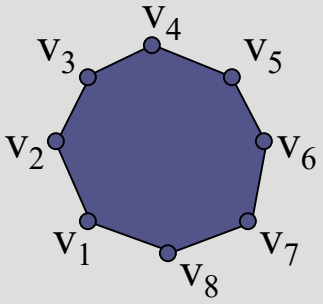
- Polygon primitives



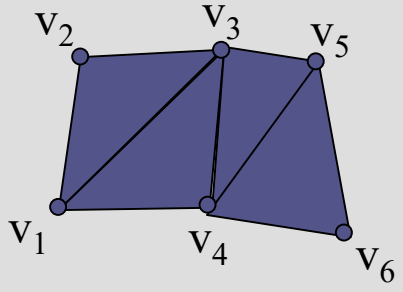
GL_TRIANGLES



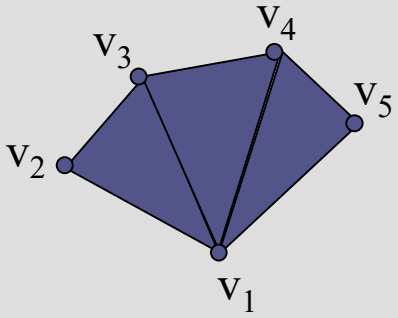
GL_QUADS



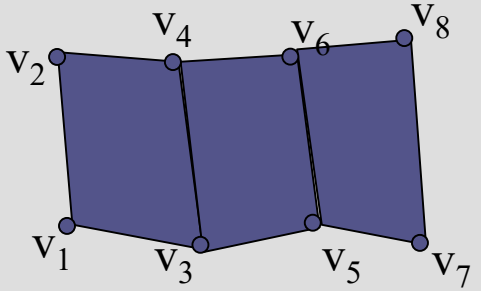
GL_POLYGON



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUAD_STRIP

OpenGL Conventions

- Conventions:
  - all function names begin with **`gl`**, or **`glut`**
    - **`glBegin(…)`**
    - **`glutInitDisplayMode(…)`**
  - constants begin with **`GL_`**, **`GLU_`**, or **`GLUT_`**
    - **`GL_POLYGON`**
  - Function names can encode parameter types, e.g. **`glVertex*:`**
    - **`glVertex2i(1, 3)`**
    - **`glVertex3f(1.0, 3.0, 2.5)`**
    - **`glVertex4fv(array_of_4_floats)`**

http://www.opengl.org/sdk/docs/man/

The Drawing Process

- `ClearTheScreen();`
- `DrawTheScene();`
- `CompleteDrawing();`
- `SwapBuffers();`

- In animation there are usually two buffers. Drawing usually occurs on the background buffer.
- When it is complete, it is brought to the front (swapped). This gives a smooth animation without the viewer seeing the actual drawing taking place. Only the final image is viewed.

- The technique to swap the buffers will depend on which windowing library you are using with OpenGL.

Clearing the Window

- `glClearColor(0.0, 0.0, 0.0, 0.0);`
- `glClear(GL_COLOR_BUFFER_BIT);`

- Typically you will clear the color and depth buffers.

- `glClearColor(0.0, 0.0, 0.0, 0.0);`
- `glClearDepth(0.0);`
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

- You can also clear the accumulation and stencil buffers.
  - `GL_ACCUM_BUFFER_BIT and GL_STENCIL_BUFFER_BIT`

Specifying a Colour

- It is possible to represent almost any colour by adding red, green and blue
- Colour is specified in (R,G,B,A) form [Red, Green, Blue, Alpha], with each value being in the range of 0.0 to 1.0.
  - 0.0 means "all the way off"
  - 1.0 means "all the way on"
- Examples:
  - `(red, green, blue, alpha);`
  - `(0.0, 0.0, 0.0); /* Black */`
  - `(1.0, 0.0, 0.0); /* Red */`
  - `(0.0, 1.0, 0.0); /* Green */`
  - `(1.0, 1.0, 0.0); /* Yellow */`
  - `(1.0, 0.0, 1.0); /* Magenta */`
  - `(1.0, 1.0, 1.0); /* White */`

Colours

- What colour does this represent in OpenGL?
  - `(0.0, 1.0, 0.0);`

Complete Drawing the Scene

- Need to tell OpenGL you have finished drawing your scene.
- **`glFinish();`**

  or

- **`glFlush();`**
- For more information see Chapter of the Red Book:
  - http://fly.srk.fer.hr/~unreal/theredbook/chapter02.html

OpenGL GLUT Overview

- Initialise GLUT and create a window
- GLUT enters event processing loop and gains control of the application
- GLUT waits for an event to occur & then checks for a function to process it
- Tell GLUT which functions it must call for each event

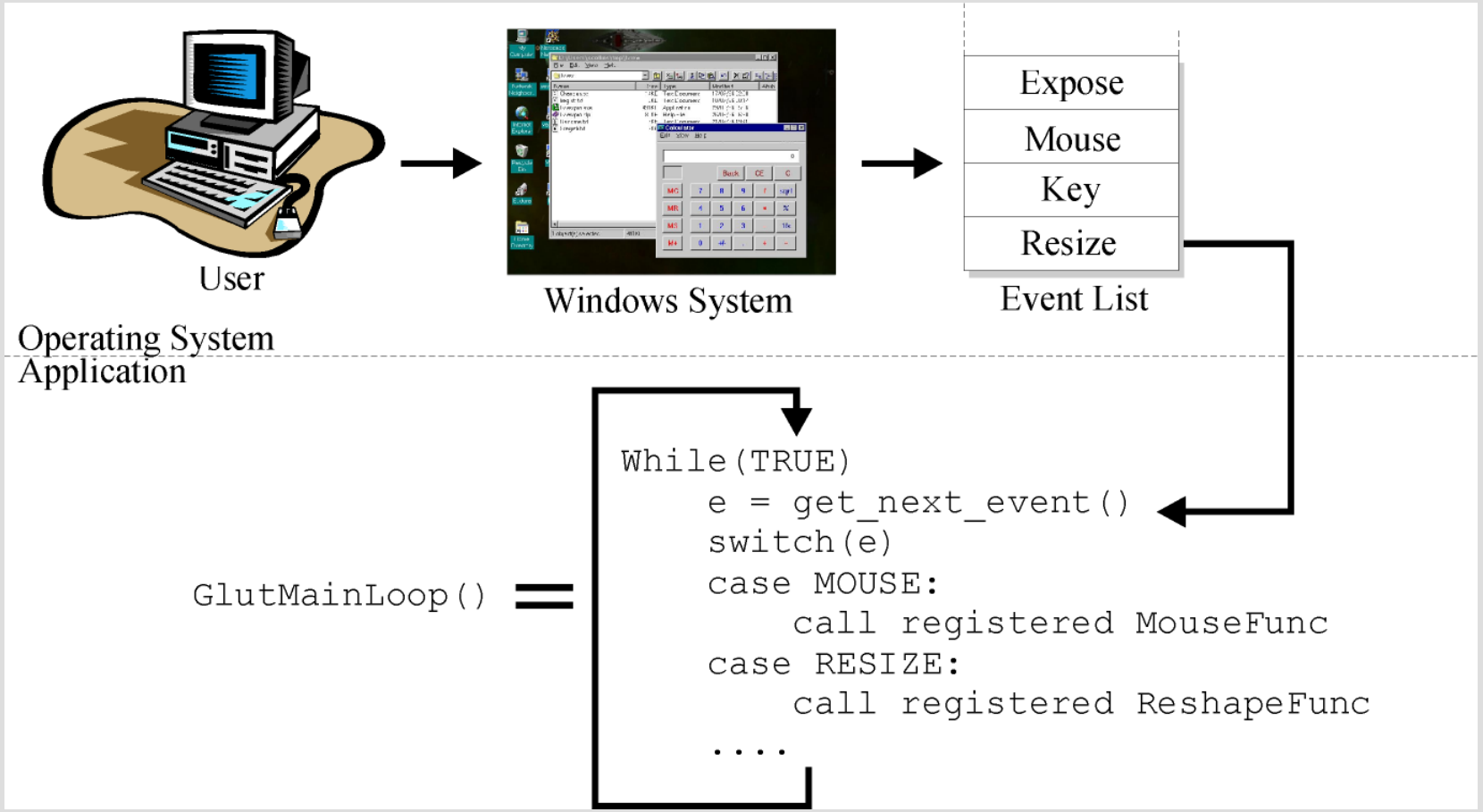
OpenGL GLUT Event Loop

- Interaction with the user is handled through an *event loop*.
- Application registers *handlers* (or *callbacks*) to be associated with particular events:
  - mouse button, mouse motion, timer, resize, redraw
- GLUT provides a wrapper on the X-Windows or Win32 core event loop.
- X-Windows or Win32 manages event creation and passing, GLUT uses them to catch events and then invokes the appropriate callback.
- GLUT is more general than X or Win32 etc.

  ⇒more portable: user interface code need not be changed.

  ⇒less powerful: implements a common subset

OpenGL GLUT Event Loop



OpenGL GLUT Event Loop

- To add handlers for events we call a callback registering function, e.g:

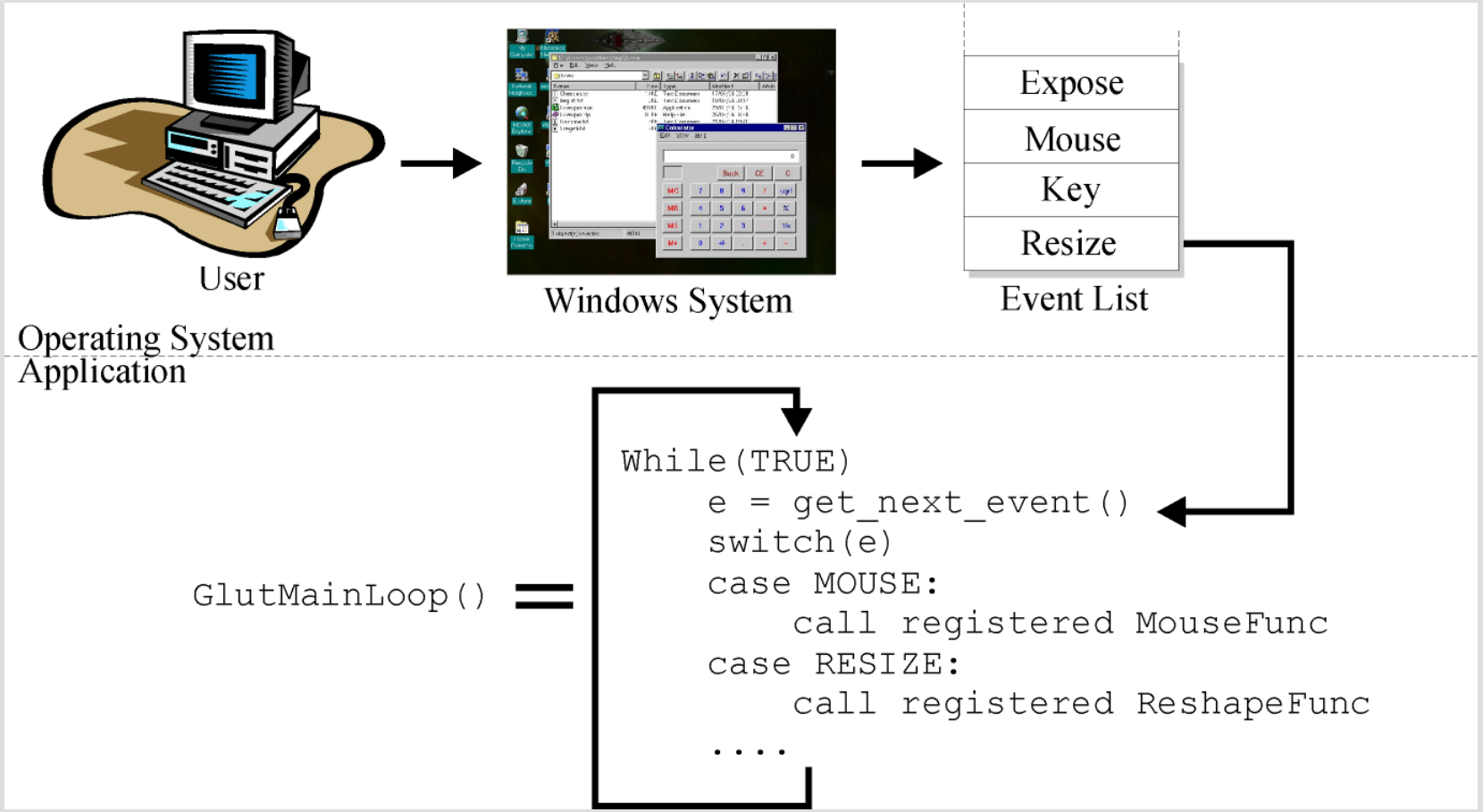
```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
```

- Takes a function (the required callback) as a parameter.
- Handlers must conform to the specification defined.
- Example:

```
void key_handler(unsigned char key, int x, int y);
glutKeyboardFunc(key_handler);
```

- In this case, **`key`** is the ascii code of the key hit and **`(x,y)`** is the mouse position within the window when the key was hit.
- The callback function is *automatically* called when a key is hit.

OpenGL GLUT Event Loop



OpenGL GLUT Event Loop

- To add handlers for events we call a callback registering function, e.g:

```
void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));
```

- Takes a function (the required callback) as a parameter.
- Handlers must conform to the specification defined.
- Example:

```
void key_handler(unsigned char key, int x, int y);

glutKeyboardFunc(key_handler);
```

- In this case, **`key`** is the ascii code of the key hit and **`(x,y)`** is the mouse position within the window when the key was hit.
- The callback function is *automatically* called when a key is hit.

```
//-----------------------------------------------------------------
//
// main
//

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutInitWindowSize(512, 512);
    glutInitContextVersion(4, 3);
    glutInitContextProfile(GLUT_CORE_PROFILE);
    glutCreateWindow(argv[0]);

    if (glewInit()) {
        cerr << "Unable to initialize GLEW ...  exiting" << endl;
        exit(EXIT_FAILURE);
    }

    init();

    glutDisplayFunc(display);

    glutMainLoop();
}
```

Creates a Window using GLUT

Call init()

Event Loop

```
//----------------------------------------------------------------------
//
// init
//

    glGenVertexArrays(NumVAOs, VAOs);
    glBindVertexArray(VAOs[Triangles]);

    GLfloat  vertices[NumVertices][2] = {
        { -0.90, -0.90 },  // Triangle 1
        {  0.85, -0.90 },
        { -0.90,  0.85 },
        {  0.90, -0.85 },  // Triangle 2
        {  0.90,  0.90 },
        { -0.85,  0.90 }
    };

    glGenBuffers(NumBuffers, Buffers);
    glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices, GL_STATIC_DRAW);

    ShaderInfo  shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders(shaders);
    glUseProgram(program);

    glVertexAttribPointer(vPosition, 2, GL_FLOAT,
                          GL_FALSE, 0, BUFFER_OFFSET(0));
    glEnableVertexAttribArray(vPosition);
}
```

Set up your object's initial position

Specify Shaders

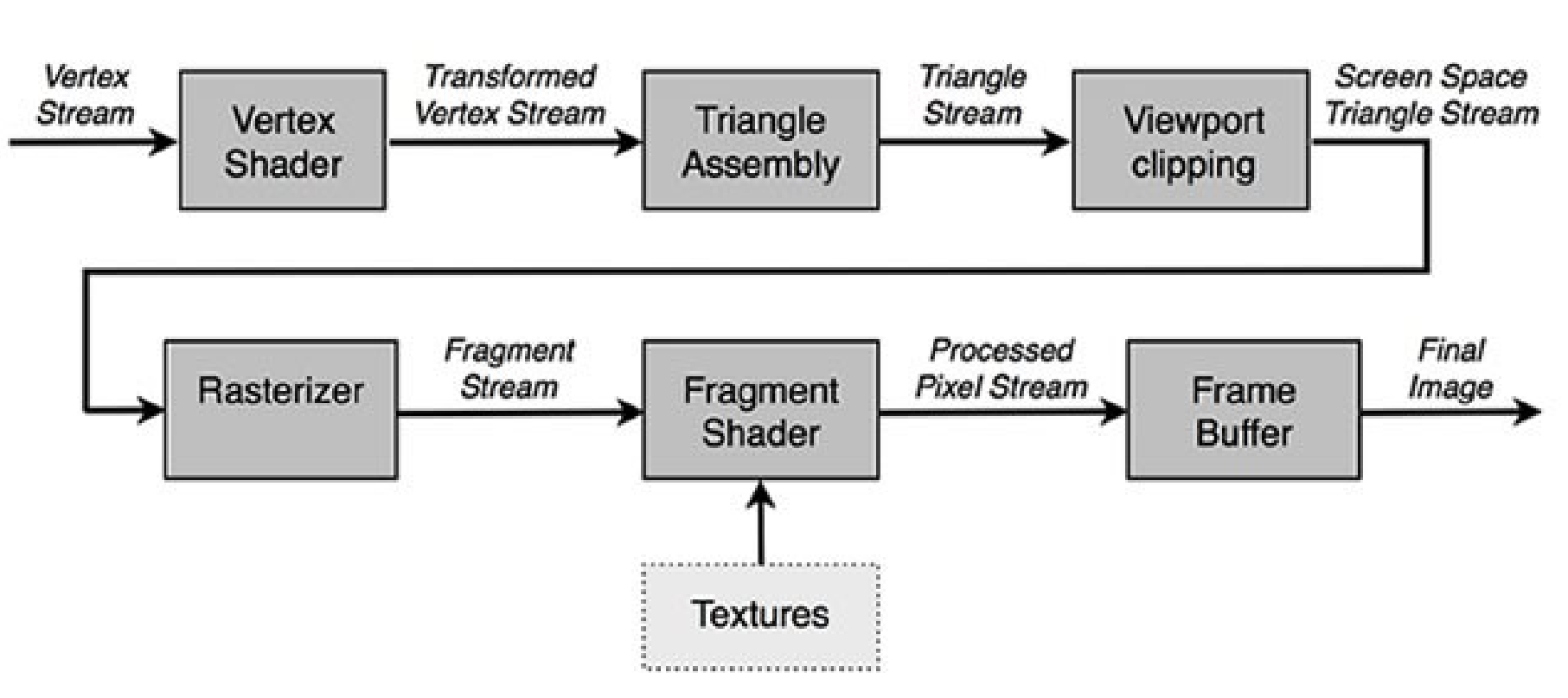
```
//---------------------------------------------------------
//
// display
//

    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(VAOs[Triangles]);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);

    glFlush();
}
```

Request that image is presented on screen

Pick current vertex array

Does the actual Drawing on the Screen

Programmable Pipeline



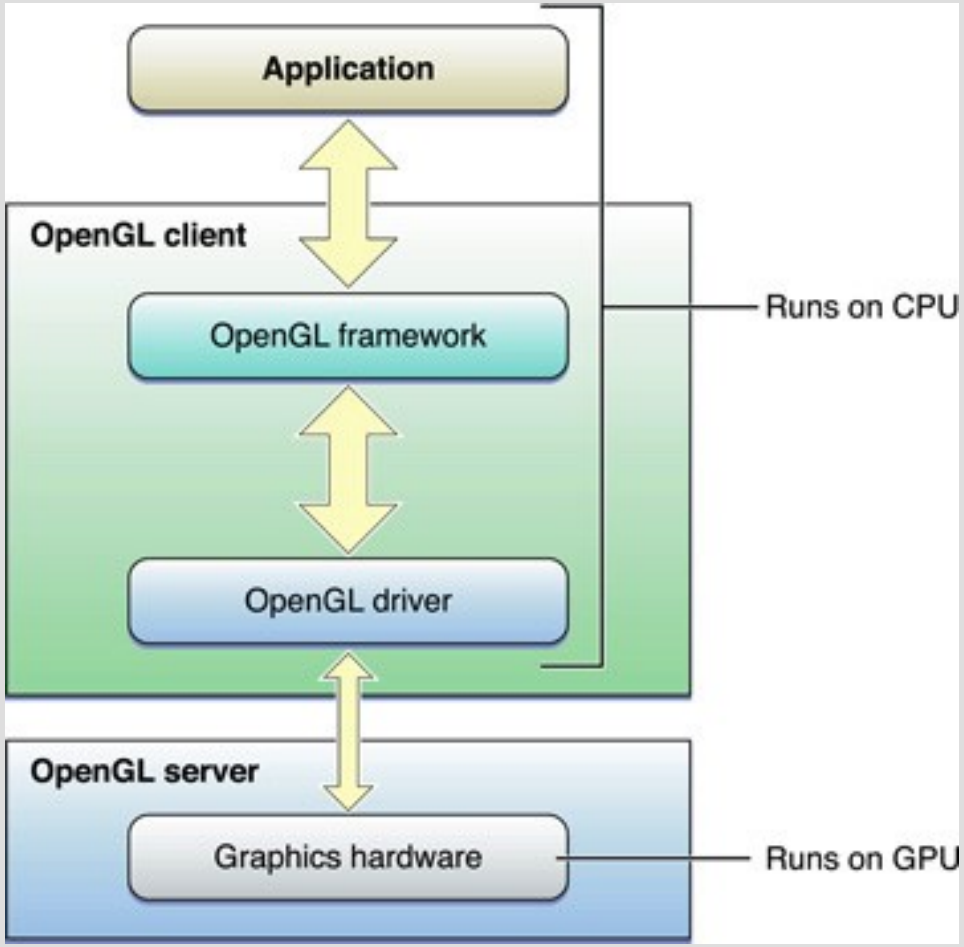
Quick Background

- Hardware has changed!
  - Was “fixed”
  - More of the graphics pipeline is programmable
- OpenGL has changed!
  - Was “fixed”
  - Now shader-based

Overview

- Shaders
- Vertex Buffer Objects

Graphics API Architecture



- Set-up & rendering loop run on CPU
- Copy mesh data to **buffers** in graphics hardware memory
- Write **shaders** to draw on the GPU
- CPU command queues drawing on GPU with *this* shader, and *that* mesh data
- CPU & GPU then run **asynchronously**

Shadertoy

- WebGL tool to experiment with shaders on-the-fly
- implemented entirely in a fragment shader

https://www.shadertoy.com/

Shaders

- A shader is a program with *main* as its entry point
  - Has source code (text file)
  - Cg, HLSL and **GLSL**
  - GLSL is a C-like language
  - Is compiled into a program
  - We get back IDs, which are just ints!

SIGGRAPH University "An Introduction to OpenGL Programming": https://www.youtube.com/watch?v=6-9XFm7XAT8

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, sampleCube`
- C++ Style Constructors
  - `vec3 a = vec3(1.0, 2.0, 3.0);`

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba`, or `stpq` (named components)

- For example:
  - `vec3 v;`
  - `v[1]`, `v.y`, `v.g`, `v.t` – all refer to the same elements

Qualifiers

- `in, out`
  - Copy vertex attributes and other variable info into and out of shaders

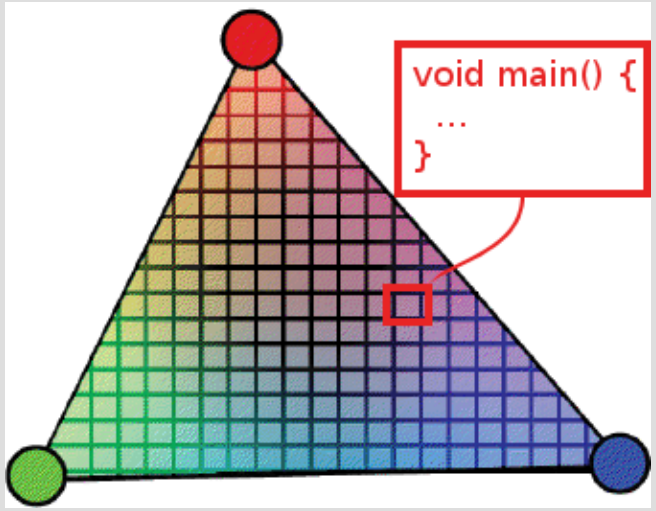
```
in vec2 texCoord;
out vec4 color;
```

- `uniform`
  - Shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Shaders

- Two primary types of shaders:
  - **Vertex shader**
    - Changes the position of a vertex (trans/rot/skew)
    - May determine colour of the vertex
  - **Fragment shader**
    - Determines the colour of a pixel
    - Uses lighting, materials, normals, etc…



Vertex Shader

- Processes vertices
- Data describing what triangles are formed is unavailable to the vertex shader
- At a minimum, a vertex shader must always output vertex location
- Usually transforms vertices into homogeneous clip space

Fragment/Pixel Shader

- Vertex shader outputs become pixel shaders inputs
- A total of 16 vectors can be passed from the vertex to fragment shader
  - E.g., flag to determine which side of a triangle is visible
- Limitation: can only influence the fragment handed it
  - It cannot send its results directly to neighbouring pixels
  - It uses the data interpolated from the vertices along with stored constants and texture data
- Not severe limitation
  - Other ways to affect neighbouring pixels

Making a shader program

- Compile a vertex shader (get an ID)
- Compile a fragment shader (get an ID)
- Check for compilation errors
- Link those two shaders together (get an ID)
  - Keep that ID!
  - Use that ID before you render triangles
  - Can have separate shaders for each model

Examples

```
#version 430
in vec4 vPosition;

void main () {
  // The value of vPosition should be between -1.0 and +1.0
  gl_Position = vPosition;
}
```

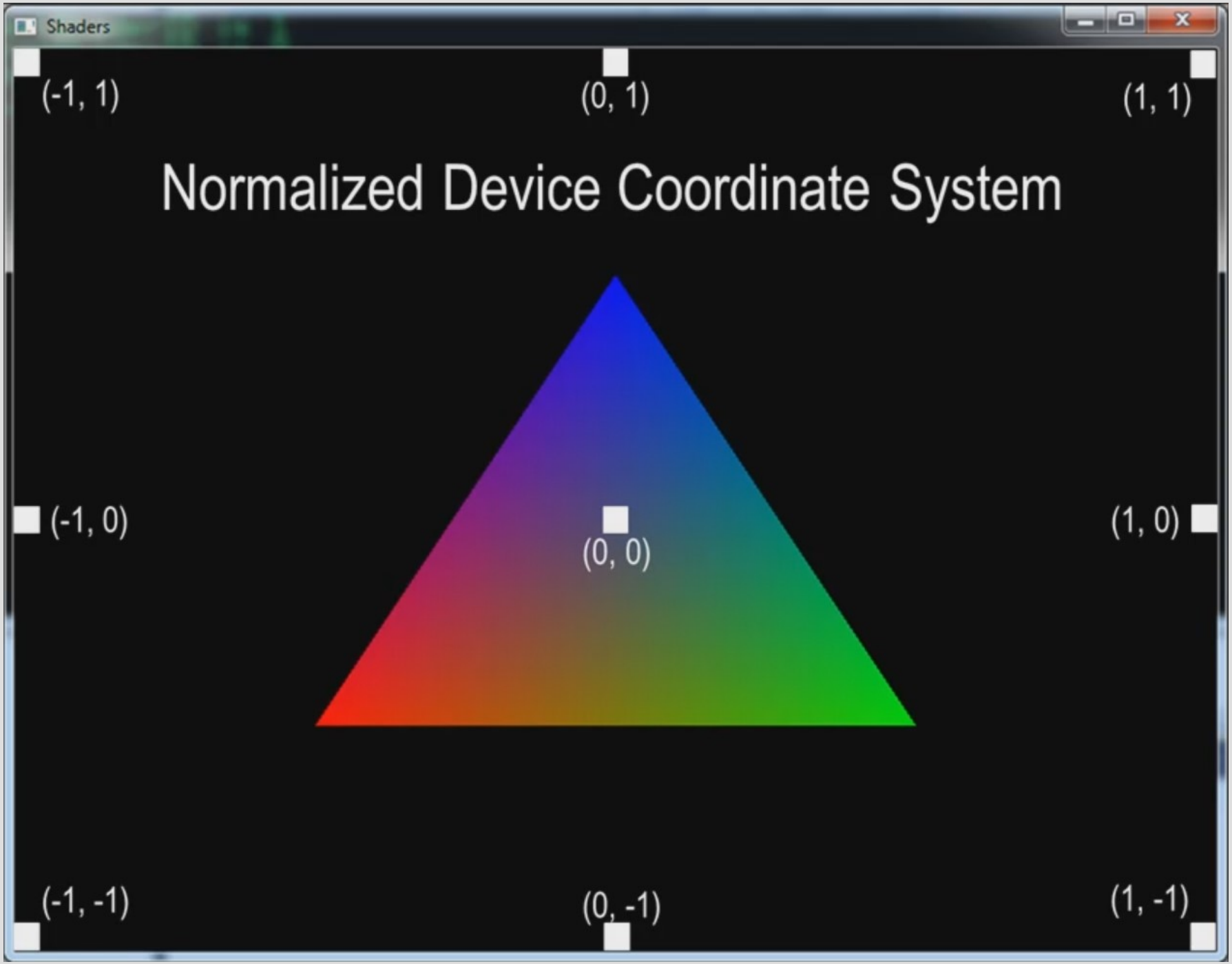
```
out vec4 fColor ;

void main () {
  // No matter what, color the pixel red!
  fColor = vec4 (1.0, 0.0, 0.0, 1.0);
}
```

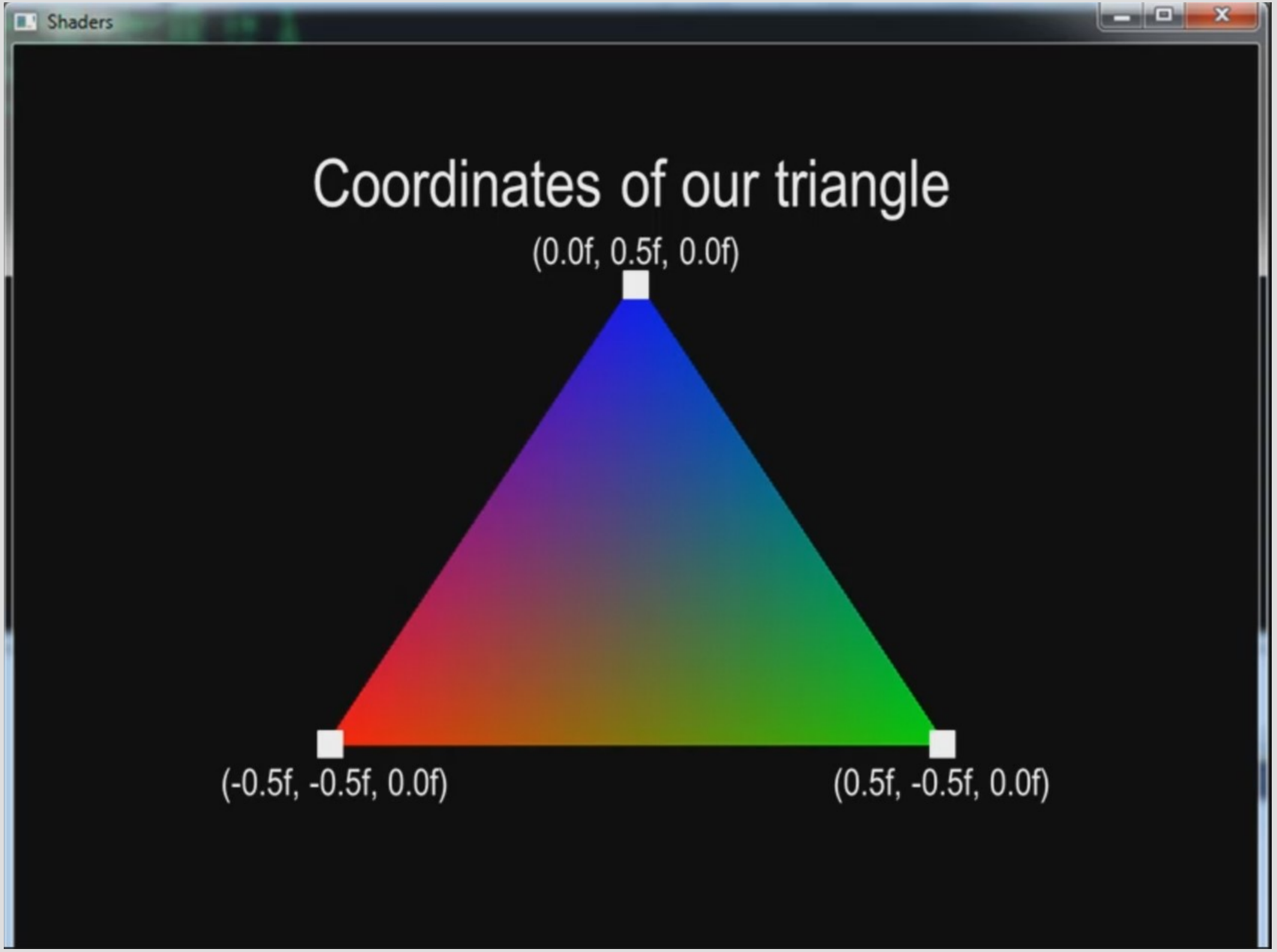
Compiling Shaders

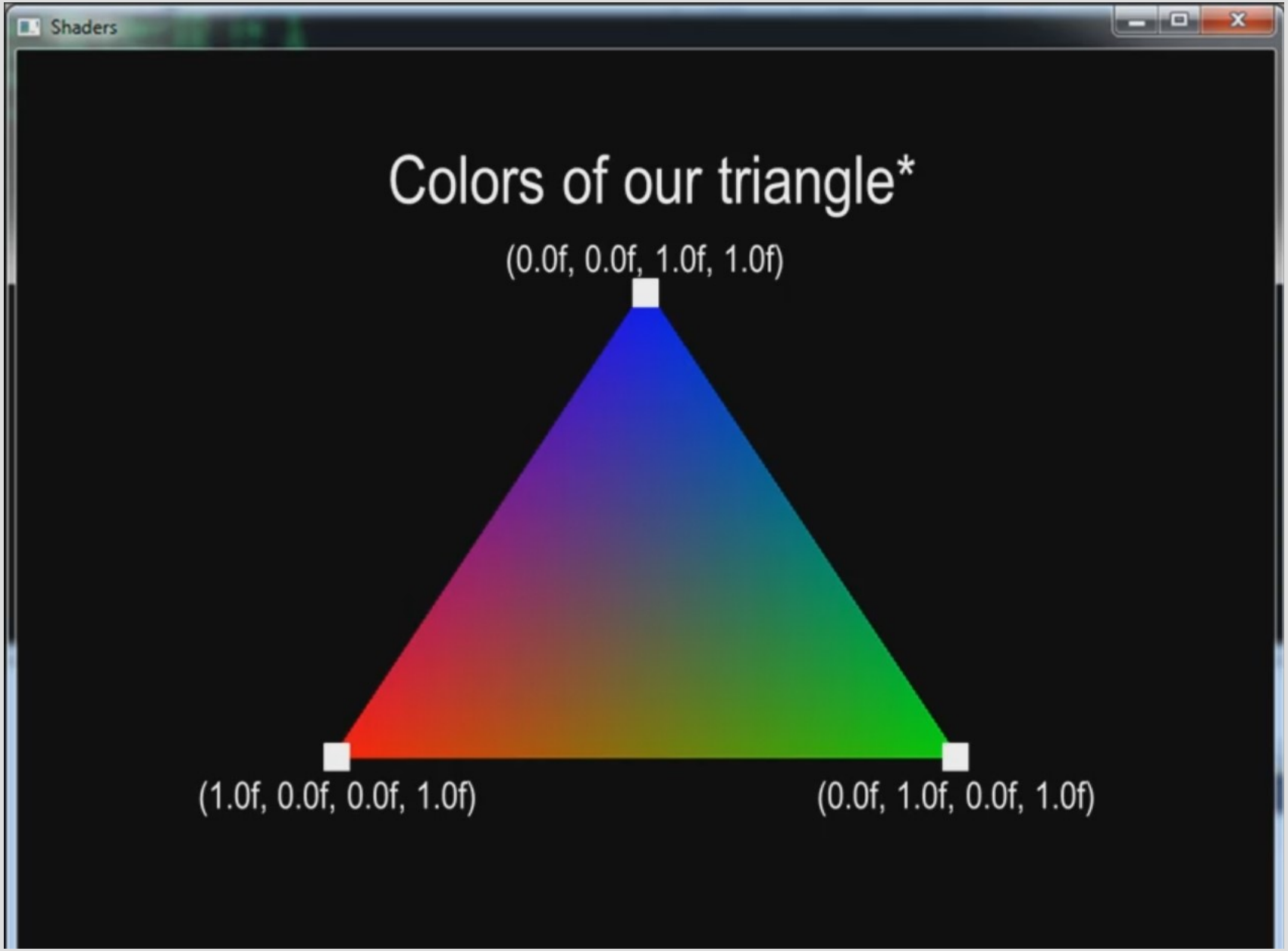
- `<GLuint> glCreateShader (<type>)`
  - Creates an ID (a GLuint) of a shader
  - Example: GLuint ID = glCreateShader(GL_VERTEX_SHADER);
- `glShaderSource (<id>, <count>, <src code>, <lengths>)`
  - Binds the source code to the shader
  - Happens before compilation
- `glCompileShader (<id>)`
  - Used to make the shader program

Creating/Linking/Using Shaders

- `<GLuint> glCreateProgram()`
  - Returns an ID – keep this for the life of the program
- `glAttachShader (<prog ID>, <shader ID>)`
  - Do this for both the vertex and fragment shaders
- `glLinkProgram(<prog ID>)`
  - Actually makes the shader program
- `glUseProgram(<prog ID>)`
  - Use this shader when you're about to draw triangles


Shaders
(-1, 1)
(0, 1)
(1, 1)
Normalized Device Coordinate System
(-1, 0)
(0, 0)
(1, 0)
(-1, -1)
(0, -1)
(1, -1)


Shaders
Coordinates of our triangle
(0.0f, 0.5f, 0.0f)
(-0.5f, -0.5f, 0.0f)
(0.5f, -0.5f, 0.0f)


Shaders
Colors of our triangle*
(0.0f, 0.0f, 1.0f, 1.0f)
(1.0f, 0.0f, 0.0f, 1.0f)
(0.0f, 1.0f, 0.0f, 1.0f)

VERTEX BUFFER OBJECTS

- How do we get the geometry and colour onto the GPU?
- Typically also need a *normal* and *texture coordinate* for each vertex!
- Ask the OpenGL driver to create a buffer object
  - This is just a chunk of memory (e.g. array)
  - Nothing to be afraid of!
  - Located on the GPU (probably)

Working with Buffers

- To create a buffer ID:

```
// This will be the ID of the buffer
GLuint buffer;
// Ask OpenGL to generate exactly 1 unique ID
glGenBuffers(1, &buffer);
```

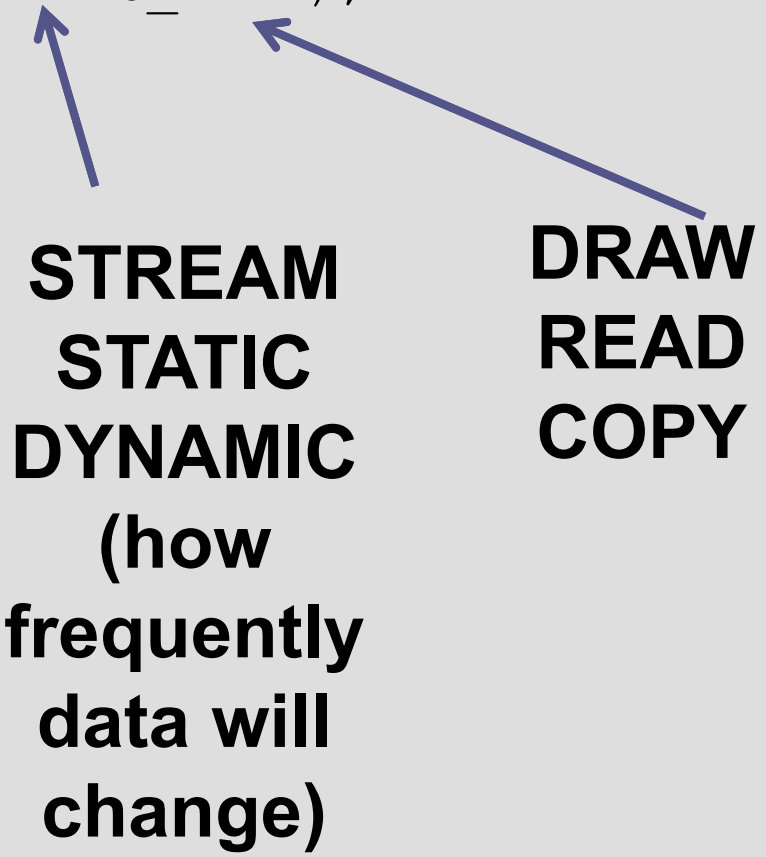
- To set this buffer as the active one and specify which buffer we're referring to:

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

- Notes:
  - That buffer is now bound and active!
  - Any "drawing" will come from that buffer
  - Any "loading" goes into that buffer

Loading the Buffer with Data

- Got some data in an array called "data" and want it to be in the GPU
- `glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);`

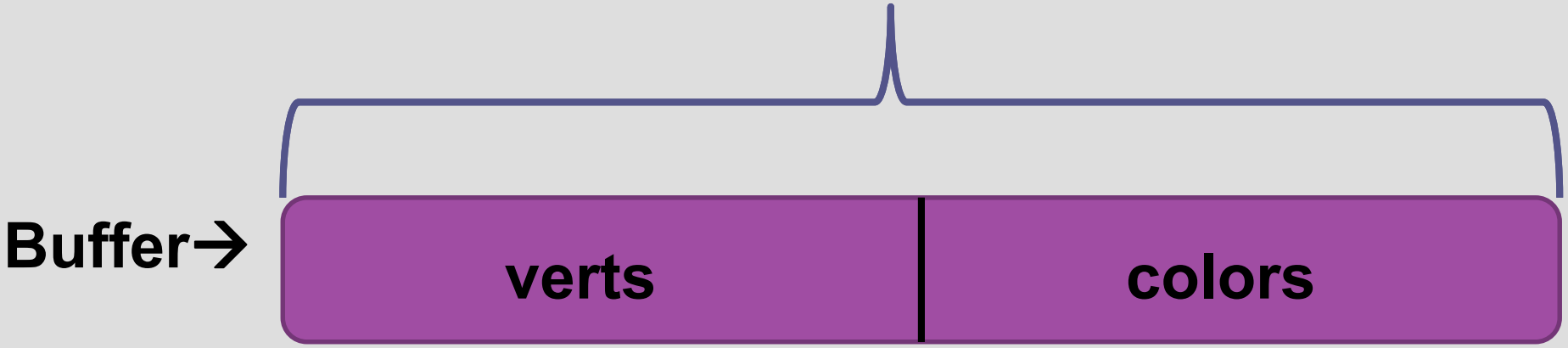


Loading the Buffer with Data

- Process:
  - Create the buffer and pass *no* data
  - Load the geometry
  - Load the colors
  - Load the normals, texture coordinates…
- Can organise buffer however you like

```
• generateObjectBuffer(GLfloat vertices[], GLfloat colors[]) {
      GLuint VBO;
      glGenBuffers(1, &VBO);
      glBindBuffer(GL_ARRAY_BUFFER, VBO);
      glBufferData(GL_ARRAY_BUFFER,
                   numVertices*7*sizeof(GLfloat),
                         NULL, GL_STATIC_DRAW);
      glBufferSubData (GL_ARRAY_BUFFER, 0,
                         numVertices*3*sizeof(GLfloat),
                               vertices);
      glBufferSubData (GL_ARRAY_BUFFER,
                   numVertices*3*sizeof(GLfloat),
                   numVertices*4*sizeof(GLfloat), colors);
```

x,y,z + r,g,b,a



What we have so far..

- We have a buffer with an ID
- That buffer lives on the graphics card
- That buffer is full of vertex position/colour data
- How do we get that info to our shader?

Link to the Shader

- Query the shader program for its variables
- The code below goes into the shader program and gets the "vPosition" ID

```
GLuint vpos;
vpos = glGetAttribLocation (programID, "vPosition");
```

- In OpenGL, we have to enable things (attributes, in this case):

```
glEnableVertexAttribArray(vpos); // turn on vPosition
```

- Finally, Tell those variables where to find their info in the currently bound buffer:

```
glVertexAttribPointer(vpos, 3, GL_FLOAT, GL_FALSE, 0, 0);

  void glVertexAttribPointer(GLuint index, GLint size,
  Glenum type, GLboolean normalized, GLsizei stride,
  const GLvoid* offset);
```


```
vpos = glGetAttribLocation (programID, "vPosition");
glEnableVertexAttribArray(vpos);
glVertexAttribPointer(vpos, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main () {
  gl_Position = s_vPosition;
  Color = vColor;
}
```

Bind that variable to a spot in the buffer

Where to find it in the buffer

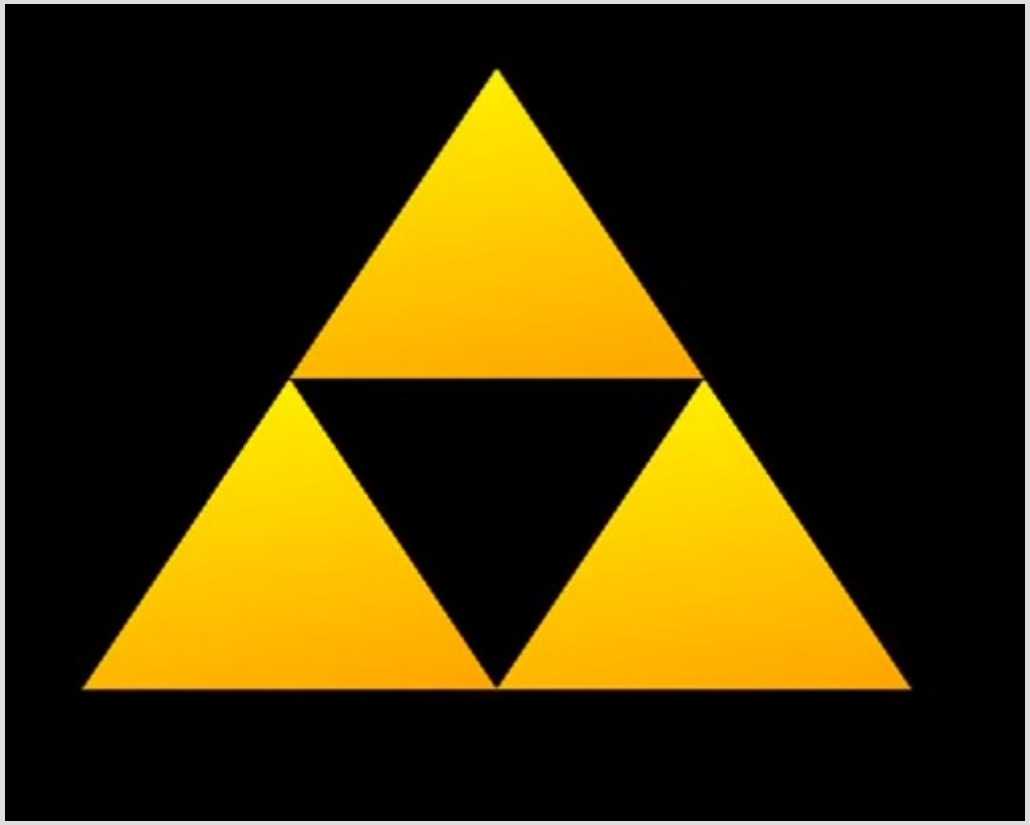

```
cpos = glGetAttribLocation (programID, "vColor");
glEnableVertexAttribArray(cpos);
glVertexAttribPointer(cpos, 4, GL_FLOAT, GL_FALSE, 0,
(BUFFER_OFFSET(numVertices * 3 * sizeof(Glfloat)));
```

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main () {
  gl_Position = s_vPosition;
  Color = vColor;
}
```

Bind that variable to a spot in the buffer

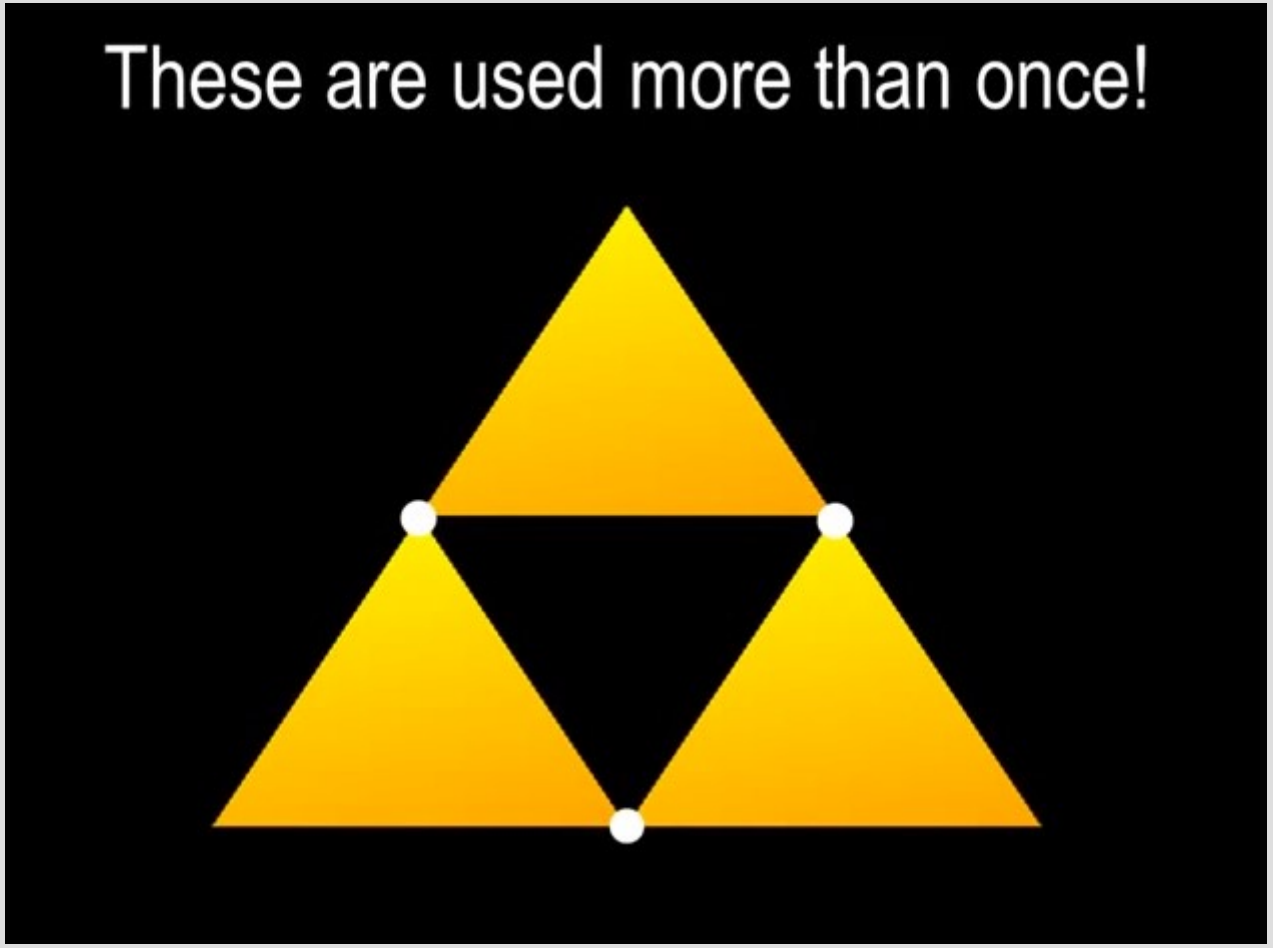
Where to find it in the buffer

Index Buffers



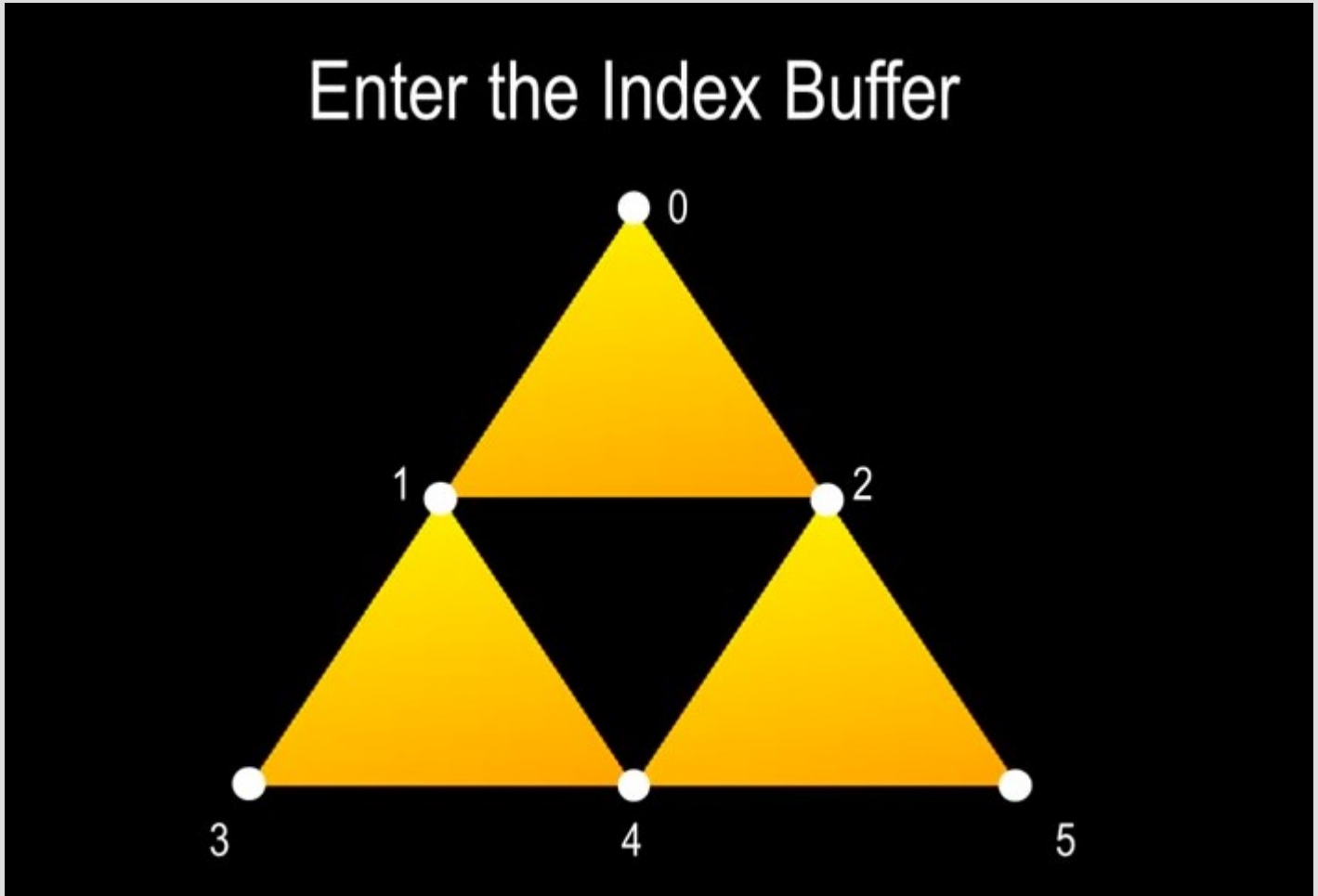
Index Buffers

- Storing 9 verts – not efficient



Index Buffers

- Index buffer - {0, 1, 2, 1, 3, 4, 2, 4, 5}



Code

```
GLuint EBO;
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, indices, GL_STATIC_DRAW);
```

- To render the triangles:

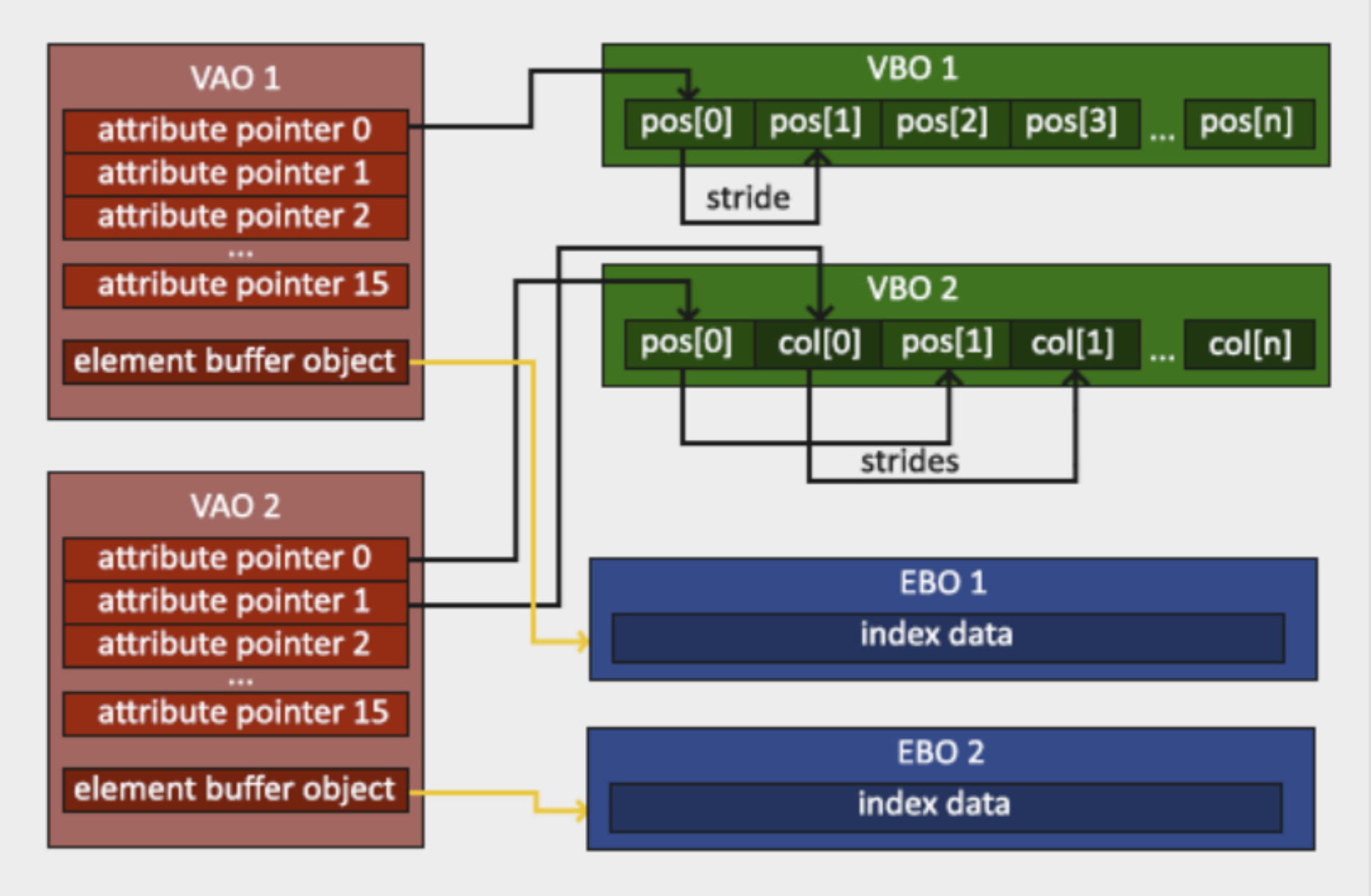
Don't use: `glDrawArrays(GL_TRIANGLES, 0, NUM_VERTICES);`

//Use both the vertex buffer and the index buffer!

```
glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_INT, NULL);
```

VAOs

- VAOs store the data of a geometric object
- Steps in using VAO
  - Generate VAO names by calling glGenVertexArrays()
  - Bind a specific VAO for initialization by calling glBindVertexArray()
  - Update VBOs associated with this VAO
  - Bind VAO for use in rendering
- This approach allows a single function call to specify all the data for objects
  - Previously, you might have needed to make many calls to make all the data current



https://learnopengl.com/Getting-started/Hello-Triangle

Uniforms

- Pass data into a shader that stays the same – is uniform
  - e.g., transformation matrix
- Get data directly from application to shaders
- Two approaches
  - Declare in default block
  - Store in buffer object
- Simply place the keyword **uniform** at beginning of variable definition
  - uniform float fTime
  - uniform mat4 modelMatrix

Using Uniforms to Transform Geometry

- Now it is time to put all our knowledge together and build a program that does a little more than pass vertices through un-transformed

The Old Vertex Shader

```
in vec4 vPosition;

void main () {
  // The value of vPosition should be between -1.0 and +1.0
  gl_Position = vPosition;
}
```

```
out vec4 fColor ;

void main () {
  // No matter what, color the pixel red!
  fColor = vec4 (1.0, 0.0, 0.0, 1.0);
}
```

A Better Vertex Shader

```
in vec4 vPosition; // the vertex in local coordinate system
uniform mat4 mM; // the matrix for the pose of the model
uniform mat4 mV; // The matrix for the pose of the camera
uniform mat4 mP; // The projection matrix (perspective)


void main () {
   // The value of vPosition should be between -1.0 and +1.0
   gl_Position = mP * mV * mM * vPosition;
}
```

Code example – matrix

```
int matrix_location = glGetUniformLocation (shaderProgramID,
"model");
```

Load the data:

```
glUniformMatrix4fv (matrix_location, 1, GL_FALSE, model.m);
```

Shader code: uniform mat4 model;

Code example - float

```
glUseProgram(shaderProgramID);
gScaleLocation = glGetUniformLocation(shaderProgramID,
"gScale");


void display(){

glClear(GL_COLOR_BUFFER_BIT);
static float Scale = 0.0f;
    Scale += 0.001f;
    glUniform1f(gScaleLocation, sinf(Scale));
glDrawArrays(GL_TRIANGLES, 0, 3);
    glutSwapBuffers();
}
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_Triangles, 0, numVertices);
```

- Usually invoked in display callback
- Initiates vertex shader

Resources

- OpenGL Home Page
  - http://www.opengl.org
- Anton's OpenGL Tutors
  - http://antongerdelan.net/opengl/
- Tutorials
  - http://ogldev.atspace.co.uk/
- OpenGL (Programming Guide)
  - http://www.glprogramming.com/
- Excellent OpenGL video tutorials on various topics
  - http://cse.spsu.edu/jchastin/courses/cs4363/lectures/videos/default.htm
  - https://www.youtube.com/watch?v=6-9XFm7XAT8
- Glut Tutorial
  - http://www.lighthouse3d.com/opengl/glut/index.php3?gameglut

Recommended Material

- Read Chapters 1-6 of OpenGL Red Book
- Familiarise yourself with OpenGL Blue Book
- Play with OpenGL Tutorials
- Learn about GLUT