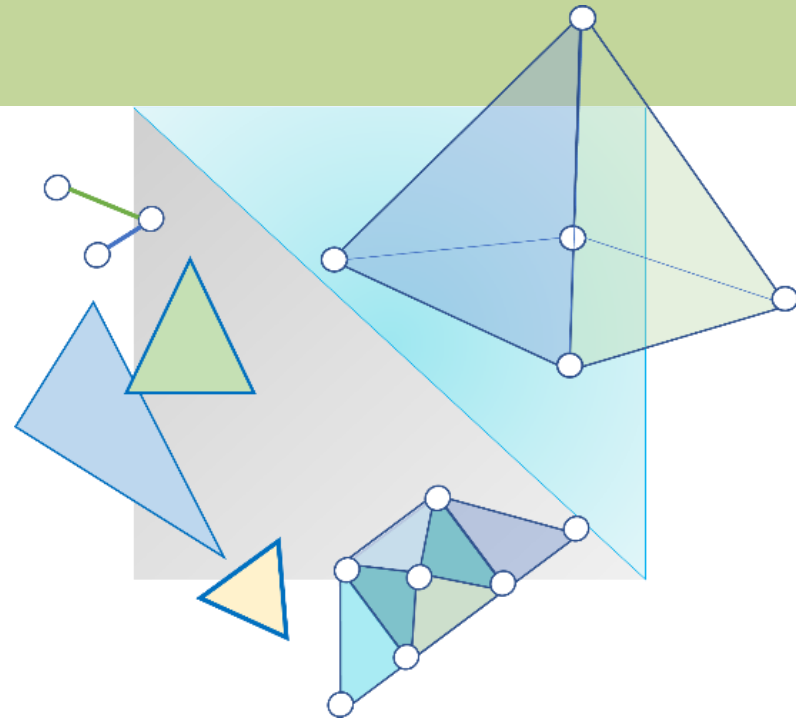


CITS3003 Graphics & Animation

Lecture 2: Programming with OpenGL



Content

- OpenGL Libraries
- OpenGL Architecture
- OpenGL Variable Types and Functions
- A Simple Program

What is OpenGL

- Its an API (specifications to be precise)
 - Allows accessing and dealing with the graphics card
- Where do I download OpenGL?
 - Its already there in your graphics driver
- Is it open source?
 - Irrelevant (its essentially just a specification)
- We still treat OpenGL as API

What is OpenGL (cont...)

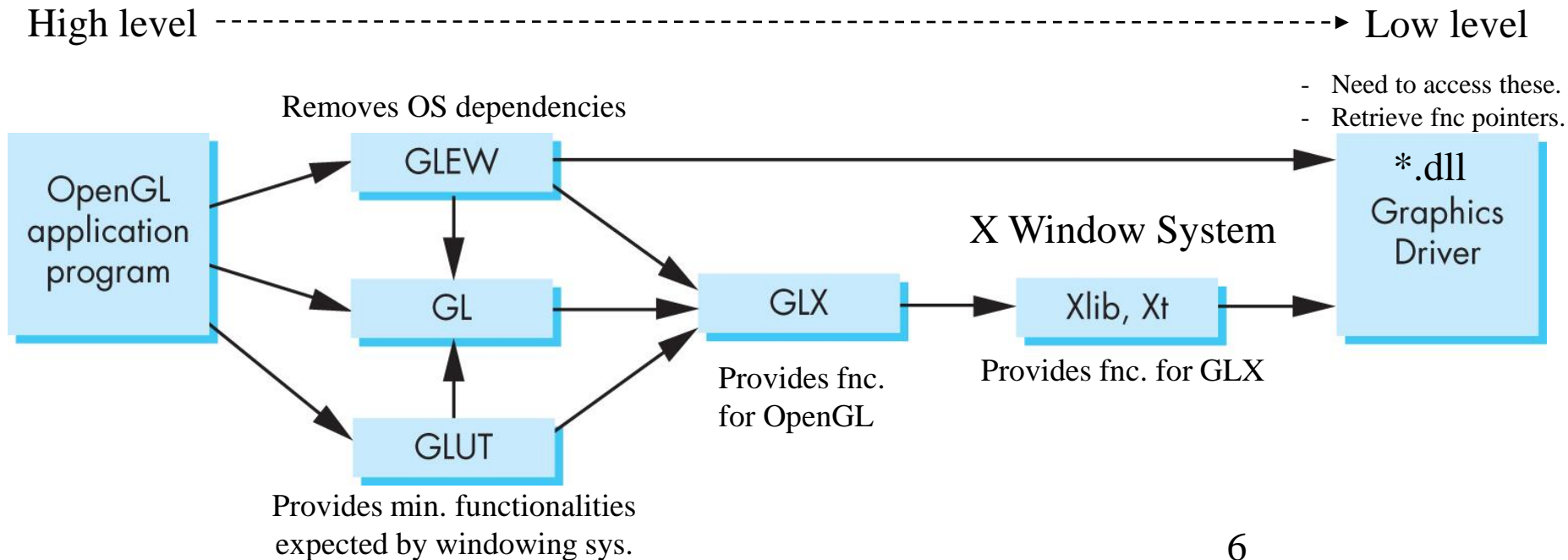
- OpenGL is one of many APIs that allow access to the graphics card
 - E.g. Vulkan, Direct 3D 11, Metal
- Why OpenGL
 - Cross-platform
 - Excellent entry point for Graphics learning

Modern OpenGL

- Legacy OpenGL uses set of pre-sets (simple but not flexible)
- Modern OpenGL Allows the computer program to achieve fast graphics performance by using GPU rather than CPU
- Allows applications to control GPU through programs known as **shaders**
- It is the application's job to send data to GPU; GPU then performs the rendering

Software Organization

- The application programs can use GLEW, GL, GLUT functions but not directly access to Xlib etc.
- The program can therefore be compiled with e.g., GLUT for other operating systems.



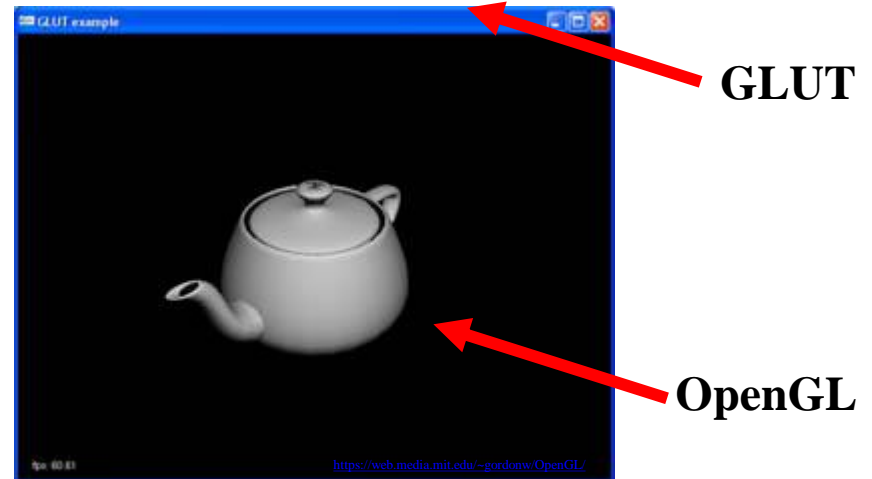
OpenGL/GLUT basics

OpenGL

- OpenGL's function is Rendering (or drawing)
 - Rendering– Convert geometric/mathematical object descriptions into images
- No window management (create, resize, etc)

GLUT

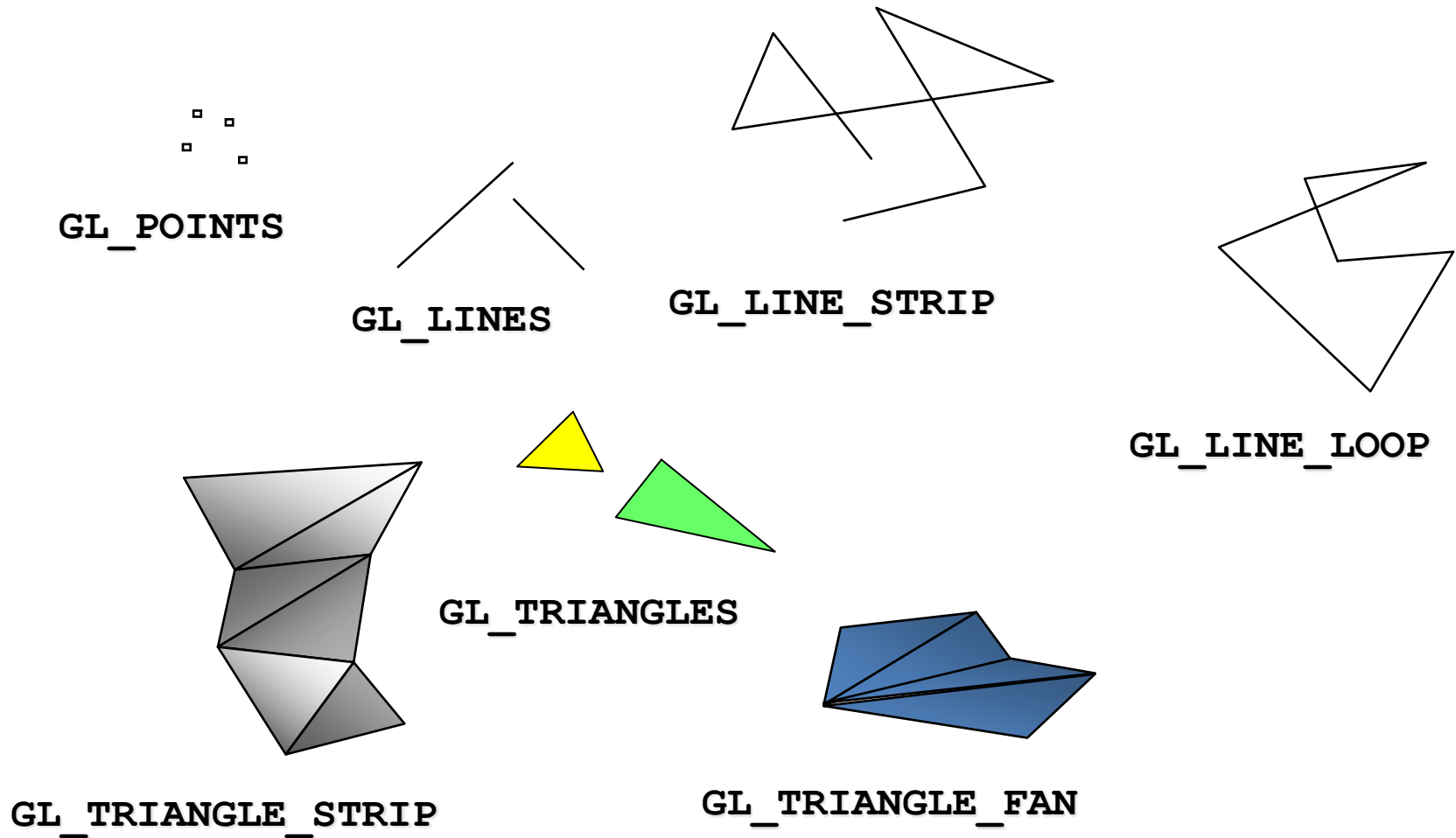
- Minimal window management
- Interfaces with different windowing systems
- Easy porting between windowing systems. Fast prototyping




OpenGL Types

- In OpenGL, we use basic OpenGL types, e.g.,
 - GLfloat,
 - GLdouble,
 - GLint, etc
 - (equivalent to float, double, and int in C/C++)
- Additional data types are supplied in header files `vec.h` and `mat.h` from Angel and Shreiner, e.g.,
 - `vec2`,
 - `vec3`,
 - `mat2`,
 - `mat3`
 - `mat4`, etc.

What are OpenGL Primitives?

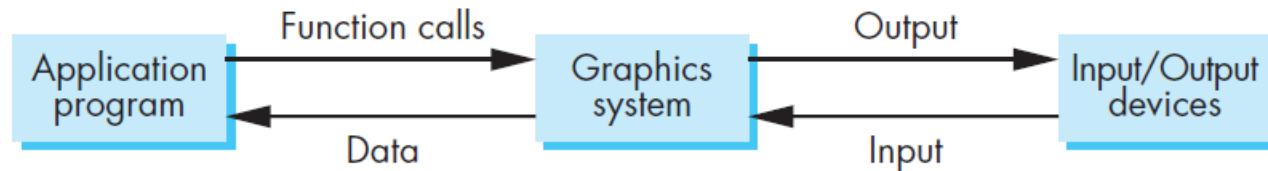


What are Attributes?

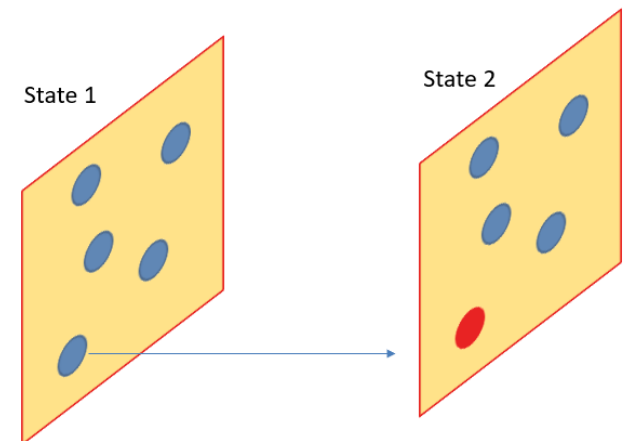
- Attributes are properties associated with the primitives that give them their different appearances, e.g.
 - Color (for points, lines, polygons)
 - Size and width (for points, lines)
 - Stipple pattern (for lines, polygons) 
 - Polygon mode
 - Display as filled: solid color or stipple pattern
 - Display edges
 - Display vertices

State Machine

- We can think of the entire graphics system as a **black box** (finite-state machine).



- This black box has inputs coming from the application program.
- These inputs can change the state of the machine or can cause the machine to produce a visible output.



State Machine (cont..)

- From the perspective of the API, there are two types of graphics functions:
 1. Functions that define primitives that flow through the state machine. These functions:
 - define how vertices are processed (the appearance of primitives are controlled by the state of the machine)
 - can cause output if the primitive is visible
 2. Functions that either change the state inside the machine or return the state information, e.g.,
 - Transformation functions
 - Attribute functions

OpenGL Functions

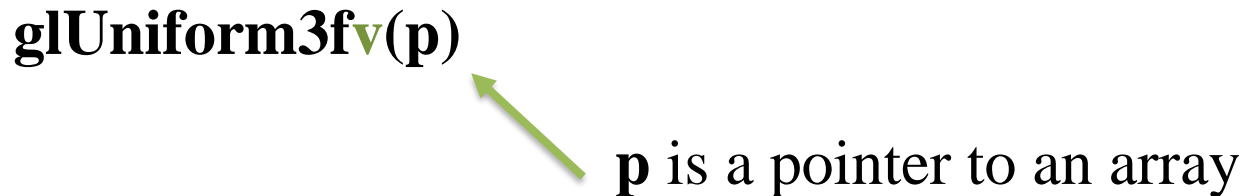
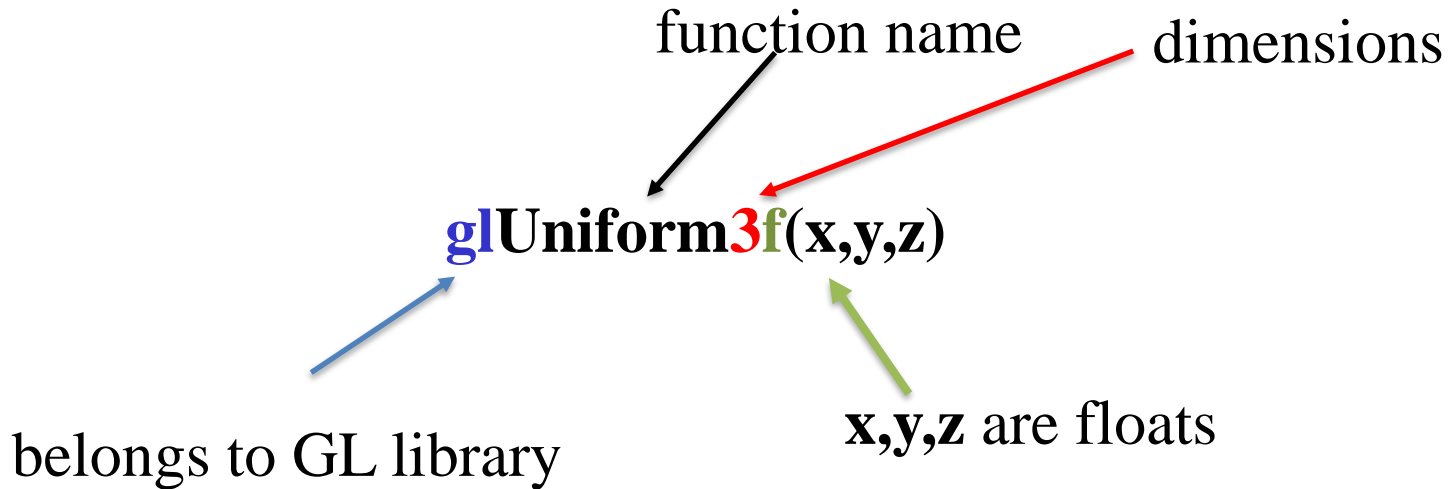
- OpenGL provides a range of functions for specifying:
 - **Primitives** → the low-level objects or atomic entities that our system can display
 - Points
 - Line Segments
 - Triangles
 - **Attributes** → the way that a primitive appears on the display
 - **Transformations** → to carry out transformations of objects, such as rotation, translation, and scaling
 - Viewing
 - Modeling
 - **Control (GLUT)** → to communicate with the window system, initialize our programs, and deal with any errors during the execution
 - **Query** → to get information about API i.e., how many colours are supported etc.,

OpenGL Functions: Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function, e.g., the following are the same function but for different parameter types: (no overloading)
 - **glUniform3f**
 - **glUniform2i**
 - **glUniform3fv**
- The major reason is efficiency (Don't wrap everything in classes when it is not required)

Format of OpenGL Functions

glUniform — Specifies the value of a uniform variable for the current program object



Uniform variables are used to communicate with vertex or fragment shaders from outside. We will come to the details later.

OpenGL #defines

- Most constants are defined in the include files **gl.h**, **glu.h** and **glut.h**
 - Note **#include <GL/glut.h>** should automatically include the others
 - Examples: the functions **glEnable** and **glClear** are both declared in **gl.h**
- The OpenGL data types **GLfloat**, **GLdouble**,.... are also declared in **gl.h**

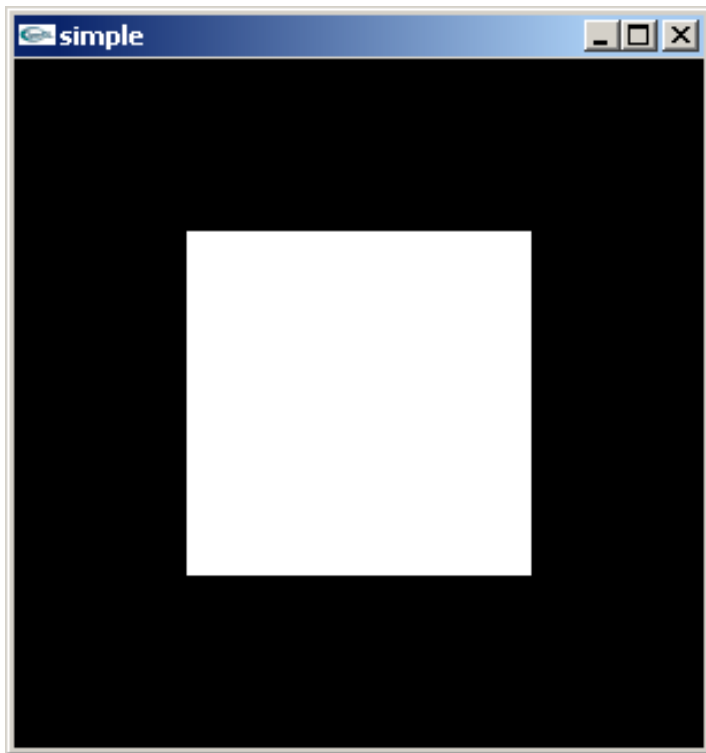
What is GLSL

GLSL is short for **OpenGL Shading Language**

- It is a C-like language with:
 - Built-in Matrix and vector types (2, 3, 4 dimensional)
 - C++ like constructors
- It is similar to Nvidia's Cg and Microsoft HLSL
- Supports loops, if-else constructs, but recursion is not allowed
- GLSL codes are not stand-alone applications, they require an application program that uses OpenGL API
- More on GLSL in later lectures

A Simple Program

`simple.cpp` - Generates a white square on a black background



For the above task, following are the

Rendering Steps:

1. Generate vertices (2 triangle = 6 vertices)
2. Store the vertices into an array
3. Create GPU buffer for vertices
4. Move array of 6 vertices from CPU to GPU buffer
5. Draw 6 points from array on GPU using `glDrawArray`

OpenGL Program

Usually has 3 files:

- **main.cpp file**: containing your main function
 - Does initialization, generates/loads geometry to be drawn
 - **simple.cpp** - Generates a white square on a black background
- **Two shader files**:
 - **Vertex shader**: functions to manipulate (e.g., move) vertices
 - **Fragment shader**: functions to manipulate pixels/fragments (e.g change color)

A Simple Program (cont.)

- Most ‘main.cpp’ (simple.cpp in our case) files have a similar structure that consists of the following functions:
 - **main()**: creates the window, calls the **init()** function, specifies *callback* functions relevant to the application, enters event loop (last executable statement)
 - A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action
 - **init()**: defines the vertices, attributes, etc. of the objects to be rendered, specifies the *shader* programs
 - **display()**: this is a *callback* function that defines what to draw whenever the window is refreshed.

simple.cpp

```
#include <GL/glut.h>
```

← includes headers

```
void init() {  
    // code to be inserted here  
}
```

```
void mydisplay() {  
    // need to fill in this part  
    // and add in shaders  
}
```

```
int main(int argc, char** argv) {  
    // create and open GLUT window;  
    // call init();  
    // register callback function;  
    // wait in glutMainLoop for events;  
}
```

simple.cpp

```
#include <GL/glut.h>
```



includes headers

```
void init() {  
    // code to be inserted here  
}
```

```
void mydisplay() {  
    // need to fill in this part  
    // and add in shaders  
}
```

```
int main(int argc, char** argv) {  
    // create and open GLUT window;  
    // call init();  
    // register callback function;  
    // wait in glutMainLoop for events;  
}
```

OpenGL
programs are
event driven

Event-driven program

- Start at main()
- Initialize
- Wait in infinite loop
 - Wait till defined event occurs
 - Event occurs => Take defined actions

Display and Event Loop

- Note that the program specifies a *display callback* function named **mydisplay**
 - Every glut program must have a display callback
 - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
 - The **main** function ends with the program entering an event loop

simple.cpp – the complete program

```
#include "Angel.h" ← You will be given file Angel.h, which includes vec.h
```

```
using namespace std;
```

```
const int NumTriangles = 2; // 2 triangles to be displayed
```

```
const int NumVertices = 3 * NumTriangles;
```

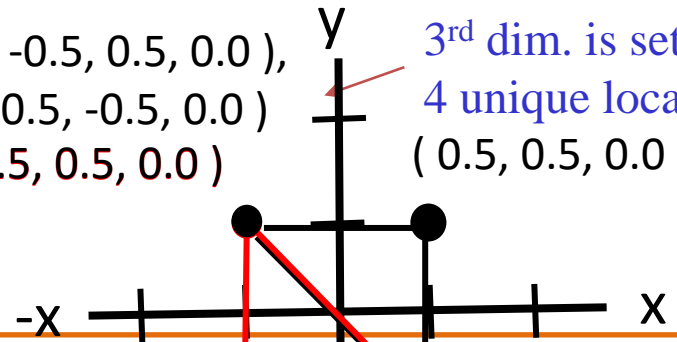
```
vec3 points[NumVertices] = {
```

```
    vec3( -0.5, -0.5, 0.0 ), vec3( 0.5, -0.5, 0.0 ), vec3( -0.5, 0.5, 0.0 ),
```

```
    vec3( 0.5, 0.5, 0.0 ), vec3( -0.5, 0.5, 0.0 ), vec3( 0.5, -0.5, 0.0 )
```

```
}; // generate vertices + store in an array
```

3rd dim. is set to 0,
4 unique locations
(0.5, 0.5, 0.0)
(-0.5, 0.5, 0.0)



```
void init( void )
```

```
{
```

```
    // code to be inserted here}
```

(-0.5, -0.5, 0.0)

(0.5, -0.5, 0.0)

-y 24

simple.cpp – the complete program

Rendering from GPU memory significantly faster. Move data there

GPU memory for data called Vertex Buffer Objects (VBO)

Array of VBOs (called Vertex Array Object (VAO)) usually created

```
void init( void )  
{
```

```
    // First Create a vertex array object
```

```
    GLuint vao;
```

```
    glGenVertexArrays( 1, &vao );
```

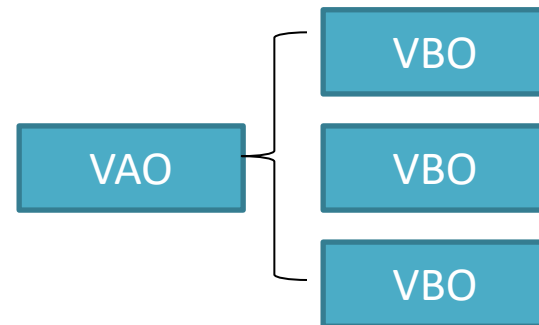
```
    glBindVertexArray( vao ); // make VAO active
```

```
    // Create and initialize a vertex buffer object
```

```
    GLuint buffer;
```

```
    glGenBuffers( 1, &buffer ); // create one buffer object
```

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



Number of buffer objects to return

Data is array of values

simple.cpp – the complete program

```
void init( void )
{
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create and initialize a vertex buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );

    // Move the six points generated earlier to VBO
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );
}
}
```

buffer object data will not be changed.
Specified once by application and used
many times to draw

Data to be transferred to GPU memory (generated earlier)

Need to link names of vertex and
fragment shaders to the main program

Vertex shader: functions to manipulate (e.g., move) vertices
Fragment shader: functions to manipulate pixels/fragments
(e.g change color)

simple.cpp – the complete program

```
void init( void )
{
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create and initialize a vertex buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );

    // Move the six points generated earlier to VBO
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );

    // Load shaders and use the resulting shader program
    GLuint program = InitShader( "vertex.glsl", "fragment.glsl" );
    glUseProgram( program );

    // Initialize the vertex position attribute from the vertex shader
    GLuint vPos = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPos );
    glVertexAttribPointer( vPos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
}
}
```

initShader() connects main program to the shader files

Want to make 6 vertices accessible as variable 'vPosition' in vertex shader

Location of vPosition

3 (x,y,z) floats per vertex

Data no normalized (0-1 range)

Data starts at offset from start of array

simple.cpp – the complete program

```
void init( void )
{
    // Create a vertex array object
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    // Create and initialize a vertex buffer object
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );

    // Move the six points generated earlier to VBO
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );

    // Load shaders and use the resulting shader program
    GLuint program = InitShader( "vertex.glsl", "fragment.glsl" );
    glUseProgram( program );

    // Initialize the vertex position attribute from the vertex shader
    GLuint vPos = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPos );
    glVertexAttribPointer( vPos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    // create black background
    glClearColor( 0.0, 0.0, 0.0, 0.0 ); /* black background */
}
```

simple.cpp – the complete program

```
void display( void )
```

```
{
```

```
    glClear( GL_COLOR_BUFFER_BIT ); // clear screen
```

```
    glDrawArrays( GL_TRIANGLES, 0, NumVertices ); // draw the two triangles
```

```
    glFlush(); // draw it now! // force rendering to show
```

```
}
```

```
glDrawArrays( GL_POINTS, 0, N );
```

Render buffered
data as points

Starting
index

Number of
points to be
rendered

```
int main( int argc, char **argv )
```

```
{
```

```
    glutInit( &argc, argv ); // initialises GLUT
```

```
    glutInitDisplayMode( GLUT_RGBA ); // sets Display mode
```

```
    glutInitWindowSize( 256, 256 ); // sets window size (Width x Height) in pixels
```

```
    glutCreateWindow( "simple" ); // open window with title "simple"
```

```
    init(); // do initializations
```

```
    glutDisplayFunc( display ); // register the callback function
```

```
    glutMainLoop(); //wait in glutMainLoop for events
```

```
}
```

Opening
a
window

No registered callback=no action

Vertex Shader for simple.cpp

Contents of the file **vertex.glsl**

```
attribute vec4 vPosition;  
  
void main()  
{  
    gl_Position = vPosition;  
}
```

Application program is **simple.cpp**. It must work with two shader programs, written in GLSL. They must work together.


Must be the same as the name chosen in **simple.cpp** (see the 2nd parameter passed to the `glGetAttribLocation` function).

Built-in variable name in GLSL, denoting the vertex coordinates in 4-dimensions.

Fragment Shader for simple.cpp

Contents of the file **fragment.glsl**

```
void main()  
{  
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0 );  
}
```



Built-in variable name in GLSL, denoting the colour (as a 4D vector) to be put at that vertex.

Further Reading

“Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL” by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec. 2.4 Primitives and Attributes (up to Sec. 2.4.1)
- Sec. 2.3.1 Graphics Functions
- Sec. 2.3.2 Graphics Pipeline and State Machine
- Sec. 8.10 Graphics and the Internet
- (Advanced) Appendix A.2