CSCD 396

Beginning Graphics

OpenGL Syntax

- Functions in OpenGL begin with the letter “gl”, immediately followed by one or more capitalized words to name the function, i.e. glBindVertexArray(), glUniform2f(), glUniform3fv() etc;

- Some functions begin with “glut”, glutInitWindowSize(), taken from OpenGL Utility Toolkit (GLUT) library;

- Constants are defined upper case separated by underscore, i.e. GL_COLOR_BUFFER_BIT, GL_ARRAY_BUFFER etc.

- Data type starts with uppercase “GL”, i.e, GLfloat, GLint etc.

OpenGL is a State Machine

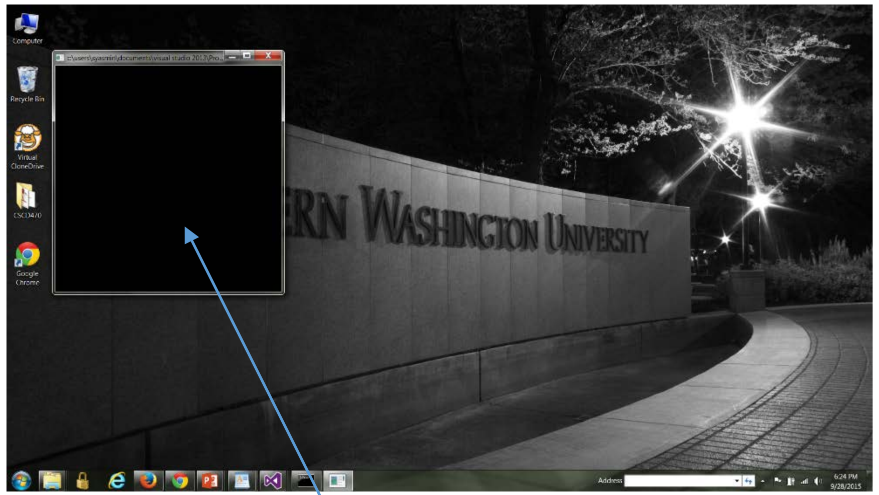
- Most operational features are turned on and off by enabling modes of operation in OpenGL; i.e. **glEnable()** and **glDisable();**

- When some commands are sent to OpenGL, **glFlush** ensures the result to appear on screen by forcing OpenGL to complete;

- **glBind()** allocates memory for a particular object and makes that object current;

A Simple Program that only draws the Window

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

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glFlush();
}

int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(500, 500);
glutInitWindowPosition(100, 100);
glutCreateWindow(argv[0]);
glutDisplayFunc(display);
glutMainLoop();
return 0;
}
```



Interface window and viewport coincides
(Default viewport)

OpenGL Drawing Basics

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

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);

    glutDisplayFunc(display);
    glutMouseFunc(mouse);

    glutMainLoop();

    return 0;
}
```

GLUT initialization and creation of window

Callback functions for different registered event types

Enters into an infinite event processing cycle

Note: Without callback functions and the main loop, only console will be drawn

OpenGL Drawing Basics

- GLUT initialization and creation of window

  - **glutInit():**
    - First GLUT function that the application calls;
    - Initializes the GLUT library;
    - Processes command line arguments provided by the program;

  - **glutInitDisplayMode():**
    - Configures the type of window;
    - Several options like GLUT_RGB, GLUT_RGBA, GLUT_DEPTH, GLUT_SINGLE, GLUT_DOUBLE etc. can be used;
    - This defines color space used by the window (GLUT_RGB, GLUT_RGBA), single or double buffer window (GLUT_SINGLE, GLUT_DOUBLE) , depth buffer enabled (GLUT_DEPTH) etc.

OpenGL Drawing Basics

- In single buffer mode, **glFlush()** is called;
- In double buffer mode, **glutSwapBuffer()** is called;
  - Have two complete color buffers
  - When the drawing of a frame is complete, the two buffers are swapped;
  - Every frame is shown only when the drawing is complete

- glutInitWindowSize();
- glutInitWindowPosition();
- glutCreateWindow();

OpenGL Drawing Basics

- Callback functions are registered in **main** after GLUT initialization;
- Handles different event types:
  - Callback functions will be called by GLUT when the contents of the windows need to be updated;
  - In main, **glutDisplayFunc(display)** sets up the display callback where a pointer to the function **display()** is provided.
  - Function **display()** is used whenever the window requires to be painted.
  - Similarly, glutReshapeFunc(Reshape), glutMouseFunc(Mouse), glutKeyboardFunc(Keyboard) etc.

OpenGL Drawing Basics

- Next we enter into infinite event processing loop, **glutMainLoop()**;

- Works with the window and operating system to process user input, thus waits for the next event to process;

- Renders the current frame;

- determines when window needs to be repainted, calls registered function for this, like **display()**;

- Similarly responds when other registered callback functions are called, **glutReshapeFunc(Reshape)**, **glutKeyboardFunc(Keyboard)**, **glutMouseFunc(Mouse)** etc.;

- No code will be executed after this loop!

What’s next

- Definitely, we don’t just want to draw window, but also the contents inside.
- Here shaders come into play.
- Hence some initialization function needs to be included in main;

```
#include <GL/freeglut.h>
#include <GL/glew.h>

void init(void){….}
void display(void){…}
int main (int argc, char** argv){

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA);
glutInitWindowSize( 512, 512);
glutInitWindowPosition(0, 0);

glutCreateWindow(argv[0]);

if (glewInit()){
cerr << "Unable to initialize GLEW ... exiting" << endl;
}
init();
std::cout << glGetString( GL_VERSION ) << std::endl;

glutDisplayFunc(display);
glutMainLoop();
}
```

OpenGL Basics (Initialization)

- In order to display object on the screen, object needs to be modeled first;
- Most commands follow ‘gen-bind-delete’ sequence

```
void init(void){

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

     GLfloat vertices[NumVertices][2] = {
     { -0.90, -0.90 }, { 0.90, -0.90 },
     { -0.90, 0.90}, { 0.90, 0.90 }};

     glGenBuffers(1, &vbo);
     glBindBuffer(GL_ARRAY_BUFFER, vbo);
     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),vertices, GL_STATIC_DRAW);

     initShaders();

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

OpenGL Basics (‘Init’ function)

```
void init(void){

glGenVertexArrays( 1, &vao);
glBindVertexArray(vao);
:
:
}
```

- A vertex-array-object is allocated first. The following function is called:
    - void **glGenVertexArrays**(GLsizei *n*, GLuint **arrays*):
        - Returns ‘n’ names for use as vertex array objects in the array ‘arrays’;
    - void **glBindVertexArray**(GLuint *array*):
        - When you bind an object for the first time (e.g., the first time glBind*() is called for a particular object name), OpenGL will internally allocate the memory it needs and make that object current;
        - When binds to a previously created vertex-array-object, that vertex-array-object becomes active;
        - When binds to an array value of zero, OpenGL stops using application-allocated vertex array object;
    - void **glDeleteVertexArrays**(GLsizei *n*, GLuint **arrays*):
        - Deletes ‘n’ vertex-array objects specified in ‘arrays’

OpenGL Basics (‘Init’ function)

- Vertex-Buffer objects:
  - A vertex-array-object holds various data related to a collection of vertices;
  - Those data are stored in buffer objects;
  - There can be many types of buffer objects;
  - A buffer object is memory that the OpenGL server allocates and owns;
  - Most data that passed into OpenGL stores data in a buffer object;
  - void **glGenBuffers**(GLsizei n, GLuint *buffers):Returns **n** currently unused names for buffer objects in the array ‘buffers’.

```
void init(void){

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

GLfloat vertices[NumVertices][2] = {
{ -0.90, -0.90 }, { 0.90, -0.90 },
{ -0.90, 0.90}, { 0.90, 0.90 }};

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);
:
:
}
```

OpenGL Basics (‘Init’ function)

- void **glBindBuffer**(GLenum *target*, GLuint *buffer*):
    - target is set to one of the following: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER;
    - Buffer specifies the buffer object to be bound to.
        - A new buffer object is created when called for the first time;
        - When binding to a previously created buffer object, that buffer object becomes the active buffer object;
        - When binding to a buffer value of zero, OpenGL stops using buffer objects for that target.

- void **glDeleteBuffers**(GLsizei *n*, const GLuint **buffers*):
    - Deletes ‘*n’* buffer objects, named by elements in the array ‘*buffers’*.

OpenGL Basics (‘Init’ function)

- **Loading Data into a Buffer Object:**
  - void **glBufferData**(GLenum *target*, GLsizeiptr *size*, const GLvoid **data*, GLenum *usage*)
  - allocates storage for holding the vertex data and copying the data from arrays in the application to the OpenGL server’s memory.
  - ***target*** may be either GL_ARRAY_BUFFER (vertex attribute data), GL_ELEMENT_ARRAY_BUFFER (index data) etc.
  - ***size*** is the amount of storage required for storing the respective data, (number of elements in the data multiplied by their respective storage size)
  - ***data*** is either a pointer to a client memory that is used to initialize the buffer object or NULL. If a valid pointer is passed, size units of storage are copied from the client to the server. If NULL is passed, size units of storage are reserved for use but are left uninitialized;
  - ***usage*** provides a hint as to how the data will be read and written after allocation. Valid values are GL_STATIC_DRAW, GL_DYNAMIC_DRAW etc.

OpenGL Basics(‘Init’ function)

```
void init(void){
:
:
glVertexAttribPointer(0, 2, GL_FLOAT,
GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
}
```

- void **glVertexAttribPointer**(GLuint *index*, GLint *size*, GLenum *type*, GLboolean *normalized*,GLsizei *stride*, const GLvoid **pointer*);
  - *index* specifies shader attribute location
  - *size* represents the number of components per vertex, (i.e.1, 2, 3, 4, or GL_BGRA);
  - *type* specifies the data type (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_FLOAT, or GL_DOUBLE)
  - *normalized* indicates whether vertex data is normalized.
  - *stride* is the byte offset between consecutive elements in the array. If *stride* is zero, the data is assumed to be tightly packed.
  - *pointer* is the offset from the start of the buffer object in ‘bytes’ for the first set of values in the array.

OpenGL Basics(‘Init’ function)

```
void init(void){
  :
  :
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(0);

}
```

- glEnableVertexAttributeArray(Gluint index):
  - turns the vertex attribute array on;
  - takes the ‘index’ of the vertex attribute array pointer;

Shaders in OpenGL

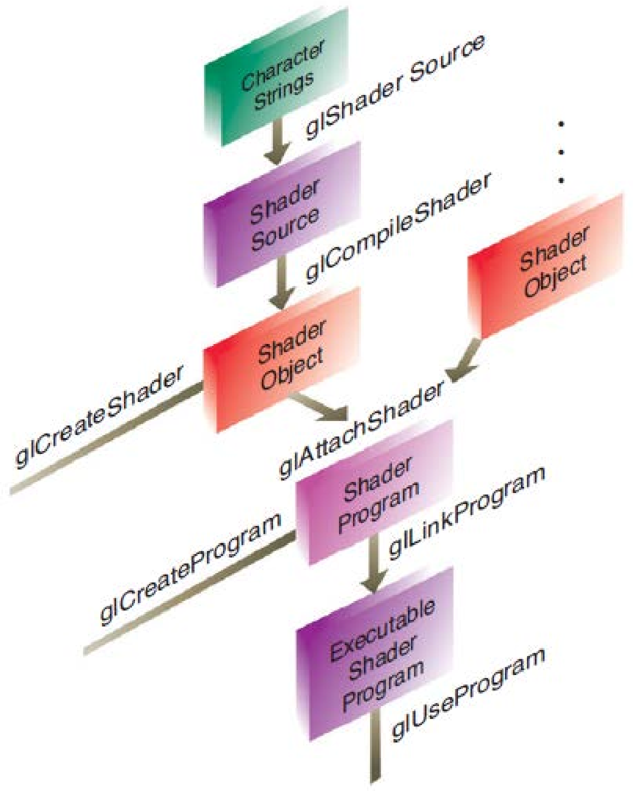


Figure 2.1 Shader-compilation command sequence

```
void init(void){

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

GLfloat vertices[NumVertices][2] = {
{ -0.90, -0.90 }, { 0.90, -0.90 },
{ -0.90, 0.90}, { 0.90, 0.90 }};

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);

initShaders();
:
:
:

}
```

Shaders in OpenGL

- Shaders in OpenGL are considered as special functions that the graphics hardware executes;

- OpenGL includes all the compiler tools to take the source code of the shader and create the code that the GPU needs to execute;

- OpenGL 3.0 and previous versions with “compatibility profile” context, includes a fixed function pipeline that processes geometric and pixel data without shaders;

- From Version 3.1 onwards, the fixed function pipeline has been removed from the “core profile” and use of shaders is mandatory;

- The following function ouputs the OpenGL version being used by your program, i.e., glGetString(GL_VERSION) ;

Shader Pipeline

1. Create a shader program.
2. Attach the appropriate shader objects to the shader program.
3. Link the shader program.
4. Verify that the shader link phase completed successfully.
5. Use the shader for vertex/ fragment processing.

Shader Pipeline

```
GLuint initShaders(char* v_shader, char* f_shader) {

    GLuint p = glCreateProgram();

    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);
    char * vs = ReadFile(v_shader);
    char * fs = ReadFile(f_shader);

    glShaderSource(v, 1, &vs, NULL);
    glShaderSource(f, 1, &fs, NULL);
    free(vs);
    free(fs);
    glCompileShader(v);
    glAttachShader(p, v);
    glAttachShader(p, f);

    glLinkProgram(p);
    glUseProgram(p);
    return p;
}
```

Simple Vertex and Fragment Shaders

430 associated with OpenGL version 4.3
OpenGL's core profile is used
'vPosition' is a regular vertex attribute
'gl_position' is a shader variable

'fColor' is an output qualifier;
Shader will output fragment's color through it

```
#version 430 core
in vec4 vPosition;
void
main()
{
gl_Position = vPosition;
}
```

A simple vertex shader

```
#version 430 core

out vec4 fColor;

void main(){
    fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

A simple fragment shader

‘in’ and ‘out’ variables

- ‘in’ variables in a shader may be
  - vertex attribute for vertex shader
  - output variables from the preceding shader stage
- ‘out’ variables:
  - Output from vertex or fragment shader;
  - Transformed homogeneous coordinates from vertex shader;
  - Final fragment color from fragment shader
- ‘in/out’ block:

```
in Lighting {
       vec3 normal;
      vec2 bumpCoord;
};
```

```
out Lighting {
       vec3 normal;
       vec2 bumpCoord;
};
```

Different Callback Functions

- **glutKeyboardFunc(Keyboard):**
  - sets the keyboard callback for the current window.
  - when a user types into the window, each key press generating an ASCII character will generate a keyboard callback.

Different Callback Functions

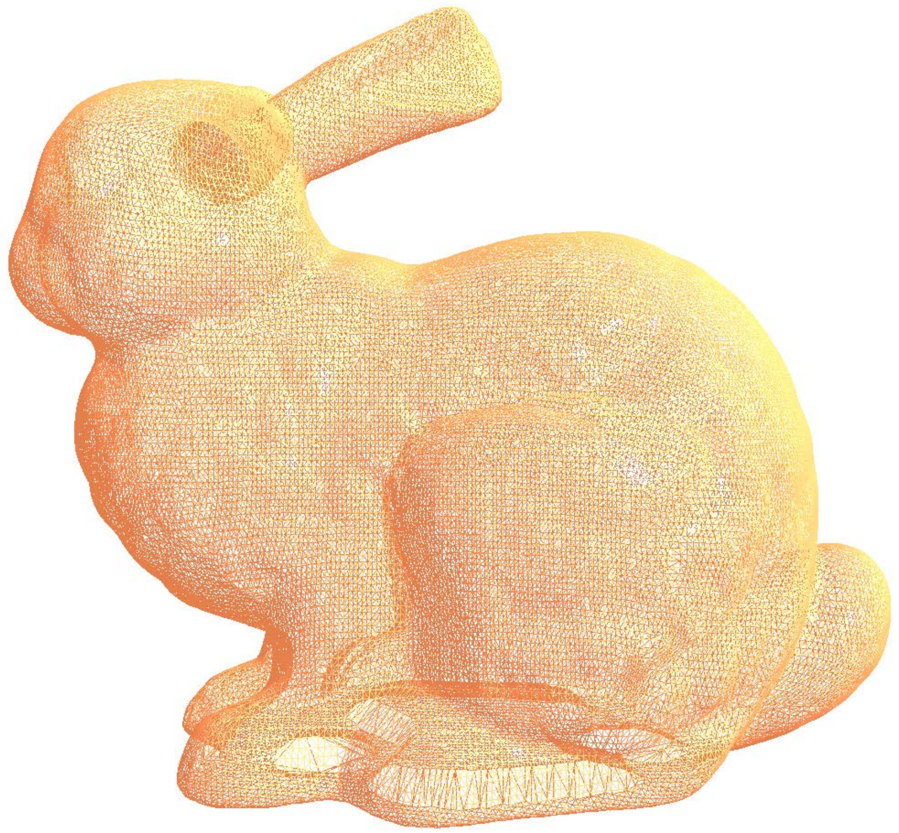
```
void Keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'q':case 'Q':
        exit(EXIT_SUCCESS);
        break;
    case 's':
        show_line = !show_line;
        break;

    case 'u':
        update_vertices = !update_vertices;
        break;
    }
    glutPostRedisplay();
}
```

Polygon and Wireframe modes

- Wireframe mode

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    Can you count how many polygons are there?



glPointSize

- https://www.opengl.org/sdk/docs/man4/html/glPointSize.xhtml
- Default size is 1 pixel
- The range (maximum and minimum) of point size can be found using the following functions:
- GLfloat fSizes[2];
- glGetFloatv(GL_POINT_SIZE_RANGE, fSizes);

- In this instance, the minimum supported point size will be returned in fSizes[0], and the maximum supported size will be stored in fSizes[1].

glPointSize

- // Let the vertex program determine the point size
- glEnable(GL_PROGRAM_POINT_SIZE);

- Now any point size declared in application will be overwritten by the vertex shader.

glLineWidth

- By default, 1 pixel wide;
- The following function gives the maximum and minimum line width;
- glGetFloatv(GL_LINE_WIDTH_RANGE, lSizes);

- You may find the following URL useful

- https://www.khronos.org/opengles/sdk/docs/man/xhtml/glLineWidth.xml