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
Overview

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

\[
\begin{align*}
&(-1, -1, -1) \\
&(1, -1, -1) \\
&(1, 1, -1) \\
&(-1, 1, -1) \\
&(-1, -1, 1) \\
&(1, -1, 1) \\
&(1, 1, 1) \\
&(-1, 1, 1)
\end{align*}
\]
... But not for this!
Modelling Program

- 3ds Max, Maya, Softimage, Blender, Auto CAD etc.
Laser Scanning
A Graphics System

- Input devices
- Central Processing Unit
- Graphics Processing Unit
- Memory
- Frame buffer
- Output devices
What is OpenGL?

- Application you can use to access and control the graphics subsystem of the device upon which it runs
  - High-end PC, video game console, mobile phone..
- Abstraction from underlying hardware
OpenGL Background

- OpenGL = Open Graphics Library
- 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).
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

- Commands are always processed in the order in which they are received
  - Each primitive is drawn completely before any subsequent command takes effect
- Does not let you describe or model complex geometric objects directly
- The commands you issue specify how a certain result should be produced rather than what exactly the result looks like
OpenGL

- OpenGL is interactive and dynamic and therefore we must handle interaction from the user ⇒ event driven model
- The GL library is the core OpenGL system:
  - modeling, viewing, lighting, clipping
- The GLUT library (GL Utility Toolkit) provides the interface with the windowing system.
  - window management, menus, mouse interaction
Geometric Primitives

- APIs typically support a *minimum set* of primitives
- GPUs optimized for points, lines, and triangles
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)`
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 color 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 */
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.cc.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
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

Operating System
Application

User → Windows System → Event List

While(TRUE)
  e = get_next_event()
  switch(e)
  case MOUSE:
    call registered MouseFunc
  case RESIZE:
    call registered ReshapeFunc
  ....

GlutMainLoop()
OpenGL GLUT Event Loop

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

  ```c
  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:**

    ```c
    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 Initialisation

Pass command line arguments to GLUT system

```c
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
glutCreateWindow("RGSquare Application");
glutReshapeWindow(400, 400);
```

Window title

RGB Colour, depth testing and double buffering

Request for window size (not necessarily accepted)
OpenGL GLUT Callback Registration

```c
void reshape(int w, int h) {...}  // window resized
void keyhit(unsigned char c, int x, int y) {...}  // key hit
void idle(void) {...}  // system idle
void motion(int x, int y) {...}  // mouse moved
void mouse(int button, int state, int x, int y) {...}  // mouse button hit
void visibility(int state) {...}  // window (de)iconified
void timer(int value) {...}  // timer elapsed

glutReshapeFunc(reshape);  // window resized
glutKeyboardFunc(keyhit);  // key hit
glutIdleFunc(idle);  // system idle
glutDisplayFunc(draw);  // Paint the window
glutMotionFunc(motion);  // mouse moved
glutMouseFunc(mouse);  // mouse button hit
glutVisibilityFunc(visibility);  // window (de)iconified
glutTimerFunc(timer);  // timer elapsed

glutMainLoop();  // Begin infinite event loop
```
```c
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();
}
```
void init(void)
{
    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);
}
Does the actual Drawing on the Screen

Request that image is presented on screen

Pick current vertex array

```c
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBindVertexArray(VAOs[Triangles]);
  glDrawArrays(GL_TRIANGLES, 0, NumVertices);
  glFlush();
}
```
Programmable Pipeline
Quick Background

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

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

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
Normalized Device Coordinate System
Coordinates of our triangle

(0.0f, 0.5f, 0.0f)

(-0.5f, -0.5f, 0.0f)  (0.5f, -0.5f, 0.0f)
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);`

STREAM
STATIC
DYNAMIC
(how frequently data will change)

DRAW
READ
COPY
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


```c
void 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);
}
```

Buffer

verts
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/color 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

```c
GLuint vpos;
vpos = glGetUniformLocation(programID, "vPosition");
```
- In OpenGL, we have to enable things (attributes, in this case):
  ```c
  glEnableVertexAttribArray(vpos); // turn on vPosition
  
  Finally, we set the location and tell it the format of the data in the buffer:
  ```c
  glVertexAttribPointer(vpos, 3, GL_FLOAT, GL_FALSE, 0, 0);
  ```
Recap Shaders

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

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

in vec4 color;
out vec4 fragColor;

void main () {
    // The color of the pixel is taken from the vertex color!
    fragColor = color;
}
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
- Glut Tutorial
Recommended Material

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