Computer Graphics (CS 543)
Lecture 1 (Part 3): Introduction to OpenGL/GLUT (Part 2)

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Triangulation

- Generally OpenGL breaks polygons down into triangles which are then rendered. Example

```c
glfwDrawArrays(GL_POLYGON, ..)
```
- convex filled polygon
Sierpinski Gasket Program

- Any sequence of points put into array `points[]` will be drawn
- Can generate interesting sequence of points
  - Put in array `points[]`, draw!!
- Sierpinski Gasket: Popular fractal
Sierpinski Gasket

Start with initial triangle with corners \((x_1, y_1, 0), (x_2, y_2, 0)\) and \((x_3, y_3, 0)\)

1. Pick initial point \(p = (x, y, 0)\) at random inside a triangle
2. Select one of 3 vertices at random
3. Find \(q\), halfway between \(p\) and randomly selected vertex
4. Draw dot at \(q\)
5. Replace \(p\) with \(q\)
6. Return to step 2
#include "vec.h"    // include point types and operations
#include <stdlib.h> // includes random number generator

void Sierpinski( )
{
  const int NumPoints = 5000;
  vec2 points[NumPoints];

  // Specify the vertices for a triangle
  vec2 vertices[3] = {
    vec2( -1.0, -1.0 ), vec2( 0.0, 1.0 ), vec2( 1.0, -1.0 )
  };
}
// An arbitrary initial point inside the triangle
points[0] = point2(0.25, 0.50);

// compute and store N-1 new points
for ( int i = 1; i < NumPoints; ++i ) {
    int j = rand() % 3; // pick a vertex at random

    // Compute the point halfway between the selected vertex
    // and the previous point
    points[i] = ( points[i - 1] + vertices[j] ) / 2.0;
}
Lack of Object Orientation

- OpenGL is not object oriented
- Multiple functions for each command
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`
OpenGL function format

- **glUniform3f(x, y, z)**
  - Function name: `glUniform3f`
  - Arguments: `x, y, z` (floats)
  - Belongs to GL library
  - `p` is a pointer to an array

- **glUniform3fv(p)**
  - Number of arguments: 3
  - `p` is a pointer to an array
  - `x, y, z` are floats
Recall: Single Buffering

- If display mode set to single framebuffers
- Any drawing into framebuffer is seen by user. How?
  - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);`
    - Single buffering with RGB colors
- Drawing may not be drawn to screen until call to `glFlush()`

```c
void mydisplay(void) {
    glClear(GL_COLOR_BUFFER_BIT); // clear screen
    glDrawArrays(GL_POINTS, 0, N);
    glFlush();               // Drawing sent to screen
}
```
### Double Buffering

- Set display mode to double buffering (create front and back framebuffers)
  - `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);`
    - Double buffering with RGB colors
- Front buffer displayed on screen, back buffers not displayed
- Drawing into back buffers (not displayed) until swapped in using `glutSwapBuffers();`

```c
void mydisplay(void){
    glClear(GL_COLOR_BUFFER_BIT); // clear screen
    glDrawArrays(GL_POINTS, 0, N);
    glutSwapBuffers();
}
```

Back buffer drawing swapped in, becomes visible here
## OpenGL Data Types

<table>
<thead>
<tr>
<th>C++</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed char</td>
<td>GLByte</td>
</tr>
<tr>
<td>Short</td>
<td>GLShort</td>
</tr>
<tr>
<td>Int</td>
<td>GLInt</td>
</tr>
<tr>
<td>Float</td>
<td>GLFloat</td>
</tr>
<tr>
<td>Double</td>
<td>GLDouble</td>
</tr>
<tr>
<td>Unsigned char</td>
<td>GLubyte</td>
</tr>
<tr>
<td>Unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>Unsigned int</td>
<td>GLuint</td>
</tr>
</tbody>
</table>

**Example:** Integer is 32-bits on 32-bit machine but 64-bits on a 64-bit machine.
Recall: 3. Create GPU Buffer for Vertices

- Already learnt to create off-screen GPU memory for vertex data called 
  *Vertex Buffer Objects*

- Steps:
  1. Create VBO and give it name (unique ID number)

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

    Number of Buffer Objects to return

  2. Make VBO created the currently active one

     ```c
     glBindBuffer(GL_ARRAY_BUFFER, buffer); // data is array
     ```

- May set up VBO in an `init()` function!!
What other Initialization do we Need?

- Also set clear color and other OpenGL parameters
- Also set up shaders as part of initialization
  - Read
  - Compile
  - Link
- Remember: every OpenGL program must now write shaders that our OpenGL program will read in
- Also need two shaders:
  - **Vertex shader**: program that is run once on each vertex
  - **Fragment shader**: program that is run once on each pixel
OpenGL Program: Shader Setup

- OpenGL programs now have 3 parts:
  - Main OpenGL program, vertex shader, fragment shader
  - In main program, specify and link in names of vertex, fragment shader
  - initShader() is homegrown shader initialization

```c
GLuint program = InitShader( "vshader1.glsl", "fshader1.glsl" );
```

initShader()
Homegrown, connects main Program to shader files
More on this later!!
Putting it all Together

● First, we create container called program object

```glsl
Gluint = program;

program = InitShader("vsource.glsl", "fsource.glsl");
glUseProgram(program);
```

● Shader sources are read in, compiled and linked
● During linking, names of all shader variables are bound to indices in tables
● Vertex shader and Fragment shader in same directory as main program
● Main program reads in vertex shader and fragment shader (as strings) and uses them for rendering
Execution Model

Vertex data
Moved to GPU

Application Program → Vertex Shader → Primitive Assembly

glDrawArrays → Vertex
Vertex Shader

- We write a simple “pass-through” shader (does nothing)
- Save to file on disk called vsource.glsl

```glsl
in vec4 vPosition

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

input vertex position

output vertex position
Execution Model

Shader Program

Rasterizer → Fragment Shader

Fragment Rendered vertices → Fragment Shader

Fragment Shader → Frame Buffer

Fragment Color
Fragment Shader

- We write a simple fragment shader (sets color to red)
- Save to file on disk called `fsource.glsl`

```cpp
void main( )
{
    gl_FragColor = vec(1.0, 0.0, 0.0, 1.0);
}
```

Set each drawn fragment color to red
Keyboard Interaction

- Declare prototype
  - `myKeyboard(unsigned int key, int x, int y)`

- Register callback:
  - `glutKeyboardFunc(myKeyboard): when keyboard is pressed`

- Key values:
  - ASCII value of key pressed

- X,Y values:
  - Coordinates of mouse location

- Large `switch` statement to check which key
Example: Keyboard Callback

- Using keyboard to control program?
- 1. register callback in main( ) function
  ```c
  glutKeyboardFunc( myKeyboard );
  ```
- 2. implement keyboard function
  ```c
  void myKeyboard(char key, int x, int y )
  {
      // put keyboard stuff here
      ........
      switch(key){  // check which key
          case 'f':  // do stuff
              break;
          case 'k':  // do other stuff
              break;
          ................
  }
  ```

**Note:** Backspace, delete, escape keys checked using their ASCII codes
Keyboard Interaction

- For function, arrow and other special-purpose keys, use

```c
glutSpecialFunc (specialKeyFcn);
...
Void specialKeyFcn (Glint specialKey, GLint, xMouse, 
                    GLint yMouse)
```

- Example: if (specialKey == GLUT_KEY_F1) // F1 key pressed
  - GLUT_KEY_F1, GLUT_KEY_F12, .... for function keys
  - GLUT_KEY_UP, GLUT_KEY_RIGHT, .... for arrow keys
  - GLUT_KEY_PAGE_DOWN, GLUT_KEY_HOME, .... for page up, home keys

- Complete list of special keys designated in `glut.h`
Mouse Interaction

- Declare prototype
  - myMouse(int button, int state, int x, int y)
  - myMovedMouse

- Register callbacks:
  - glutMouseFunc(myMouse): mouse button pressed
  - glutMotionFunc(myMovedMouse): mouse moves with button pressed
  - glutPassiveMotionFunc(myMovedMouse): mouse moves with no buttons pressed

- Button returned values:
  - GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON

- State returned values:
  - GLUT_UP, GLUT_DOWN

- X,Y returned values:
  - x,y coordinates of mouse location
Mouse Interaction Example

- Each mouse click generates separate events
- Store click points in **global** or **static** variable in mouse function
- **Example**: draw (or select ) rectangle on screen
- Mouse y returned assumes y=0 at top of window
- OpenGL assumes y=0 at bottom of window. Solution? Flip mouse y

```c
void myMouse(int button, int state, int x, int y)
{
    static GLintPoint corner[2];
    static int numCorners = 0;  // initial value is 0
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        corner[numCorners].x = x;
        corner[numCorners].y = screenHeight - y;  //flip y coord
        numCorners++;
    }
}
```

**Screenheight is height of drawing window**
Mouse Interaction Example (continued)

if(numCorners == 2)
{
    // draw rectangle or do whatever you planned to do
    Point3 points[4] = corner[0].x, corner[0].y,
                        corner[1].x, corner[0].y,
                        corner[1].x, corner[1].y,
                        corner[0].x, corner[1].y);

    glDrawArrays(GL_QUADS, 0, 4);

    numCorners == 0;
}
else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    glClear(GL_COLOR_BUFFER_BIT); // clear the window
    glFlush();
}
Menus

- Adding menu that pops up on mouse click

1. Create menu using `glutCreateMenu(myMenu);`

2. Use `glutAddMenuEntry` adds entries to menu

3. Attach menu to mouse button (left, right, middle) using `glutAttachMenu`
Menus

- Example:

```c
void mymenu(int value){
    if(value == 1){
        glClear(GL_COLOR_BUFFER_BIT);
        glFlush( );
    }
    if (value == 2) exit(0);
}
```
GLUT Interaction using other input devices

- Tablet functions (mouse cursor must be in display window)

```c
  glutTabletButton (tabletFcn);
```

```c
  void tabletFcn(Glint tabletButton, Glint action, Glint xTablet, Glint yTablet)
```

- Spaceball functions
- Dial functions
- Picking functions: use your finger
- Menu functions: minimal pop-up windows within your drawing window
- Reference: *Hearn and Baker, 3rd edition (section 20-6)*
References