CS380: Introduction to Computer Graphics
Introduction to OpenGLSL

Min H. Kim
KAIST School of Computing

Code (hw2d.zip)

- we use c++
  – see ‘C++ primer’.
- now we look at the code from the textbook
- Core codes:
  – asst1.cpp
  – glsupport.h/.cpp
  – asst1-gl3.fshader
  – asst1-gl3.vshader
- Auxiliary codes:
  – ppm.h/.cpp

Code (hw2d.zip)

asst1.cpp

- Set g_Gl2Compatible = true to use GLSL 1.0 and g_Gl2Compatible = false to use GLSL 1.3.
- on Mac OS X currently there is no way of using OpenGL 3.x with GLSL 1.3 when GLUT is used.
- If g_Gl2Compatible=true, shaders with -gl2 suffix will be loaded.
- If g_Gl2Compatible=false, shaders with -gl3 suffix will be loaded.
- To complete the assignment you only need to edit the shader files that get loaded

asst1.cpp

- How to check the version you have

```
cout << "GL ver: " << glGetString(GL_VERSION) <<"\n";
cout << "GLEW ver: " << glewGetString(GLEW_VERSION) <<"\n";
cout << "GLSL ver: "<<
glGetString(GL_SHADING_LANGUAGE_VERSION) <<"\n";
```

- Outputs

```
GL ver: 2.1 NVIDIA-8.12.47 310.40.00.05f01
GLEW ver: 1.10.0
GLSL ver: 1.20
```


asst1.cpp

```
int main(int argc, char **argv) {
 try {
     initGlutState(argc,argv);
     glewInit(); // load the OpenGL extensions

     initGLState();
     initShaders();
     initGeometry();
     initTextures();

     glutMainLoop();
     return 0;
 }
 catch (const runtime_error& e) {
     cout << "Exception caught: " << e.what() << endl;
     return -1;
 }
}
```


OpenGL API

- library of routines to control graphics
- calls to compile and load shaders
- calls to load vertex data to vertex buffers
- calls to load textures
- draw calls
- calls to set various options in the pipeline

System issues

- needs include and library files, and installed drivers.
- cross platform
- the alternative would be DirectX graphics – dominant for PC games
  – not cross platform
- we will use 3.0
  we use the glew library to access the API
- Our current codes on Mac:
  - OpenGL 3.2 (GLSL 1.3) doesn't work
  - OpenGL 2.1 (GLSL 1.2) works instead

GLUT

- library of functions to talk with the windowing system
- open up windows
- glut can inform you when some "event" occurs – mousemove, buttonpress, windowresize
- you register callback functions with glut
  - the callback function is called when the event occurs
  - and passed relevant info (ex. the mouse location)
- cross platform (windows/X)

GLSL

- gl shading language
- you write small programs to be executed for each vertex
- you write small programs to be executed for each fragment
- you tell openGL to compile/link/load these "shaders"
  - they operate in parallel on the vertices and fragments (SIMD)
- competitors
  - microsoft's hlsl:
    - dominant for pc games
    - only works with directX
  - nvidia's cg
    - almost identical to hlsl
    - also works with openGL
      - need cgGL library
    - future support unclear.

main()

- initializes lots of stuff
- communicates with openGL API by loading glew.
- hands off all control to glut
  - glut will call back our own functions when needed to do updating and drawing

initGlutState()

- turns on glut
- requests a window with color, depth, and "double buffering"
- registers the names of our callback functions
  - we will look at them soon

initGlutState()

```
static void initGlutState(int argc, char **argv) {
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
    glutInitWindowSize(g_width, g_height);    // create a window
    glutCreateWindow("CS492B: Hello World");    // title the window

    glutDisplayFunc(display);            // display rendering callback
    glutReshapeFunc(reshape);            // window reshape callback
    glutMotionFunc(motion);              // mouse movement callback
    glutMouseFunc(mouse);                // mouse click callback
    glutKeyboardFunc(keyboard);
}
static void reshape(int w, int h) {
 g_width = w;
 g_height = h;
 glViewport(0, 0, w, h);
 glutPostRedisplay();
}
```


Double buffering

- The monitor displays one image at a time
- So if we render the next image to screen, then rendered primitives pop up
- And even worse, we often clear the screen before generating a new image
- A better solution is ”double buffering”

Double buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

initGLState()

- sets some openGL control states
  - background color
  - plumbing for r/w of framebuffer
  - modern color space

```
static void initGLState() {
    glClearColor(128./255,200./255,1,0);
    glEnable(GL_FRAMEBUFFER_SRGB);
}
```

Vertex Buffer Object (VBO)

```
static GLfloat sqVerts[12] = {
   -.5, -.5,
   .5,  .5,
   .5,  -.5,

   -.5, -.5,
   -.5, .5,
   .5,  .5
  };

static GLfloat sqCol[18] = {
   1, 0, 0,
   0, 1, 1,
   0, 0, 1,

   1, 0, 0,
   0, 1, 0,
   0, 1, 1
  };
```


initVBOs()

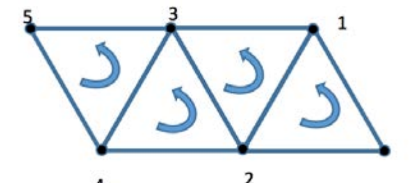
- Transmitting the vertex data from the CPU to the GPU is an expensive process!
- Vertex Buffer Object is to transfer vertex data over to OpenGL.
- in our case, we have attributes for position, color, (and later, texture coordinates)
- when the square geometry is created, the data is passed to the VBOs

initVBOs()

```
static void initVBOs(void) {
    glGenBuffers(1, &sqVertBO); // create a vertex butter
    glBindBuffer(GL_ARRAY_BUFFER, sqVertBO); // bind the buffer to the GL buffer
    glBufferData( // pass data to the GL buffer
        GL_ARRAY_BUFFER,
        12*sizeof(GLfloat),
        sqVerts,
        GL_STATIC_DRAW);

    glGenBuffers(1, &sqColBO)
    glBindBuffer(GL_ARRAY_BUFFER, sqColBO);
    glBufferData(
        GL_ARRAY_BUFFER,
        18*sizeof(GLfloat),
        sqCol,
        GL_STATIC_DRAW);
}
```


initVBOs()

- OpenGL Order: (0, 1, 2), (2, 1, 3), (2, 3, 4), ...
- Consist orientation of the triangles generated.



Display

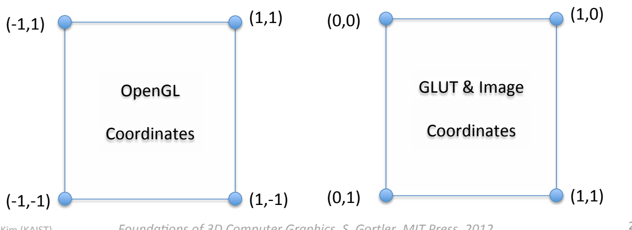
- called by glut when the screen needs updating
- clears screen (you can ignore depth for now)
- sets the program (from the shaderState)
- sets some uniform variables in the shaders (more later)
- draws the geometry

Display()

```
static void display(void) {
      glUseProgram(h_program)
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      drawObj(sqVertBO, sqColBO, 6);
      glutSwapBuffers();
}
void drawObj(const ShaderState& curSS) {
      glBindBuffer(GL_ARRAY_BUFFER, vertbo);
      safe_glVertexAttribPointer2(h_aVertex);
      safe_glEnableVertexAttribArray(h_aVertex);

      glBindBuffer(GL_ARRAY_BUFFER, colbo);
      safe_glVertexAttribPointer3(h_aColor);
      safe_glEnableVertexAttribArray (h_aColor);

      glDrawArrays(GL_TRIANGLES,0,numverts);

      safe_glDisableVertexAttribArray(h_aVertex);
      safe_glDisableVertexAttribArray(h_aColor);
}
```


Vertex Shader

- Typical use of the v-shader:
  - to determine the final position of the vertices on the screen
- Input: the attribute variables (position, color, text. coor.)
- Output: the reserved output variable, gl_Position
- The x and y coordinates of this variable are interpolated as positions within the drawing window.



Vertex Shader

```
#version 130

uniform float uVertexScale;

in vec2 aPosition;
in vec3 aColor;
in vec2 aTexCoord0, aTexCoord1;

out vec3 vColor;
out vec2 vTexCoord0, vTexCoord1;

void main() {
  gl_Position = vec4(aPosition.x * uVertexScale, aPosition.y, 0,1);
  vColor = aColor;
  vTexCoord0 = aTexCoord0;
  vTexCoord1 = aTexCoord1;
}
```

Fragment Shader

- After rasterization (it's not programmable)
- Input: interpolated data of varying variables
- Output: color values in the framebuffer (a part of GPU memory)
- Control the material and geometric properties of the material at that pixel

Fragment Shader

```
#version 130

uniform float uVertexScale;
uniform sampler2D uTexUnit0, uTexUnit1;

in vec2 vTexCoord0, vTexCoord1;
in vec3 vColor;

out vec4 fragColor;

void main(void) {
 vec4 color = vec4(vColor.x, vColor.y, vColor.z, 1);
 vec4 texColor0 = texture(uTexUnit0, vTexCoord0);
 vec4 texColor1 = texture(uTexUnit1, vTexCoord1);

 float lerper = clamp(.5 *uVertexScale, 0., 1.);
 float lerper2 = clamp(.5 * uVertexScale + 1.0, 0.0, 1.0);

 fragColor = ((lerper)*texColor1 + (1.0-lerper)*texColor0) * lerper2 + color * (1.0-lerper2);
}
```

As varying variables, the texture coordinates are interpolated from their vertex values to appropriate values at each pixel.

Texture

- auxiliary image data
- read from a file, loaded to openGL, used in fragment shader
- initTextures
- glTexture is a wrapper around a texture handle
- loadTexture
- – reads from file, turns on any "texture unit", turns on a texture, passes the data.
- binds each texture to a unit (we have 2 here)
- we will send the "texture unit" info as a uniform (see display)

Reshape

- called by glut when the window size changes.
- we tell openGL of the new size
- we store this info for our own use
- then we call glutPostRedisplay so that glut will know to trigger a display callback.

Interaction

- we will use mouse motion to change the global g object
- this will be sent to the uniform uVertexScale
- this will be used in the vertex shader to change the x coordinate of the vertices

Interaction: Mouse

- callback
- called (due to our registration) whenever the mouse
- is clicked down or up
- we store this information
  – we need to flip the y coordinate from glut

Interaction: Motion

- callback
- called whenever the mouse is moved
- here is where we update g objectScale
- then we call glutPostRedisplay so that glut will know to trigger a display callback.
- see display for use of scale
- see vertex shader for use of scale