# 1: Get the project up and running

* Follow the instructions found in the ‘Handout-Unofficial-OpenGL’ document to build ‘glsdk’
* Create a new empty C++ Win32 project, call it ‘Lab05’
* Download ’06-ShaderManagerWithProjection.zip’ from D2L and extract it into the project folder (Do this in Windows).
* Add all the C++ and ‘glsl’ files to the project.
* Set up the OpenGL ‘include’ and ‘library’ paths
* Set up the ‘GLM’ include path: <path>\glsdk\_0\_5\_2\glm
* Set up the ‘glutil’ include path: <path>\glsdk\_0\_5\_2\glutil\include
* Set up the ‘glutil’ library path: <path>\glsdk\_0\_5\_2\glutil\lib
* Set up the ‘boost’ include path: <path>\glsdk\_0\_5\_2\boost
* Build and then run the application. You should see the following output:

|  |
| --- |
|  |

* Use the arrow keys to move the camera around the objects.
* **Milestone 1**: Show the application to the instructor.

# 2: Creating a surface in the XZ plane

* Open ‘ObjectGenerator.h’ and ‘ObjectGenerator.cpp’
* In ‘ObjectGenerator.h,’ add the following code:

|  |
| --- |
| static float\* generateXZSurface(float width, float depth); |

* In ‘ObjectGenerator.cpp’ add the following code:

|  |
| --- |
| float\* ObjectGenerator::generateXZSurface(float width, float depth)  {  // This object's origin is in the center.  float halfWidth = width / 2.0f;  float halfDepth = depth / 2.0f;  // width  // D  // { A +--+ F  // depth{ |\ |  // { | \|  // { B +--+ C  // E  // 1 - size + (2 triangles \* 3 vertices each \* (4 position + 4 color))  // 1 + (2 \* 3 \* (4 + 4)) = 49  float\* data = new float[49];  // The first element stores the number of values  data[0] = 48;  int i = 1; // index  // Positions  // A  data[i++] = -halfWidth; // x  data[i++] = 0.0f; // y  data[i++] = -halfDepth; // z  data[i++] = 1.0f; // w  // B  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // C  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // D  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // E  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // F  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Colors  // A  data[i++] = 0.0f; // red  data[i++] = 0.4f; // green  data[i++] = 0.0f; // blue  data[i++] = 1.0f; // alpha  // B  data[i++] = 0.0f;  data[i++] = 0.4f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // C  data[i++] = 0.0f;  data[i++] = 0.4f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // D  data[i++] = 0.0f;  data[i++] = 0.4f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // E  data[i++] = 0.0f;  data[i++] = 0.4f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // F  data[i++] = 0.0f;  data[i++] = 0.4f;  data[i++] = 0.0f;  data[i++] = 1.0f;  return data;  } |

* In ‘OGLRenderer.cpp,’ in the ‘create()’ method, add the following code after the section that creates the “Axes”:

|  |
| --- |
| this->objects["Surface"] = new OGLObject("Surface");  this->objects["Surface"]->setVertexData(ObjectGenerator::generateXZSurface(10, 20));  data = this->objects["Surface"]->getVertexData();  VBOObject \* triangles = OGLObject::createVBOObject("triangles");  triangles->buffer = &data[1];  triangles->primitiveType = GL\_TRIANGLES;  triangles->bufferSizeInBytes = (unsigned int)data[0] \* sizeof(float);  triangles->numberOfVertices = 6;  triangles->positionComponent.count = 4;  triangles->positionComponent.type = GL\_FLOAT;  triangles->positionComponent.bytesToFirst = 0;  triangles->positionComponent.bytesToNext = 4 \* sizeof(float);  triangles->colorComponent.count = 4;  triangles->colorComponent.type = GL\_FLOAT;  triangles->colorComponent.bytesToFirst = sizeof(float) \* 24;  triangles->colorComponent.bytesToNext = 4 \* sizeof(float);  this->objects["Surface"]->addVBOObject(triangles);  this->objects["Surface"]->shaderProgram = shaderProgram3d; |

* Change ‘OGLRenderer::renderObjects()’ to the following:

|  |
| --- |
| void OGLRenderer::renderObjects()  {  GLuint shaderProgram;  GLuint localToWorldMatrixUnif;  for (auto iterator = this->objects.begin();  iterator != this->objects.end();  iterator++) {  shaderProgram = iterator->second->shaderProgram;  localToWorldMatrixUnif = glGetUniformLocation(shaderProgram, "localToWorldMatrix");  glUseProgram(shaderProgram);  glUniformMatrix4fv(  localToWorldMatrixUnif, 1, GL\_FALSE,  glm::value\_ptr(iterator->second->referenceFrame.orientation));  glUseProgram(0);  iterator->second->render();  }  } |

* Remove the following lines of code from ‘OGLRenderer::create()’:

|  |
| --- |
| glUseProgram(shaderProgram3d);  glUniformMatrix4fv(  localToWorldMatrixUnif, 1, GL\_FALSE,  glm::value\_ptr(this->objects["Axes"]->referenceFrame.orientation));  glUseProgram(0);  glUseProgram(shaderProgram3dv3);  glUniformMatrix4fv(  localToWorldMatrixUnif, 1, GL\_FALSE,  glm::value\_ptr(this->objects["Pyramid"]->referenceFrame.orientation));  glUseProgram(0); |

* Build and then run the application. The output should resemble the following:

|  |
| --- |
|  |

* **Milestone 2**: Show your output to the instructor
* Move the camera to under the surface. What do you notice?
* In ‘OGLRenderer::create(),’ comment out the following lines of code:

|  |
| --- |
| glEnable(GL\_CULL\_FACE);  glCullFace(GL\_BACK);  glFrontFace(GL\_CCW); |

* Run the program and move the camera under the surface. What do you notice? Uncomment the code.
* Repeat the same experiment with

|  |
| --- |
| glEnable(GL\_DEPTH\_TEST);  glDepthMask(GL\_TRUE);  glDepthFunc(GL\_LEQUAL);  glDepthRange(0.0f, 1.0f); |

# 3: Creating Pyramids (Step-by-Step)

* The pyramid is currently created by ‘OGLRenderer::createPyramid().’ Let’s add a new method to ‘ObjectGenerator’ that creates pyramids.
* In ‘ObjectGenerator.h,’ add the following interface:

|  |
| --- |
| static float\* generatePyramid(float width, float depth, float height); |

* In ‘ObjectGenerator.cpp’ add the following empty method:

|  |
| --- |
| float\* ObjectGenerator::generatePyramid(float width, float depth, float height)  {  float\* data;  return data;  } |

* The pyramid’s origin will be situated in its base. So we’ll need half the width and the height:

|  |
| --- |
| float\* ObjectGenerator::generatePyramid(float width, float depth, float height)  {  // This pyramid's origin is in the center of the base.  float halfWidth = width / 2.0f;  float halfDepth = depth / 2.0f;  float\* data;  return data;  } |

* We calculate the number of float elements and allocate that amount of memory. The first element stores the number of elements.

|  |
| --- |
| float\* ObjectGenerator::generatePyramid(float width, float depth, float height)  {  // This pyramid's origin is in the center of the base.  float halfWidth = width / 2.0f;  float halfDepth = depth / 2.0f;  // 1 size + (6 triangles \* 3 vertices each \* (4 position + 4 color))  // 1 + (6 \* 3 \* (4 + 4)) = 145  float\* data = new float[145];  // The first element stores the number of elements.  data[0] = 144;  return data;  } |

* When specifying the vertices we do so in a counter-clockwise manner (because of glFrontFace(GL\_CCW)) for each triangle. Here’s the south face (face with positive z):

|  |
| --- |
| int i = 1;  // South face  // Lower-left  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = height;  data[i++] = 0.0f;  data[i++] = 1.0f; |

* East face (face with positive x):

|  |
| --- |
| // East face  // Lower-left  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = height;  data[i++] = 0.0f;  data[i++] = 1.0f; |

* North face (face with negative z):

|  |
| --- |
| // North face  // Lower-left  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = height;  data[i++] = 0.0f;  data[i++] = 1.0f; |

* West face (face with negative x):

|  |
| --- |
| // West face  // Lower-left  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = height;  data[i++] = 0.0f;  data[i++] = 1.0f; |

* Since we are looking up at the base, the z is flipped compared to the surface we had created earlier.

|  |
| --- |
| // Base  // Upper-left  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Lower-left  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Upper-left  data[i++] = -halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f;  // Lower-right  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = -halfDepth;  data[i++] = 1.0f;  // Upper-right  data[i++] = halfWidth;  data[i++] = 0.0f;  data[i++] = halfDepth;  data[i++] = 1.0f; |

* Let’s make the colors interesting:

|  |
| --- |
| // Colors  // South face  // Lower-left  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // East face  // Lower-left  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // North face  // Lower-left  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Apex  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // West face  // Lower-left  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  // Apex  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Base  // Upper-left  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Lower-left  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Upper-left  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Lower-right  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  // Upper-right  data[i++] = 0.0f;  data[i++] = 0.0f;  data[i++] = 1.0f;  data[i++] = 1.0f;  return data; |

* In ‘OGLRenderer::create()’ add the following code after the surface creation section:

|  |
| --- |
| this->objects["Pyramid2"] = new OGLObject("Pyramid2");  this->objects["Pyramid2"]->setVertexData(ObjectGenerator::generatePyramid(2, 2, 2));  data = this->objects["Pyramid2"]->getVertexData();  triangles = OGLObject::createVBOObject("triangles");  triangles->buffer = &data[1];  triangles->primitiveType = GL\_TRIANGLES;  triangles->bufferSizeInBytes = (unsigned int)data[0] \* sizeof(float);  triangles->numberOfVertices = 18;  triangles->positionComponent.count = 4;  triangles->positionComponent.type = GL\_FLOAT;  triangles->positionComponent.bytesToFirst = 0;  triangles->positionComponent.bytesToNext = 4 \* sizeof(float);  triangles->colorComponent.count = 4;  triangles->colorComponent.type = GL\_FLOAT;  triangles->colorComponent.bytesToFirst = sizeof(float) \* 72;  triangles->colorComponent.bytesToNext = 4 \* sizeof(float);  this->objects["Pyramid2"]->addVBOObject(triangles);  this->objects["Pyramid2"]->shaderProgram = shaderProgram3d; |

* Run the application, there should be a pyramid at the world’s origin.
* Set the position of the new pyramid by adding this line of code after the pyramid creation code:

|  |
| --- |
| this->objects["Pyramid2"]->referenceFrame.setPosition(0.0f, 0.0f, 5.0f); |

* Run the application.
* **Milestone 3**: Show the output to the instructor.

# 4: Adding 10 Pyramids

* You can generate random numbers by first seeding the random number generator using the time and then calling the ‘rand()’ function to generate the random numbers.

|  |
| --- |
| // These includes are needed  #include <cstdlib>  #include <ctime>  // Call this once  srand((unsigned int)time(NULL));  // Then call rand() to generate random numbers  v1 = rand() % 100; // v1 in the range 0 to 99  v2 = rand() % 100 + 1; // v2 in the range 1 to 100  v3 = rand() % 30 + 1985; // v3 in the range 1985-2014 |

* Use a loop to create 10 pyramids, each pyramid has a random height in the range1 to 3, a random x position in the range -10 to 10, and a random z position in the range -10 to 10.
* **Milestone 4**: Demonstrate your application to the instructor

# 5: Adding a Cuboid Generator

* Use the technique used to generate pyramids and write a method to generate cuboid objects.
* **Milestone 5**: Add some cuboids to the scene and demonstrate the application to the instructor

**End of Lab**