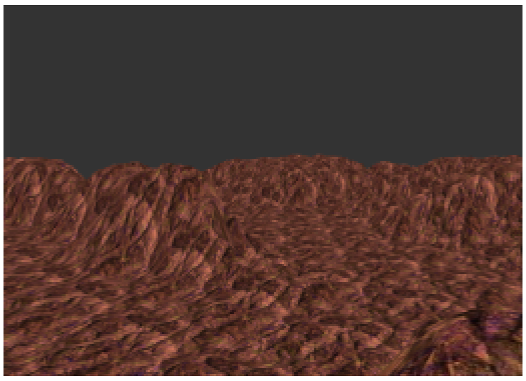
Tutorial 8: Index Buffers & Face Culling



Summary

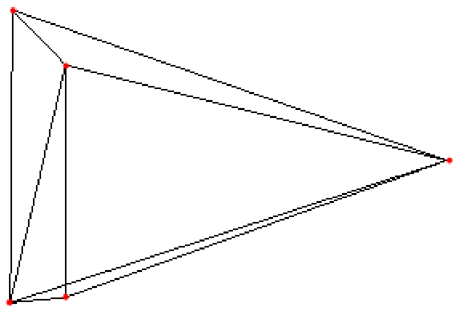
This tutorial will show you how to use indices to access your geometry data, allowing you to access a vertex multiple times in one draw call - useful when drawing certain shapes, such as a cube. This will be demonstrated by generating some terrain, which takes up less space when used with indices. Additionally, the ability to *cull* faces is introduced, via determining the *winding* of the geometric primitives being sent to the vertex shader.

New Concepts

Indices, Index Buffers, heightmap generation, face culling, triangle winding

Indices

So far in this tutorial series you've been drawing fairly simple geometry - triangles and quads. But what about if you wanted to draw a pyramid?



The pyramid above only has 5 unique vertices (shown as red dots), but if you were to draw it using **GL_TRIANGLES**, you'd need 18 vertices - 3 for each side, and 6 for the square base. Many of them would just be identical, which is not very space efficient! Thankfully, OpenGL provides a mechanism to reference a vertex multiple times in a single draw call - indices! Just like how the arrays you have been using in your C++ programs can be accessed via an index, OpenGL can access its Vertex Buffers using an index. So instead of just iterating through a list of vertices from beginning to end, OpenGL can choose which vertex to draw next using a series of index values.

As well as reducing the number of individual vertices required to form mesh geometry, the reduction in the number of vertices also serves another potential purpose. If we wanted to change the colour of a vertex of an indexed mesh, we just change a single colour value. However, if we want to change the colour value of a non-indexed array, we'd have to change potentially many different vertices, one for each triangle the vertex appears in. Take the pyramid example - the indexed pyramid has only one 'point' vertex, while the non-indexed pyramid has four 'point' vertices - one for each side. So we'd have to find four vertices to change the colour value of the point, meaning indices are both a space saver, and a potential time saver, too! There's one final performance improvement behind using indices - cache coherency. Modern graphics processing hardware generally has a vertex cache *before* the vertex shader stage, to cache the vertex attributes coming from a VBO, and one *after*, to store the results of the vertex processing. If a mesh uses indices rather than duplicates, each vertex is therefore more likely to be in either the pre or post vertex cache, and so will be processed quicker.

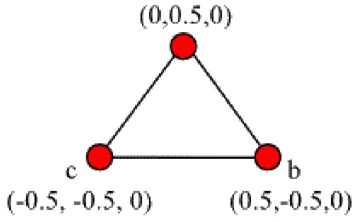
Index Buffers

So, indices are quicker, more convenient, and smaller. But how much smaller? In order to use indices, OpenGL needs an *Index Buffer* loaded into graphics memory, just like the Vertex Buffers you are accustomed to. This is a list of bytes, unsigned shorts or integers that tell OpenGL which vertex in the currently bound VBOs to draw next. So, in our pyramid example, instead of needing space for 18 vertices, we only need space for 5 *vertices*, and 18 *indices*, instead. If each vertex had a position (12 bytes), colour (16 bytes), and texture coordinate (8 bytes), that'd be 36 bytes per vertex, for a total of 648 bytes for the entire pyramid. If we stored the indices as **shorts** (2 bytes), we'd only need 180 bytes for the vertices, and 36 bytes for the indices, for a total of 216 bytes. Although these are still small amounts, you should be able to see how the savings can build up when using multiple, high polygon meshes - like the one you'll be making in the example program!

Face Culling

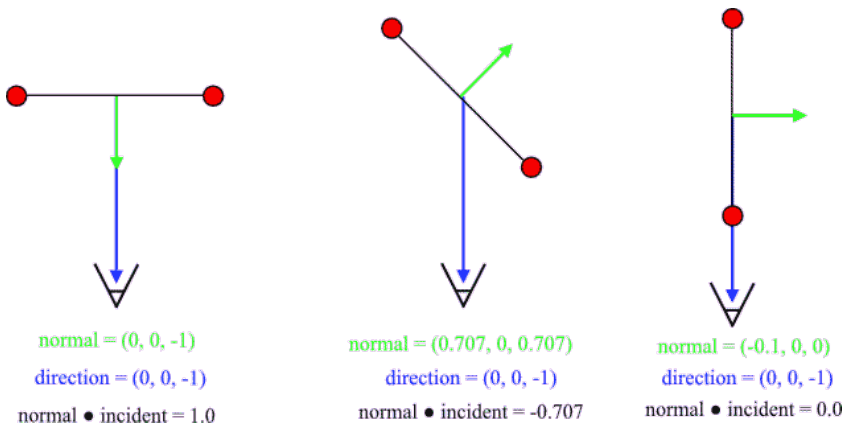
Up until now, these tutorials have used simple geometry - a single triangle, or a small number of tris to make up a cube. In the example program for this tutorial, however, we're going to be generating *a lot* of triangles, so now seems a good time to introduce yet another potential performance enhancement - *face culling*. Even with the frustum culling and early-z depth testing concept introduced earlier in the seriesl, you might still end up wasting vertex processing on vertices that will never be seen in the final image - those that get covered by other vertices in the same mesh. Think about a closed mesh like a cube - although it has 6 sides, no matter how you orient it, you'll never see more than 3 of the cube's sides; in rendering terms, the faces further away from the viewpoint get wholly covered over by the closer ones, or to think of it another way, only the faces actually facing towards the camera are visible.

So far, we haven't really paid any attention to the ordering of the vertices that make up the triangles in our scenes. This ordering, known as its *winding*, can be either clockwise, or anticlockwise. To demonstrate this, here's the triangle we made back in tutorial 1:



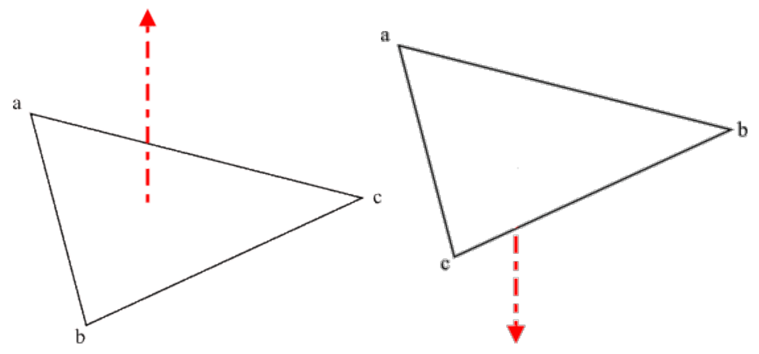
If the vertices were defined in clockwise order, we'd get the triangle *abc*, and if they were defined anticlockwise, we'd get *acb*. Depending on the exact implementation of your graphics hardware drivers, this winding will be used in one of two ways to determine whether a polygon faces the screen or not.

One way is to take the *dot product* of the triangle normal vector and the direction vector between the camera viewpoint and the current vertex - if the dot product is less than zero, the current triangle is facing away from the screen, and can be culled.



Dot Product Method of Determining Facet Direction: Left to Right: Facing, Not Facing, Edge Case

This normal can be determined by taking the *Cross Product* of the vectors (c-a) and (b-a) - you'll learn more about how to generate normals and how to do cross products later in the tutorial series, but for now know that the cross product of (c-a) and (b-a) is the opposite of the normal formed by the cross product of (b-a) and (c-a), so the normals between a clockwise and anticlockwise winding are flipped:



The surface normals created by anticlockwise, and clockwise vertex ordering, respectively

The second method is to take the *signed area* of the polygons as they appear on screen - so after they have been transformed to their final screen space coordinates. For a triangle, this signed area can be calculated from its 2D screen coordinates as follows:

$$area = \frac{1}{2}\ (a_x b_y - b_x a_y) + (b_x c_y - c_x b_y)$$

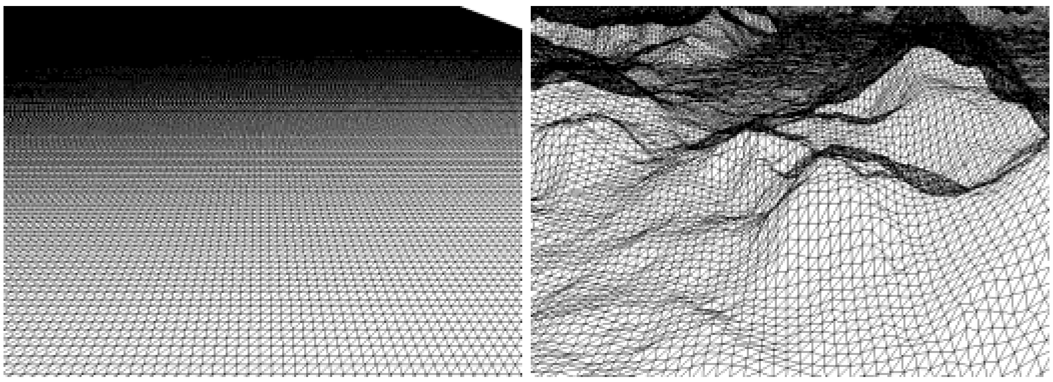
Again, depending on the order of the vertices in the triangle, we get opposite values that can be tested against - in this case, triangles with signed areas less than 0 are culled:

$$\text{area of triangle acb} = -0.25 = \frac{1}{2}\ (0 \cdot -0.5 - 0.5 \cdot 0.5) + (0.5 \cdot -0.5 - -0.5 \cdot -0.5)$$
$$\text{area of triangle abc} = 0.25 = \frac{1}{2}\ (0 \cdot -0.5 - -0.5 \cdot 0.5) + (-0.5 \cdot 0.5 - 0.5 \cdot -0.5)$$

Face culling does place some limitations on the meshes used. Flat geometry, like the triangle and quad we've used as mesh geometry so far, will get back face culled if the camera goes behind them, and with no geometry to hide this (as in the cube example) they will simply disappear. It works best with 'closed' geometry where there is always front facing facets to cover the back facing ones. Although OpenGL defaults to using anticlockwise triangle definition, as with most other operations, face culling is programmable - whether to cull front, back or even *both* faces, whether it is clockwise or anticlockwise winding that determines the face normal direction, and so on. It's also worth pointing out that this only counts for opaque objects; if they are transparent, you might actually want to temporarily disable face culling to render them correctly - yet another reason why transparent objects are a pain to deal with in rasterised graphics.

Example Program

To better demonstrate the benefits of index buffers, we're going to draw something new - a 3D landscape! If we use a simple algorithm to create a 2D grid of triangles, and then set their height to different values, we can create a basic, but quite effective landscape:


Left: A flat grid of triangles. Right: The same grid, with vertex heights adjusted to create a landscape

The landscape we're going to create will have 66,049 unique vertices, but 393,216 indices - imagine if all those indices had to be a separate vertex! Told you using indices would add up to big savings in memory. To get decent looking height values, we're going to load in the data from a heightmap - a simple file containing 257*257 **unsigned chars**, generated by a heightmap generation tool called *TerraGen*.

Mesh Class

Before we get to the heightmap generation, we must modify our *Mesh* class to support index buffers. To start off with, in order to automatically create an index array slot in our VBO array, we have to modify the *Mesh* class *MeshBuffer* enum, to include an additional named constant, *INDEX_BUFFER*. Remember to put it before *MAX_BUFFER*! Also, we need two new **protected** member variables in our *Mesh* class - an **unsigned int** to store the number of indices in the mesh, and a pointer to the index data itself.

```
enum MeshBuffer {
   VERTEX_BUFFER,COLOUR_BUFFER,
   TEXTURE_BUFFER,INDEX_BUFFER,
   MAX_BUFFER
};
...//Dots mean 'keep existing contents of this class!'
protected:
   GLuint          numIndices;
   unsigned int*   indices;
...
```

Mesh.h

Obviously, as we have new member variables, we must initialise them in our **constructor**, and **delete** the pointer in the **destructor**.

```
Mesh::Mesh(void)  {
...   //Dots mean 'keep existing contents of this function!'
indices        = NULL;
numIndices     = 0;
...
```

Mesh.cpp

```
Mesh::~Mesh(void) {
...
delete[]indices;
...
}
```

Mesh.cpp

Just as we modified the *BufferData* function when we introduced texturing into our rendering capabilities, we must also do so to use indices. So, we generate an index buffer if the mesh has indices, and buffer data to it much the same way as we do for colours and positions. Note, that instead of being of type **GL_ARRAY_BUFFER**, our index buffer is of type **GL_ELEMENT_ARRAY_BUFFER**, and instead of the array size being calculated using *numVertices*, it is done so using *numIndices*.

```
void  Mesh::BufferData()   {
...
   if(indices) {
      glGenBuffers(1, &bufferObject[INDEX_BUFFER]);
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
                   bufferObject[INDEX_BUFFER]);
      glBufferData(GL_ELEMENT_ARRAY_BUFFER, numIndices*sizeof(GLuint),
                   indices, GL_STATIC_DRAW);
   }
...
```

Mesh.cpp

The final change to the *Mesh* class is to the *Draw* function. Depending on whether you are rendering a direct list of vertices, or an indexed array of vertex elements, there are two slightly different OpenGL commands. So, we have an **if** statement to choose between them, which checks for indices by looking for a non-zero value in the *INDEX_BUFFER* slot in the *bufferObject* array. If there's no indices, we use **glDrawArrays**, just as we did back in tutorial 1. If there are indices, we use the new OpenGL function **glDrawElements**, which takes 4 parameters - the *type*, the number of *indices* to draw, the *size* of each index, and a pointer to some index data. This last parameter is a bit of legacy functionality from OpenGL 2.x, when geometry data could be in system memory, and so should be kept at 0.

```
void Mesh::Draw() {
...
   glBindVertexArray(arrayObject);
   if(bufferObject[INDEX_BUFFER]) {
      glDrawElements(type, numIndices, GL_UNSIGNED_INT, 0);
   }
   else{
      glDrawArrays(type, 0, numVertices);
   }
   glBindVertexArray(0);
...
}
```

Mesh.cpp

HeightMap Class

Now we have a *Mesh* class that can make use of indices, let's do something interesting! In your *nclgl* project, make a new **Subclass** of *Mesh*, called *HeightMap*. This class will create some terrain, using indices to cut down on the number of vertices in memory.

Header file

The header file doesn't contain too much. We'll be loading in the heightmap data from a file, so we must have the relevant **includes**. We also have some defines, which will be explained as they are used in the Class file. Note how the **constructor** takes in a file name as a **string**.

```
#pragma once

#include <string>
#include <iostream>
#include <fstream>

#include ".\nclgl\mesh.h"

#define RAW_WIDTH     257
#define RAW_HEIGHT    257

#define HEIGHTMAP_X   16.0f
#define HEIGHTMAP_Z   16.0f
#define HEIGHTMAP_Y   1.25f
#define HEIGHTMAP_TEX_X 1.0f / 16.0f
#define HEIGHTMAP_TEX_Z 1.0f / 16.0f

class HeightMap : public Mesh {
public:
   HeightMap(std::string name);
   ~HeightMap(void){};
};
```

HeightMap.cpp

Class file

We only have one major function in the *HeightMap* class, and that is its **constructor**. It starts off by creating a new file handle, using the **string** passed to it as a funciton parameter. This file will contain the actual heightmap data. Assuming it has successfully opened, we then set the correct number of vertices and indices for the mesh that will result from the heightmap. The heightmaps created by TerraGen are an array of 257 by 257 **unsigned chars**, represented by the *RAW_WIDTH* and *RAW_HEIGHT* defines from the header file. 257 values in each dimension is enough to create a grid of 256 by 256 square patches, with each patch made up of two triangles, each of which requiring 3 indices to be rendered. We also initialise the *vertices*, *textureCoords*, and *indices* pointers to memory locations large enough for the vertex and index data we are about to create.

```
#include "HeightMap.h"

HeightMap::HeightMap(std::string name) {
   std::ifstream file(name.c_str(), ios::binary);
   if(!file) {
      return;
   }
   numVertices     = RAW_WIDTH*RAW_HEIGHT;
   numIndices      = (RAW_WIDTH-1)*(RAW_HEIGHT-1)*6;
   vertices        = new Vector3[numVertices];
   textureCoords   = new Vector2[numVertices];
   indices         = new GLuint[numIndices];
```

HeightMap.cpp

We then read in the file data into a temporary memory location, appropriately enough called *data*. This data is simply a y-axis height value between 0 and 255, with the x and z axis' inferred by the position in the *data* array. So, to turn it into data suitable for transferral to graphics memory, we use a nested **for** loop, to transform the height data into *Vector3s*. Note how we scale the height and position data by the **defines** in the header file - we could make *HEIGHTMAP_Y* a larger value to make the terrain more rocky, for example. Once we exit the loops, we don't need *data* any more, so we **delete** it.

```
   unsigned char*data = new unsigned char[numVertices];
   file.read((char*)data,numVertices*sizeof(unsigned char));
   file.close();

   for(int x = 0; x < RAW_WIDTH; ++x) {
      for(int z = 0; z < RAW_HEIGHT; ++z) {
         int offset = (x * RAW_WIDTH) + z;

         vertices[offset]        = Vector3(
         x * HEIGHTMAP_X,data[offset] * HEIGHTMAP_Y,z * HEIGHTMAP_Z);

         textureCoords[offset]   = Vector2(
         x * HEIGHTMAP_TEX_X, z * HEIGHTMAP_TEX_Z);
      }
   }

   delete data;
```

HeightMap.cpp

We now have our vertices in system memory, but we still need to create our indices. We do this using another nested **for** loop. We want to create 256 by 256 square 'patches', each of which is made up out of 2 triangles. So, we create a for loop which creates each of these patches in turn, generating the indices that will draw their respective triangles. You should be able to see how using indices saves memory - we're reusing vertices *a* and *c* in each patch, but instead of having to make a duplicate copy of the vertices' attribute data, we just write another index. As with the *GenerateTriangle* function we wrote back in tutorial 1, we finish our geometry definition with a call to the newly modified *BufferData* function, which will now upload both the vertex *and* index data to the graphics card.

```
   numIndices = 0;

   for(int x = 0; x < RAW_WIDTH-1; ++x) {
      for(int z = 0; z < RAW_HEIGHT-1; ++z) {
         int a = (x      * (RAW_WIDTH)) + z;
         int b = ((x+1) * (RAW_WIDTH)) + z;
         int c = ((x+1) * (RAW_WIDTH)) + (z+1);
         int d = (x      * (RAW_WIDTH)) + (z+1);

         indices[numIndices++] =  c;
         indices[numIndices++] =  b;
         indices[numIndices++] =  a;

         indices[numIndices++] =  a;
         indices[numIndices++] =  d;
         indices[numIndices++] =  c;
      }
   }

   BufferData();
}
```

HeightMap.cpp

Renderer header file

Not much new in our *Renderer* class, this time around. Note how we include the new *HeightMap* class header file, and have a pointer to a *HeightMap* as a protected member variable.

```
#pragma once

#include "./nclgl/OGLRenderer.h"
#include "./nclgl/camera.h"
#include "./nclgl/HeightMap.h"

class Renderer : public OGLRenderer {
public:
   Renderer(Window &parent);
   virtual ~Renderer(void);

   virtual void RenderScene();
   virtual void UpdateScene(float msec);

protected:
   HeightMap*  heightMap;
   Camera*     camera;
};
```

renderer.h

Renderer Class file

The actual *Renderer* class for this tutorial is nothing new, as all of the new functionality is contained in the *Mesh* class. We create a new *HeightMap* and *Camera*, and create the same shader program we used back in the texturing tutorial. We want our terrain to be textured, so we set it to an earthy texture on line 14, and on line 21 we set it to repeat, using the *SetTextureRepeating* function we made back in tutorial 3. We also set a *perspective* projection matrix, using a *farZ* value of 10,000, so that we can see all of the terrain at once. Finally, we enable depth testing, and set *init* to **true**.

```
 1 #include "Renderer.h"
 2 
 3 Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
 4    heightMap      = new HeightMap("../Textures/terrain.raw");
 5    camera         = new Camera();
 6 
 7    currentShader  = new Shader("../Shaders/TexturedVertex.glsl",
 8                               "../Shaders/TexturedFragment.glsl");
 9 
10    if(!currentShader->LinkProgram()) {
11       return;
12    }
13 
14    heightMap->SetTexture(SOIL_load_OGL_texture(
15    "../Textures/Barren Reds.JPG",
16    SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
17 
18    if(!heightMap->GetTexture()) {
19       return;
20    }
21    SetTextureRepeating(heightMap->GetTexture(),true);
22 
23     projMatrix = Matrix4::Perspective(1.0f,10000.0f,
24                            (float)width/(float)height,45.0f);
25
26     glEnable(GL_DEPTH_TEST);
27     glEnable(GL_CULL_FACE);
28     glCullFace(GL_BACK);
29
30     init = true;
31 }
```

renderer.cpp

You'll also see a new pair of OpenGL functions - a **glEnable** call on line 27 using the symbolic constant **GL_CULL_FACE**, which turns on face culling, and **glCullFace** on line 28, which selects whether to cull front or back facing polygons. We usually want back facing polygons to be culled, so the symbolic constant **GL_BACK** is used.

Our **destructor** simply **deletes** the *heightMap* and *camera* variables, while *UpdateScene* updates the camera and forms a new view matrix.

```
32 Renderer::~Renderer(void)  {
33     delete heightMap;
34     delete camera;
35 }
36
37 void Renderer::UpdateScene(float msec) {
38     camera->UpdateCamera(msec);
39     viewMatrix = camera->BuildViewMatrix();
40 }
```

renderer.cpp

RenderScene enables the texturing shader, updates its matrices and texture sampler, then draws the heightmap. The changes we made to the *Mesh* class handles everything we need to in regard to index buffers, so the rest of our program doesn't need to know whether the geometry we are rendering is indexed or not. It's generally good programming practice to try and reduce the 'footprint' of a new feature in this way, and not being able to 'hide' the details of a new feature is generally a symptom of poor software design somewhere along the way.

```
41 void Renderer::RenderScene()  {
42     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
43
44     glUseProgram(currentShader->GetProgram());
45     UpdateShaderMatrices();
46
47     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
48                 "diffuseTex"), 0);
49
50     heightMap->Draw();
51
52     glUseProgram(0);
53     SwapBuffers();
54 }
```

renderer.cpp

Main file

Our **main** file is also nothing new! It's the same as the main file we used in the view matrix tutorial.

```cpp
#include "./nclgl/window.h"
#include "Renderer.h"

#pragma comment(lib, "nclgl.lib")

int main() {
    Window w("Index Buffers!", 800,600,false);
    if(!w.HasInitialised()) {
        return -1;
    }

    Renderer renderer(w);
    if(!renderer.HasInitialised()) {
        return -1;
    }

    w.LockMouseToWindow(true);
    w.ShowOSPointer(false);

    while(w.UpdateWindow() &&
    !Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
        renderer.UpdateScene(w.GetTimer()->GetTimedMS());
        renderer.RenderScene();
    }
    return 0;
}
```

main.cpp

Tutorial Summary

If you run the program, you should be able to see your new heightmap terrain, and fly around it using your camera controls. It's pretty basic, but looks better than the triangles you've been using up to now! More importantly, it has shown you how to use indices when rendering geometry. Not every mesh is suitable for being rendered using indices, but for those that are, they provide an effective means of reducing the memory footprint of your geometry data. You've also had a quick look at face culling, a simple and effective way of reducing the number of vertices processed in a draw call.

Further Work

1) In this tutorial you used the default *Mesh* geometry draw type - **GL_TRIANGLES**. What other geometry types can be rendered using indices? Which geometry type would be unsuitable for indices?

2) Suppose you wanted to use **GL_TRIANGLE_STRIP** as the geometry draw type for your heightmap - how would you achieve that? Investigate what a *degenerate* triangle is in relation to geometry rendering.

3) Try making your landscape look a little more realistic by creating some 'fake' shadows. Add colour data to your mesh geometry, by ranging from white to black, determined by the vertices' height. Or maybe add snow to high peaks using colours?

4) Can you use indices to draw a *textured* cube using only 8 vertices? Why / why not?