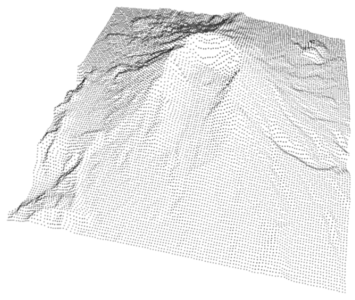
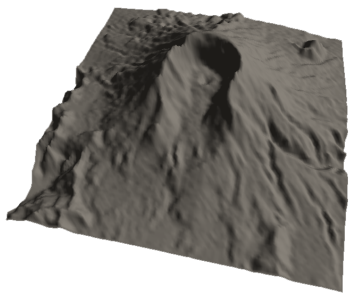
Computer Graphics Tick 3

Rasterization with OpenGL



(a) The mesh's 16384 vertices



(b) The heightmap shaded

Figure 1: In this exercise you will visualize a heightmap as a mesh using OpenGL. (a) shows the individual vertices of the mesh, and (b) shows the mesh shaded with diffuse lighting.

1 Introduction

In this exercise you will write code for rendering an image with OpenGL using *rasterization*.

You will first use OpenGL to transfer data from the CPU memory onto the GPU memory to draw a simple cube. Secondly, you will illuminate this cube with diffuse illumination using *shaders* — little programs compiled for the GPU. Finally, you'll load some heightmap data onto the GPU to visualize a real-world volcano.

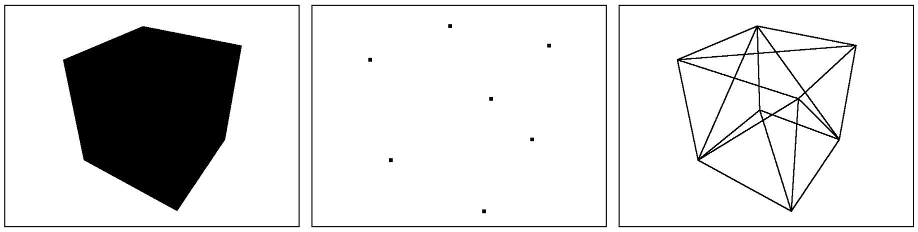


Figure 2: If you have set Tick 3 up correctly, you will see a black cube. Press $\boxed{\texttt{P}}$ to see the vertices, and $\boxed{\texttt{W}}$ to see that the cube is made up of triangles.

2 Getting started

The practical materials are available online: `http://www.cl.cam.ac.uk/TODO/cst1a-gfx/tick3.zip`. It contains the following files and directories:

tick3	
├ lib	
│ ├ JOML	*Java math library for OpenGL calculations*
│ └ lwjgl	*Lightweight Java Game Library*
├ resources	
│ ├ fragment_shader.glsl	*Processes fragments created by the rasterizer*
│ ├ vertex_shader.glsl	*Processes vertex data*
│ └ mtsthelens.png	*Heightmap that you will render*
└ src/gfx/tick3	
├ Camera.java	*Controls 3D camera position*
├ OpenGLApplication.java	*Uses OpenGL to interface with the GPU*
├ Shader.java	*Compiles a shader – a program for the GPU*
├ ShaderProgram.java	*Uses the vertex and fragment shaders together*
└ Tick3.java	*Main class*

The files you will modify and submit for this exercise have been highlighted. Follow the instructions in the setup documentation to compile and run the source code for this exercise.

You should see a black cube like the one in Figure 2. You can move the camera around by clicking and dragging with your mouse. You can press the $\boxed{\texttt{W}}$ key to enable wireframe drawing, and the $\boxed{\texttt{P}}$ key to enable vertex-only drawing.

2.1 OpenGL

OpenGL is an application programming interface (API) for graphics hardware, i.e. graphics processing units (GPU). OpenGL contains hundreds of special commands that are used to make a GPU draw shapes. You will use some of the more common OpenGL commands in this exercise.

You will need a computer that can run OpenGL 3.0. Most computers will support OpenGL 3.0, but if your personal machine does not, you can use the MCS machines instead.

The typical OpenGL rendering pipeline is as follows:

1. Specify data for shapes using geometric primitives (generally triangles).
2. Run a *vertex shader* on input primitives to determine their position on the screen, and other optional rendering attributes (e.g. colour).
3. Perform *rasterization*. This converts geometric primitives into fragments (potential pixels) with locations on the screen.
4. Finally, run a *fragment shader* on each fragment generated by rasterization to determine its final colour and position on the screen.

For more information, the OpenGL Programming Guide 8th Edition provides a good reference. Make sure to only consult up-to-date documentation.

2.2 LWJGL — Light-Weight Java Games Library

OpenGL is a C library available on most platforms (Windows, OSX, Linux, Android, iOS). The standard Java libraries do not provide access to OpenGL, so it is necessary to use an external library. In this course we will use the Lightweight Java Game Library[1] (LWJGL) to provide wrappers around OpenGL's C functions. The Java LWJGL functions work largely the same as the C ones.

The LWJGL library is included in the ZIP file with the template code, in the lib/lwjgl directory.

2.3 JOML — Java OpenGL Math Library

In addition to LWJGL you will also use JOML library, which contains classes for operations on vectors and matrices. The library is more powerful than the simple Vector class you used in the previous tics and is in particular intended for OpenGL applications. It is worth checking a few examples at `https://github.com/JOML-CI/JOML/wiki/JOML-and-modern-OpenGL`.

The JOML library is included in the ZIP file with the template code, in the lib/JOML.jar file.

[1]Lightweight Java Game Library 3 – `https://www.lwjgl.org/`

2.4 Compiling and running the code

The instruction below is for compiling and running the code form the command line. Refer to the document *Working with IDEs* for an instruction for an IDE.

When compiling the code, it is necessary to specify the classpath to the libraries we will use. To compile, change the current directory to tick3, then run:

```
javac -classpath lib/JOML.jar:lib/lwjgl/jar/lwjgl.jar -d ./out src/
    gfx/tick3/*.java
```

The -d option specifies where to put the compiled classes. It is a good practice not to mix the sources with compiled code.

Since the program needs to read a few files from the RESOURCES directory, it must be started from the tick3 directory. Moreover, LWJGL library consist of both JAVA classes and a native library, which needs to be specified at start-up using -Djava.library.path argument. You can start the program with the following command:

```
java -classpath lib/JOML.jar:lib/lwjgl/jar/lwjgl.jar:./out -Djava
    .library.path=lib/lwjgl/native gfx.tick3.Tick3 --input
    resources/mtsthelens.png
```

On OSX you may need to add -XstartOnFirstThread argument. Another common issue on OSX is that the application window is hiden behind other windows. If this is the case, use *Mission Control* to find the OpenGL window.

3 What has already been done for you

OpenGL programs can be long and complex, so several steps required for setting up an interactive application have been already implemented for you. This section will explain what some of the existing code in Tick 3 does. You do not need to modify any of this code.

3.1 Initializing the application

Let's first look at OpenGLApplication.initializeOpenGL method.

```
public void initializeOpenGL() {
    if (glfwInit() != true)
        throw new RuntimeException("Unable_to_initialize_the_
            graphics_runtime.");

    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

```
        window = glfwCreateWindow(WIDTH, HEIGHT, "Tick␣3", NULL,
            NULL);
        if (window == NULL)
            throw new RuntimeException("Failed␣to␣create␣the␣
                application␣window.");

                ... // removed for brevity

        // Enable v-sync
        glfwSwapInterval(1);

        // Cull back-faces of polygons
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        // Do depth comparisons when rendering
        glEnable(GL_DEPTH_TEST);

        // Create camera, and setup input handlers
        camera = new Camera((double) WIDTH / HEIGHT, FOV_Y);
        initializeInputs();

        // Create shaders and attach to a ShaderProgram
        Shader vertShader = new Shader(GL_VERTEX_SHADER, VSHADER_FN
            );
        Shader fragShader = new Shader(GL_FRAGMENT_SHADER,
            FSHADER_FN);
        shaders = new ShaderProgram(vertShader, fragShader, "colour
            ");

        // Initialize mesh data in CPU memory
        float vertPositions[] = initializeVertexPositions(
            heightmap );
        int indices[] = initializeVertexIndices( heightmap );
        float vertNormals[] = initializeVertexNormals( heightmap );
        no_of_triangles = indices.length;

        // Load mesh data onto GPU memory
        loadDataOntoGPU( vertPositions, indices, vertNormals );
    }
```

The main steps for initializing OpenGLApplication are:

1. The method initializes OpenGL with glfwInit(), creates a window with glfwCreateWindow()
2. OpenGL rendering parameters are set: v-sync is enabled, glEnable(G_CULL_FACE)

stops the triangles that are facing away from camera from being drawn, glEnable(GL_DEPTH_TEST) enables Z-buffer depth testing.

3. A Camera object is created so you can pan around the scene.
4. Two shaders are compiled from source code files: a vertex shader and a fragment shader. These are encapsulated into a single ShaderProgram.
5. The heightmap image is read into CPU memory.
6. Mesh data is generated on the CPU side, and stored in three arrays: vertPositions, vertNormals, and indices. These store the geometry of a cube for now.
7. The three arrays vertPositions, vertNormals, and indices are copied into GPU memory for drawing with OpenGL with loadDataOntoGPU().

3.2 Rendering with OpenGL

Next, let's look at run(): the main loop, and render().

```
public void run() {
   initializeOpenGL();
   while (glfwWindowShouldClose(window) != true) {
        render();
   }
}

public void render() {

    // Step 1: Pass a new model-view-projection matrix to the
        vertex shader
    Matrix4f mvp_matrix; // Model-view-projection matrix
    mvp_matrix = new Matrix4f(camera.getProjectionMatrix()).mul(
        camera.getViewMatrix());

    int mvp_location = glGetUniformLocation(shaders.getHandle(), "
        mvp_matrix");
    FloatBuffer mvp_buffer = BufferUtils.createFloatBuffer(16);
    mvp_matrix.get(mvp_buffer);
    glUniformMatrix4fv(mvp_location, false, mvp_buffer);

    // Step 2: Clear the buffer

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set the background
        colour to dark gray
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Step 3: Draw our VertexArray as triangles

    glBindVertexArray(vertexArrayObj); // Bind the existing
        VertexArray object
    glDrawElements(GL_TRIANGLES, no_of_triangles, GL_UNSIGNED_INT,
        0); // Draw it as triangles
    glBindVertexArray(0);              // Remove the binding

    // Step 4: Swap the draw and back buffers to display the
        rendered image
    glfwSwapBuffers(window);
    glfwPollEvents();
    checkError();
}
```

The render() method represents a typical main loop for any interactive graphics application. It repeats the following four steps:

1. First, we need to update the camera position by passing a new model-view-projection matrix to the vertex shader.
2. glClear(...) clears the OpenGL colour and depth buffers. If we didn't do this, we would re-draw over the previous frame's image.
3. glDrawElements() draws the geometry that we loaded earlier with loadDataOntoGPU(). This is rendered to the back buffer.
4. Since our window is double-buffered, swap the front and back buffers to display the fully-drawn image to the user. Check for any input events, e.g. mouse movements, with glfwPollEvents(). Finally check if any OpenGL errors have occurred.

4 Loading surface normals onto the GPU

Your first task is to pass the surface normal data to OpenGL. Once we know the surface normals, we can illuminate our objects properly. For this you will modify the code in the loadDataOntoGPU() method in OpenGLApplication.java.

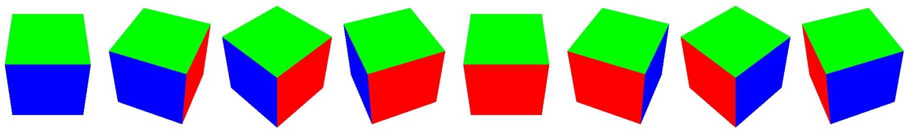


Figure 3: The cube rendered over several rotations, with surface normals represented using `abs(frag_normal)`. Red corresponds to a $\pm X$ vector, green corresponds to a $\pm Y$ vector, and blue corresponds to $\pm Z$ vector.

```
public void loadDataOntoGPU() {
    int shaders_handle = shaders.getHandle();

    // LOAD VERTEX POSITIONS
    FloatBuffer vertex_buff = BufferUtils.createFloatBuffer(
        vertPositions.length);
    vertex_buff.put(vertPositions);
    vertex_buff.flip();
    int vertex_handle = glGenBuffers();
    glBindBuffer(GL_ARRAY_BUFFER, vertex_handle);
    glBufferData(GL_ARRAY_BUFFER, vertex_buff, GL_STATIC_DRAW);
    int pos_loc = glGetAttribLocation(shaders_handle, "position");
    if (pos_loc != -1) {
        glVertexAttribPointer(pos_loc, 3, GL_FLOAT, false, 0, 0);
        glEnableVertexAttribArray(pos_loc);
    }

    // LOAD VERTEX NORMALS
    TODO: Put normal array into a buffer in CPU memory
    TODO: Create an OpenGL buffer and load it with normal data
    TODO: Get the location of the “normal” variable in the shader
    TODO: Specify how to access the variable, and enbale it

    // LOAD VERTEX INDICES
    ...
}
```

You can see we are already loading the vertex position data in the array vertPositions onto the GPU. That is why the cube is being drawn on the screen. We now need to load the vertex normals in the vertNormals array onto the GPU.

The data stored in float[] array cannot be directly used by OpenGL and it needs to be first transferred into a FloatBuffer. Since OpenGL is a C rather than Java library, it expects normal data to be stored in a continuous segment of memory, as they would be stored in C. FloatBuffer ensures that normals are stored in such format. The following code allocates FloatBuffer and puts the data into that butter.

```
FloatBuffer normal_buffer = BufferUtils.createFloatBuffer(
    vertNormals.length);
normal_buffer.put(vertNormals).flip();
```

The flip() method flips the state from writing to reading so that OpenGL can start reading from our normal_buffer object. Next, we need to get a handle for a new OpenGL buffer for the normals:

```
int normal_handle = glGenBuffers();
```

We can then bring that buffer into existence on the GPU with glBindBuffer(), and load it with the CPU data from normal_buffer using glBufferData():

```
glBindBuffer(GL_ARRAY_BUFFER, normal_handle);
glBufferData(GL_ARRAY_BUFFER, normal_buffer, GL_STATIC_DRAW);
```

We must then make sure the normal data can be accessed in the vertex shader. First, we get the location normal_loc of the variable named "normal" in the shader program. If this variable doesn't exist, normal_loc will be -1, and we can ignore the normal data.

```
int normal_loc = glGetAttribLocation(shaders_handle, "normal");
```

If the variable does exist, and we want to render using vertex normals, we specify its layout with glVertexAttribPointer(). There are three numbers for each normal, and they are floating point numbers. We finally mark the attribute as enabled with glEnableVertexAttribArray

```
if (normal_loc != -1) {
    glVertexAttribPointer(normal_loc, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(normal_loc);
}
```

5 Illumination with the fragment shader

Now we have given OpenGL access to the surface normal of each vertex in the mesh, let's render our cube with illumination. We will illuminate our cube using the fragment shader and illumination model:

$$\texttt{colour} = \underbrace{C_{\mathit{diff}}\, I_a}_{\text{Ambient}} + \underbrace{C_{\mathit{diff}}\, k_d\, I \max(0, N \cdot L)}_{\text{Diffuse}} \qquad (1)$$

where C_{diff} is the mesh colour, I_a the ambient light colour, k_d the diffuse coefficient, I the directional light colour, N the world-space surface normal, and L the world-space direction towards the light. The '$\cdot$' operator is a dot product.

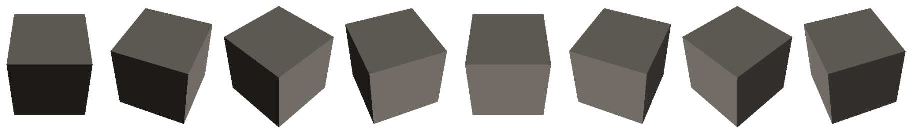


Figure 4: The cube rendered over several rotations, illuminated with simple ambient+diffuse shading.

5.1 Shaders

A shader is a little program that is compiled just for the GPU. They are writen in GLSL – the OpenGL Shading Language, which is similar to both C and Java. The complete specification of the GLSL language can be found at `https://www.opengl.org/documentation/glsl/`. However, you for this exercise you only need to know a small set of GLSL instructions, explained in the lectures and below. To draw any shape in OpenGL, we need two shaders: a vertex shader, which processes vertices, and a fragment shader, which processes fragments (pixels).

The vertex shader's job is to transform and project the 3D coordinates of each vertex into 2D coordinates on the screen and store that 2D coordinates in the special variable gl_Position. In vertex_shader.glsl, this is done by:

```
gl_Position = projection * view * model * vec4(position, 1.0);
```

where model, view, and projection are the $4/times4$ M, V, and P matricies that we set using setModelViewProjectionMatrices(). The vertex shader can also set other "out" variables, like frag_position and frag_normal that will be interpolated by the rasterizer, and assigned to individual fragments.

The fragment shader's job is to set the colour variable for each fragment — the colour that will be drawn on the screen. This is where you will implement some simple illumination, using its frag_normal variable.

5.2 Fragment shader code

Let's look at the source code in fragment_shader.glsl.

```
#version 330

in vec3 frag_normal;    // fragment normal in world space

out vec3 colour;

void main()
{
    const vec3 I_a = vec3(0.2, 0.2, 0.2);       // Ambient light
        intensity (and colour)

    const float k_d = 0.8;                      // Diffuse light
        factor
    vec3 C_diff = vec3(0.560, 0.525, 0.478);    // Diffuse light
        colour

    const vec3 I = vec3(0.941, 0.968, 1);   // Light intensity (and
         colour)
    vec3 L = normalize(vec3(2, 1.5, -0.5)); // The light direction
        as a unit vector
    vec3 N = frag_normal;                   // Normal in world
        coordinates

    // TODO: Calculate colour using the illumination model
    colour = abs(frag_normal);
}
```

Your task here is to calculate the RGB vector colour using the illumination model given in Equation 1. The cube should now look like Figure 4.

6 Rendering a heightmap

Now you are rendering a cube with illumination. However, this is not very exciting, and your GPU can handle many more triangles than the 12 it is processing currently.

Your final task is to render something more interesting: terrain data from the real world. This data is in the form of a heightmap, and is stored in mtsthelens.png. It represents Mt. Saint Helens, the volcano in Washington state, which erupted spectacularly in 1980.

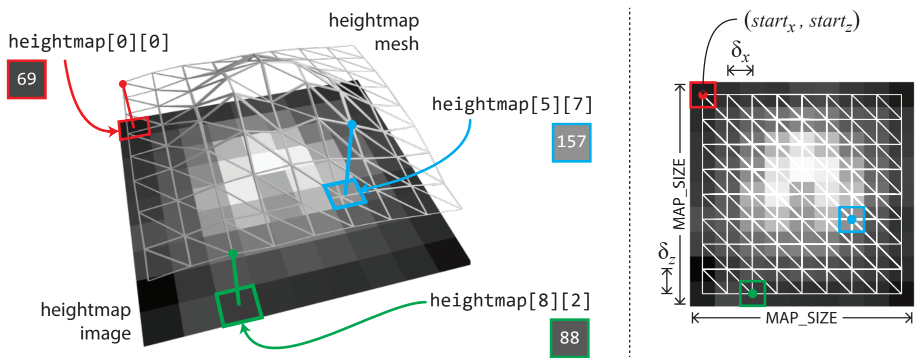


Figure 5: This shows how a heightmap image is used to create a heightmap mesh. Three pixels in image have been highlighted to show where they line up in the mesh. In this exercise, your mesh will be more detailed.

6.1 Preparing vertex positions

Currently, the vertex positions in the float array vertPositions have been predefined to represent the cube. The M mesh vertices are stored in as follows:

$$\text{vertPositions} = [x_0, y_0, z_0, x_1, y_1, z_1, \ldots, x_M, y_M, z_M]$$

where $[x_m, y_m, z_m]$ represents the 3D coordinate of the m^{th} vertex. There are currently $M = 36$ vertices represented in vertPositions: 6 for each face of the cube. Each cube face, a "quad", is made of two triangles.

You will replace these values so each value in the heightmap array corresponds to one vertex position in vertPositions.

```
/**
 * Create an array of vertex positions.
 *
 * @param heightmap 2D array with the heightmap
 * @return Vertex positions in the format { x0, y0, z0, x1, y1,
      z1, ... }
 */
public float[] initializeVertexPositions( float[][] heightmap )
     {

    int heightmap_width_px = heightmap[0].length;
    int heightmap_height_px = heightmap.length;

    float start_x = -MAP_SIZE / 2;   // X coordinate of first
        vertex
    float start_z = -MAP_SIZE / 2;   // Z coordinate of first
        vertex

    // Gaps between vertices along the X and Z axes
    float delta_x = MAP_SIZE / heightmap_width_px;
    float delta_z = MAP_SIZE / heightmap_height_px;

    TODO: create float array for vertPositions of the right size

    for (int row = 0; row < heightmap_height_px; row++) {
        for (int col = 0; col < heightmap_width_px; col++) {
          float x, y, z;
          TODO: Work out x, y, and z coordinates of vertex
          TODO: Calculate the index into the vertPositions array
          TODO: Set three elements in vertPositions to x, y, and z
        }
    }

    return vertPositions;
}
```

The main steps are:

1. Assign a suitably sized float array to vertPositions. It should be large enough to hold 3× the number of values in the heightmap.
2. Work out x and z coordinates of each vertex corresponding to the pixel row rows down and col columns along in the heightmap image. Use start_x, start_z, δ_x and δ_z to help you.
3. Assign the y coordinate of each vertex to heightmap[row][col].

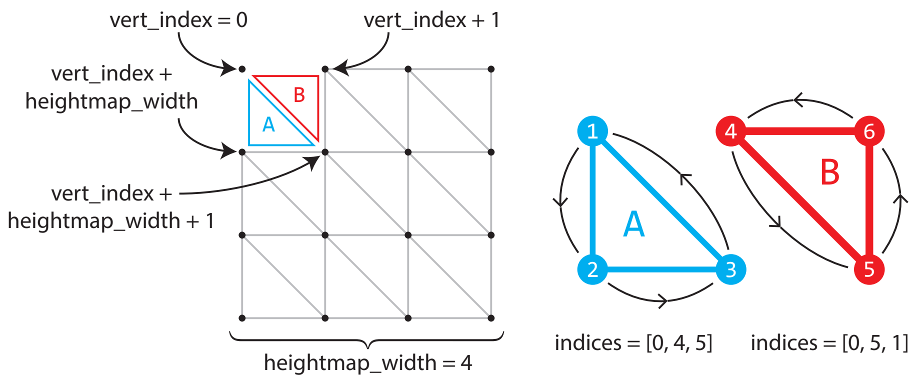


Figure 6: This figure shows how you will use triangles to join up your vertices. In this example, there are $4 \times 4 = 16$ vertices, and $3 \times 3 \times 2 = 18$ triangles. The four top left vertices are joined by triangles A and B. Each triangle is defined as indices into the vertex array, anti-clockwise.

4. Calculate the index into vertPositions that corresponds to that vertex.
5. Write three float values for x, y, z into vertPositions.

6.2 Preparing vertex indices

You will now define how to join up the vertices with indices. Figure 6 shows how we will use two triangles to join groups of four vertices.

```
    /**
     * Create an array of vertex indices.
     *
     * @param heightmap 2D array with the heightmap
     * @return Table with the vartex indices, three indices for
         each triangle
     */
    public int[] initializeVertexIndices( float[][] heightmap ) {

      int heightmap_width_px = heightmap[0].length;
      int heightmap_height_px = heightmap.length;

      TODO: create int array for indices of the right size

      int index_count = 0;
      for (int row = 0; row < heightmap_height_px - 1; row++) {
        for (int col = 0; col < heightmap_width_px - 1; col++) {
            TODO: Get vert_index for the corresponding vertex at (row,col)
            TODO: Add three indices to index_count for lower triangle ‘A’
            TODO: Add three indices to index_count for upper triangle ‘B’
        }
      }

      return indices;
    }

}
```

The main steps are:

1. Assign an int array to indices that is large enough to hold six vertices for each group of four vertices.

   ```
   indices = new int[6 * (heightmap_width_px - 1) * (
       heightmap_height_px - 1)];
   ```

2. Assign vert_index to the index of the top left vertex in that group of four.

   ```
   int vert_index = heightmap_width_px * row + col;
   ```

3. Add the three indices corresponding to the lower triangle A for each group of four vertices. It is important the indices are given in anti-clockwise order, or the triangles will not be drawn.[2]

   ```
   indices[count++] = vert_index;
   indices[count++] = vert_index + heightmap_width_px;
   ```

[2]Note how we assign to `indices[count++]` to set an index at location `count`, and then incrememnt count in a single call.

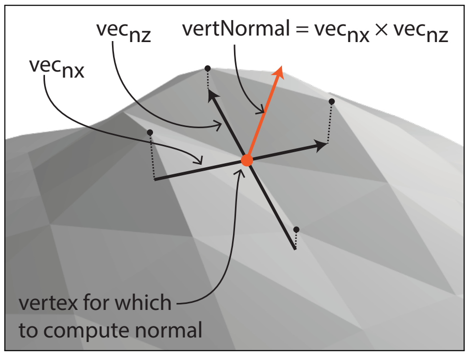

vec_nz
vertNormal = vec_nx × vec_nz
vec_nx
vertex for which to compute normal

Figure 7: One way to compute the direction of the normal at a vertex (shown in red) is to compute a cross product of two tangent vectors (shown in black).

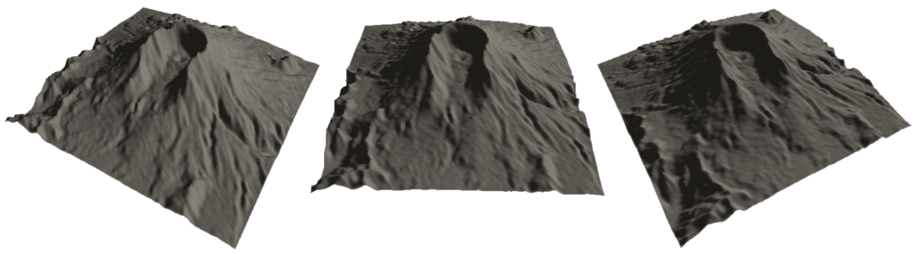


Figure 8: The mountain rendered with diffuse illumination (replace with better figure).

```
indices[count++] = vert_index + heightmap_width_px + 1;
```

4. Add the three indices corresponding to the upper triangle B in a similar way to triangle A. See Figure 6 to see which indices to use.

6.3 Preparing vertex normals

The final task is to work out surface normals for each vertex. This will allow you to illuminate your heightmap surface. Similarly to vertex positions, the M mesh normals are stored as follows:

$$\mathsf{vertNormals} = [nx_0, ny_0, nz_0, nx_1, ny_1, nz_1, \ldots, nx_M, ny_M, nz_M]$$

where $[nx_m, n_ym, nz_m]$ represents the 3D normal vector of the m^{th} vertex.

```
/**
 * Create an array of vertex normals.
 *
 * @param heightmap 2D array with the heightmap
 * @return Array of vertex normals in the format { n_x0, n_y0,
     nz0, n_x1, n_y1, n_z1, ... }
 */
public float[] initializeVertexNormals( float[][] heightmap ) {

  int heightmap_width_px = heightmap[0].length;
  int heightmap_height_px = heightmap.length;

  int num_verts = heightmap_width_px * heightmap_height_px;
  vertNormals = new float[3 * num_verts];

  TODO: Initialize each normal to (0,1,0) so that valid normals can be found at edges

  float delta_x = MAP_SIZE / heightmap_width_px;
  float delta_z = MAP_SIZE / heightmap_height_px;

  for (int row = 1; row < heightmap_height_px - 1; row++) {
    for (int col = 1; col < heightmap_width_px - 1; col++) {

        TODO: Create Vector3f Tx
        TODO: Create Vector3f Tz

        TODO: Calculate Vector3f vertNormal by as the normalized
        cross product of vecNx and vecNz and put in vertNormals
    }
  }

    return vertNormals;
}
```

Let the surface of our heightmap be defined as a function $f(x, z) = y$. Note that y and z variables are swapped to be consistent with the OpenGL coordinates. If we can find two vectors that are tangent to the surface along x- and z-axis directions, the direction of the normal is given by the cross product of these two vectors — see the illustration in Figure 7.

Since we are dealing with a discrete representation of the surface, we need to compute tangent vectors using a discrete approximation, so called central differences. The vector tangent to the slice of the surface along the x-axis can be calculated as:

$$T_x = [2\,\delta_x,\ f(x+\delta_x, z) - f(x-\delta_x, z),\ 0]\ , \tag{2}$$

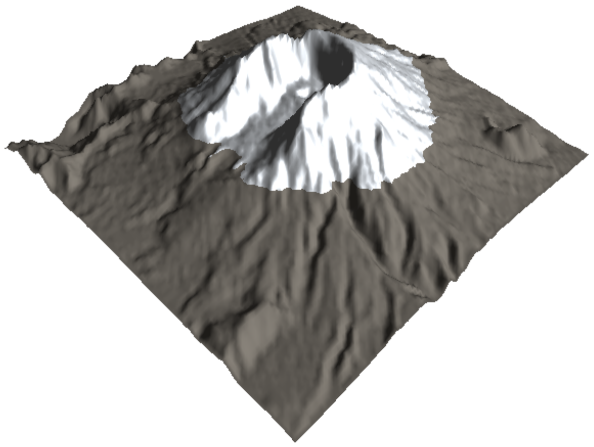


Figure 9: The result you should achieve if you implement the Tick Star option.

where δ_x is the projected distance between vertices along x-axis (refer to the right panel in Figure 5). Use the same value of δ_x as you used for computing vertex positions. The second coordinate of vector T_x (y coordinate) is computed as a central difference: the difference between the height of the next and previous vertices along x. Similarly, the tangent vector T_z can be computed as:

$$T_z = [0,\, f(x, z + -\delta_z) - f(x, z - \delta_z),\, 2\,\delta_z]\ . \tag{3}$$

Then, the normal direction at the vertex can be computed as:

$$D = T_x \times T_z\,, \tag{4}$$

and the normal is the normalized value of this vector:

$$N = \frac{D}{|D|}\,. \tag{5}$$

Note that you can (and should) use the methods Vector3f.cross() and Vector3f.normalize() to compute a cross product and to normalize the vector value. The normals at the edges of the surface should be set to (0,1,0).

7 Tick-Star

Note that this part is optional and you should attempt it only if you completed everything else. To get a tick star mark in this task, you need to modify the shaders and the Java code to render the peak of the volcano in white and with a specular component. You should achieve the result similar to one shown in Figure 9.

The change of shading and material can be easily introduced in the fragment shader. Introduce a conditional statement testing the altitude of the current fragment to decide on the material to use. Note that you need to pass the altitude from vertex to fragment shader. In order to compute the specular color component, you need to compute reflected ray and pass the camera position in the world coordinates from your Java code to the fragment shader.

Use *Tick 3 star* submission item to submit your tick-star. You do not need to submit the regular tick if you completed tick-star. You need to submit 3 files: OpenGLApplication.java and the code for both shaders. Note that the tester is identical for both tick and tick-star, so the outcome of the test does not indicate that the solution is correct. This will be checked during the ticking session.

8 Submission

You need to submit 2 files for this tick:

- fragment_shader.glsl
- OpenGLApplication.java

Note that this time only very basic unit tests will be run by the test script. This is because the servers we use do not have a GPU, which could run OpenGL. The program will be checked in detail during the ticking session.