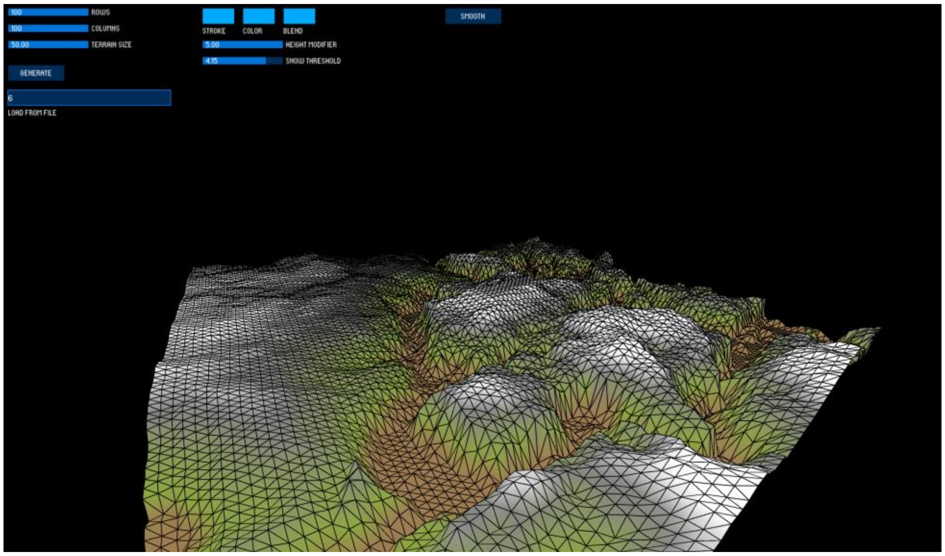
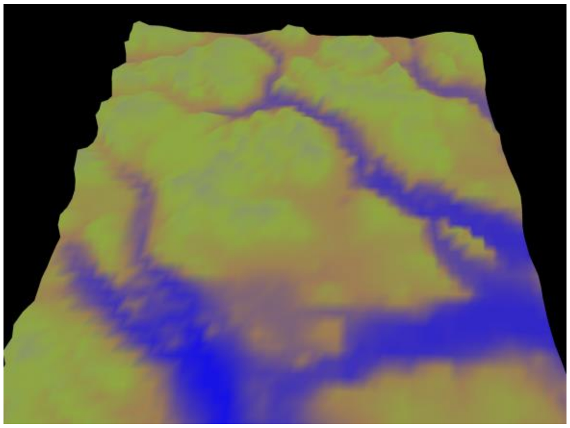
Project 4 – Procedural Terrain

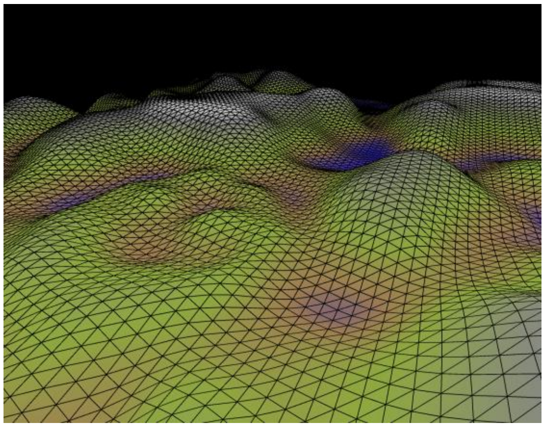
Overview

For this project, you are going to create a grid of vertex and triangle data to create a 3D terrain algorithmically.



100 ROWS
100 COLUMNS
50.00 TERRAIN SIZE
6 LOAD FROM FILE
5.00 HEIGHT MODIFIER
4.15 SNOW THRESHOLD





Description

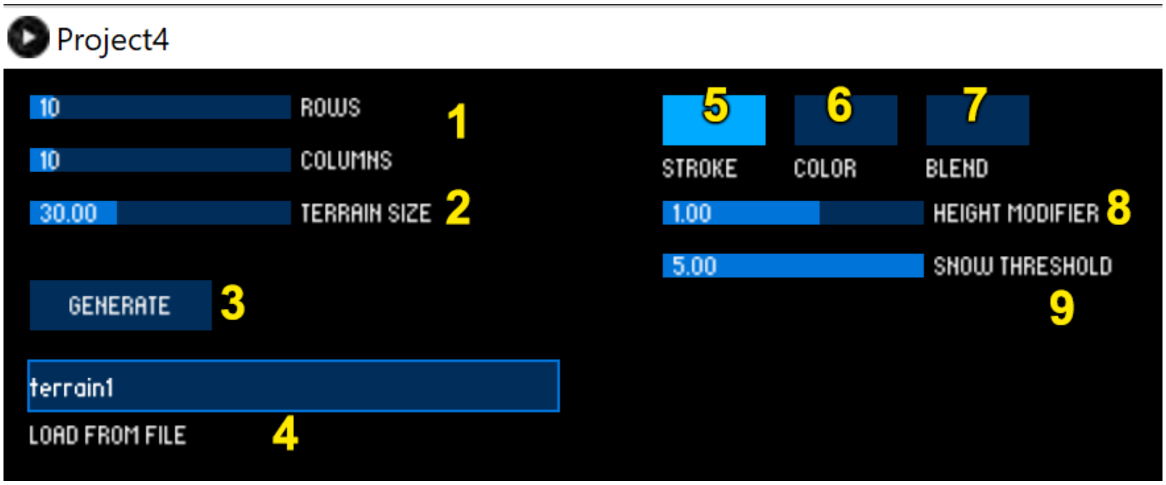
Complex-looking data like terrain can seem daunting at first, but on a basic level, it's really just **a grid of vertices and triangles**. Using that as a base, additional features can be added such as color and changing the elevation of points. Small adjustments to a lot of data simultaneously can make for interesting and impressive results.

In this project there will be three major components:

1. The UI
2. The terrain itself
3. A camera that is very similar to a previously implemented camera.

UI Specifications

The window size for this project is **size(1200, 800, P3D);**



Project4

10 ROWS

10 COLUMNS

30.00 TERRAIN SIZE

1.00 HEIGHT MODIFIER

5.00 SNOW THRESHOLD

terrain1

LOAD FROM FILE

1. Sliders for controlling how many rows and columns of polygons the terrain will have. **Range: 1 to 100**
2. Slider for controlling **total size** of the terrain, or **Grid Size** as it will be referred to elsewhere in this document. Vertices are arranged in a square grid on the X/Z plane, centered at 0, 0, 0. **Range: 20 to 50**
3. A button to generate the terrain according to the previous sliders, plus the name of a file from the textfield. If the textfield (#4) is empty, create a grid of triangles with an elevation (Y-value) of 0.
4. A textfield to allow user input for the name of a file to load. You can set a default value for the control by calling the **setValue()** function, and you should call **setAutoClear(false)** to leave the text in the box if the user hits Enter (which should generate terrain, as if the Generate button as pressed).
5. A toggle for drawing triangles with a stroke or not
6. A toggle to draw the terrain with colored vertices, or solid white
7. A toggle to change between simple vertex color, and blending colors between ranges, based on the height of the vertex
8. A slider to for a value that affects the height of each vertex (a multiplier). **Range: -5.0 to 5.0**
9. A slider to control how high a vertex must be to have the “snow” on it (color value: white).
   Range: 1.0 to 5.0

Camera

The camera in this program will use the orbiting camera from the previous project as a base, with a few slight adjustments. The camera will no longer follow the mouseX and mouseY positions automatically. Instead, you'll adjust the camera only when the user clicks and drags their mouse. To do this, you can use the mouseDragged() function, which is a Processing function that is called when the user has clicked and holds the mouse button down. In that function, you should:

1. Call the function **isMouseOver()** from the your main ControlP5 object. If this function returns true, the mouse is over a control, and you should just return from the function, there's nothing to do.
2. Otherwise, calculate the distance the mouse has moved, and modify your camera's angles accordingly. To do this, you can use the variables mouseX and pmouseX (previous mouse x position), subtracting the previous from the current. The same goes for the y position as well.
   These values alone will likely be somewhat high, and your camera might turn extremely quickly. You can "soften" this by multiplying the result by a modifier. For example:

   ```
   deltaX = (mouseX – pmouseX) * 0.15f; // 0.15f is chosen arbitrarily, modify up or down as you like
   cameraObject.phi += deltaX; // Or pass the value in via a function
   // Repeat for Y, and update theta. Remember to still clamp theta between 1 and 179 degrees
   ```

Because this project is dealing with a larger environment, the range of the camera may need to be adjusted as well. The radius of camera in the demo version has a **minimum value of 10**, and a **maximum value of 200**.

The perspective() function uses the same near and far values of 0.1 and 1000.0, respectively, while the fov is set to 90 degrees (converted to radians when actually being used in the perspective() function).

If your fov is something other than 90, nothing inherently bad will happen, but there may be some slightly noticeable differences at certain camera distances. Without a side-by-side comparison, these differences may be unnoticeable and irrelevant.

Extremely large or small values may cause significant distortion

Terrain generation

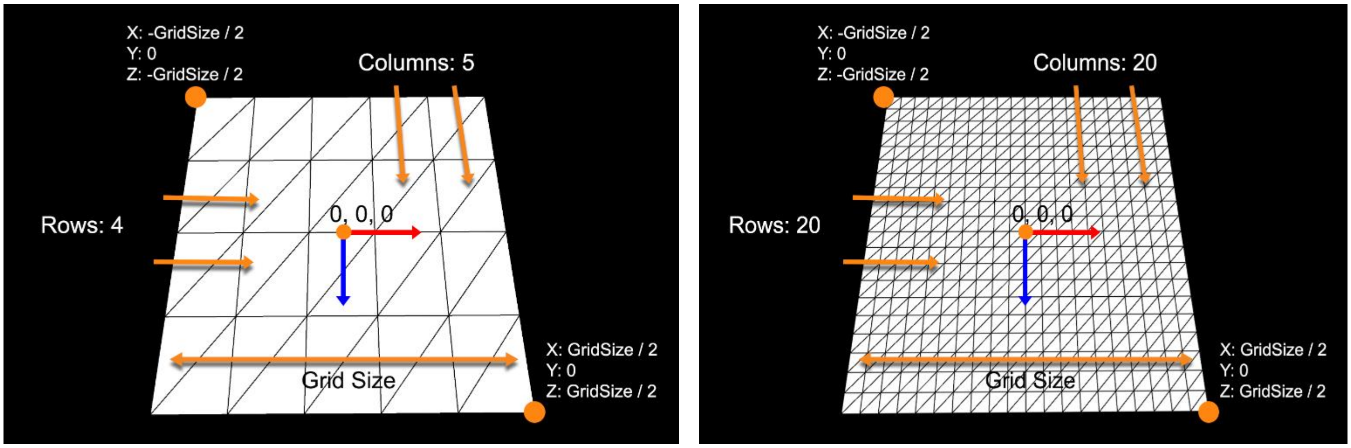
Before diving in, two primary things you will need to store all of this information:

- **ArrayList<PVector>** - The vertex data for the terrain, a list of points
- **ArrayList<Integer>** - The triangle index data. Each triangle needs 3 integers, stored in this container, each of which reference a vertex from the other container

A journey of a thousand polygons starts with a single vertex

The terrain will ultimately be a grid of triangles, which are based off a grid of vertices. Let's start by creating the vertices.

The range of the vertex data will be based on the **Grid Size** variable set by the UI slider. Since the grid is centered at 0, 0, 0 the range will be from –(Grid Size / 2) to +(Grid Size / 2) in both the X-axis and the Z-axis. The number of rows and columns will determine how many subdivisions along the Z-axis and X-axis, respectively.



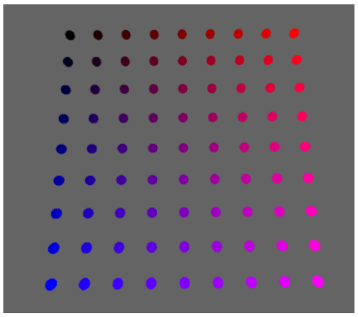
Whether it's a simple mesh or a dense much, much of it will be the same.

You'll want a nested for loop to generate the vertex positions. Whether you loop through rows or columns first doesn't necessarily matter, though it will affect how build the index data later on. It's recommended that you start with rows first, with an inner loop handling column in that row—examples in this document will work off that format. Initially the Y-value of every vertex will be 0 (these will be adjusted later to form the terrain features).

Tip: Create a single row or column first, before trying to solve everything! It's much easier to debug a single row or column that isn't quite right, rather than hundreds or thousands of incorrect points.

Each vertex is spaced out according to the **Grid Size**, divided by the number of rows or columns, depending on which axis we're dealing with. So if the total size was 30 (the default), with 5 rows and 6 columns, each row of vertices vertices would be 30/5 units apart from each other, and each column of vertices would be 30/6 units apart.

After initially generating the values for each vertex, it would be a good idea to draw SOMETHING at those locations (a simple box or sphere will suffice). Otherwise, we just find ourselves looking at the void! Also, having some sort of color-coding to indicate axes or values can be helpful, but is not required for this assignment. (Remember the grid from the previous assignment? Same concept!)



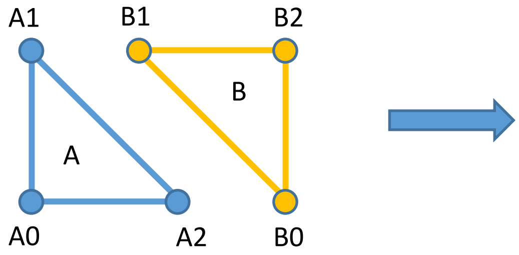
Color-coded spheres, mapping X/Z values to R/B values – brighter colors are located at positive X/Z values

Generating Triangle Indices

Once the vertices are done, you will have to build triangles out of them. Before we can do that, you have to understand what a **triangle index** is. This can take a bit of explanation, so keep reading!

An index is simply an integral value that, along with two other values, indicates which vertices should be used from a list of vertices to form a triangle. To understand these values we have to re-examine how triangles are drawn.

For example, say we have two triangles, A and B. If we wanted to store the data for their vertices, we could simply create an array and add 6 points to it.



```
PVector[] points = new PVector[6];

points[0] = new PVector(A0x, A0y, A0z);
points[1] = new PVector(A1x, A1y, A1z);
points[2] = new PVector(A2x, A2y, A2z);
points[3] = new PVector(B0x, B0y, B0z);
points[4] = new PVector(B1x, B1y, B1z);
points[5] = new PVector(B2x, B2y, B2z);
```

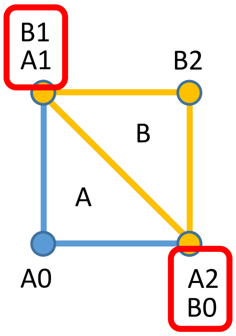
Three vertices per triangle is standard stuff, no surprises here. Drawing the two triangles is easy enough:

```
beginShape(TRIANGLES);

// Triangle A
vertex(points[0].x, points[0].y, points[0].z);
vertex(points[1].x, points[1].y, points[1].z);
vertex(points[2].x, points[2].y, points[2].z);

// Triangle B
vertex(points[3].x, points[3].y, points[3].z);
vertex(points[4].x, points[4].y, points[4].z);
vertex(points[5].x, points[5].y, points[5].z);

endShape();
```

Or, we can just loop through the array

```
beginShape(TRIANGLES);

for (int i = 0; i < points.length; i++)
    vertex(points[i].x, .y, .z);

endShape();
```

What if the two triangles were adjacent? What if they shared an edge, and two vertices were right on top of each other?

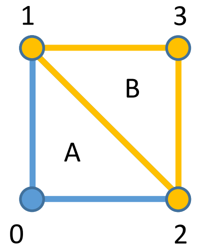


```
PVector[] points = new PVector[6];

points[0] = new PVector(A0x, A0y, A0z); // Unique
points[1] = new PVector(A1x, A1y, A1z); // Unique
points[2] = new PVector(A2x, A2y, A2z); // Unique
points[3] = new PVector(B0x, B0y, B0z); // Redundant
points[4] = new PVector(B1x, B1y, B1z); // Redundant
points[5] = new PVector(B2x, B2y, B2z); // Unique
```

Total vertices: 6 | Total **UNIQUE** vertices: 4

Repeating two vertices, in the grand scheme of things, is pretty trivial. However, if a large mesh were made of 60,000 vertices, and we had 20,000 redundant points… not quite as trivial. Not only is it efficient from a memory usage perspective, it can also be beneficial to share vertices so that triangles appear connected, and can transform together.

Rather than duplicate vertices, we might build a mesh out of just 4 points, like this:



Maximum vertex efficiency

The vertex data alone won't be sufficient. To draw a triangle requires 3 vertices. From this example:

- Triangle A is made up of vertex[0], vertex[1], and vertex[2]
- Triangle B is made up of vertex[1], vertex[3], and vertex[2]

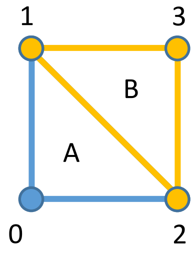
How do we store this information and not just hard-code a simple example? How can we also communicate this storage to a rendering system? What do we actually need to store for each triangle?

Look at what is different between those two triangles:

A: vertex[0], vertex[1], and vertex[2] → 0, 1, 2
B: vertex[1], vertex[3], and vertex[2] → 1, 3, 2

Those three integers are index values, and represent vertices from some other list—the 3 points of a triangle.

Drawing index-based triangles



```
PVector[] points = new PVector[4];

// Assume appropriate xyz values
// for 0x, 0y, etc
points[0] = (0x, 0y, 0z);
points[1] = (1x, 1y, 1z);
points[2] = (2x, 2y, 2z);
points[3] = (3x, 3y, 3z);
```

```
int[] triangles = new int[6];

// From the list of points…
// Which points for Triangle A?
triangles[0] = 0; // points[0]
triangles[1] = 1; // points[1]
triangles[2] = 2; // points[2]

// Which points for Triangle B?
triangles[3] = 2; // points[2]
triangles[4] = 1; // points[1]
triangles[5] = 3; // points[3]
```

These are triangle indices.
They **index** a vertex array.

In order to draw any triangle, we have to provide three vertices. However, to STORE the data for any given triangle, we don't necessarily need to store three vertices for each. Instead, we could store an array with all the vertex data for the entire mesh, and then store a separate array of integer values, which **references** the first array. This second list will contain **three index values for every triangle**, and the total size of the array will ALWAYS be: NumberOfTriangles*3. This is sometimes referred to as an **index-based triangle list**.

If we're using index values, the process of drawing a triangle will be slightly different. We still need to provide three vertices to form a triangle. What's different is how we access those triangles. We use a value from the array of indices to look up the array of vertices.

```
beginShape(TRIANGLES);

// Triangle A
// index == 0, vertex == 0
vertex(points[triangles[0]].x, points[triangles[0]].y, points[triangles[0]].z);
vertex(points[triangles[1]].x, points[triangles[1]].y, points[triangles[1]].z);
vertex(points[triangles[2]].x, points[triangles[2]].y, points[triangles[2]].z);

// Repeat for Triangle B...

endShape();
```

This is not the prettiest code in the world…

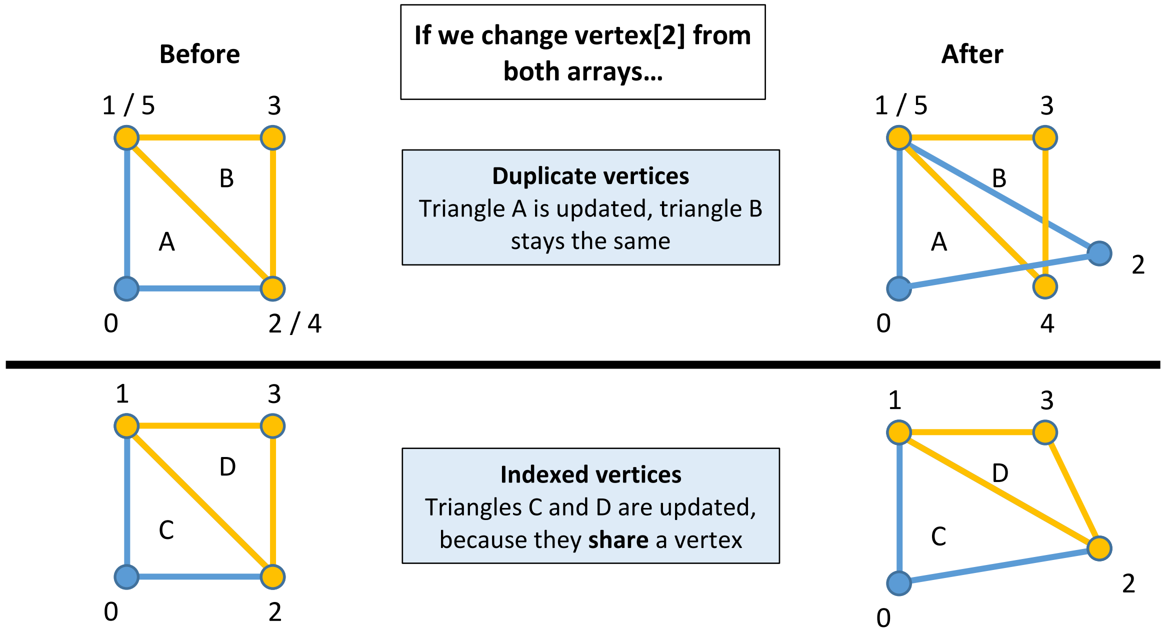
```
beginShape(TRIANGLES);

for (int i = 0; i < triangles.length; i++)
{
    // Using the current value from the index array...
    // Look up a vertex from the points array
    int vertIndex = triangles[i];
    PVector vert = points[vertIndex];

    vertex(vert.x, vert.y, vert.z); // A bit cleaner!
}
endShape();
```

Triangles or Indices?

It's just a terminology difference – you may see some people call this list of integers "triangles" (which might not seem accurate) or "indices" which may be more intuitive to some.

Visual benefits of index-based triangles

Let’s take two separate pairs of triangles. The first pair, A and B, have their “own” three vertices. The second pair, C and D, use an index-based list.

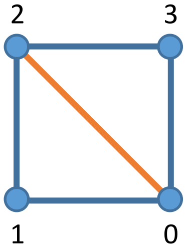


If we change vertex[2] from both arrays…

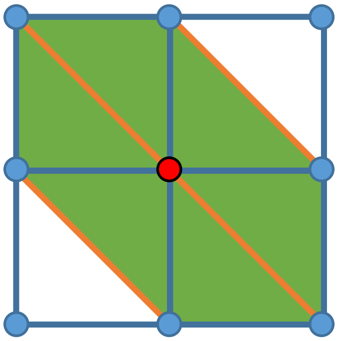
Duplicate vertices
Triangle A is updated, triangle B stays the same

Indexed vertices
Triangles C and D are updated, because they **share** a vertex

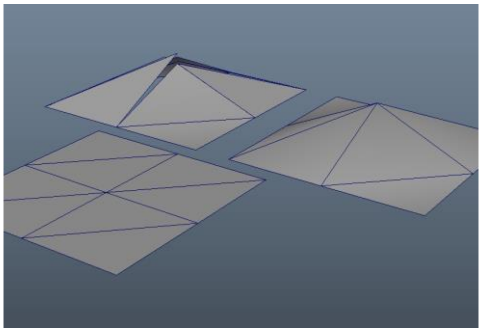
A more complex example:



Triangles A and B share vertices 0 and 2



The vertex in the middle is shared by a whopping 6 triangles!



Right: a single center point
Left: multiple, separate vertices can result in gaps

Moving a single point versus having to move six separate points

Is one of these better than the other?

It depends on your application, but there may be times when you want one of these options. If a mesh is going to break apart, it may be beneficial to have duplicate vertices, so you don’t need to split geometry at runtime.

On the other hand, a mesh representing terrain with smooth hills or other features would likely be best if data was “connected” in order to make deformations easier. Why move multiple points when a single point will accomplish the same task?

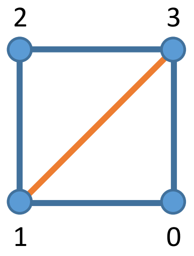
Actually creating triangle indices

For a small number of vertices, we can often hard-code index values. For example:

For example:

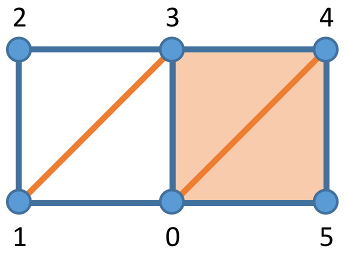
Assume this as your container to store indices

```
ArrayList<Integer> triangles = new ArrayList<Integer>();
```



```
triangles.add(0);
triangles.add(1);
triangles.add(3);

triangles.add(3);
triangles.add(1);
triangles.add(2);
```

Adding the first quad



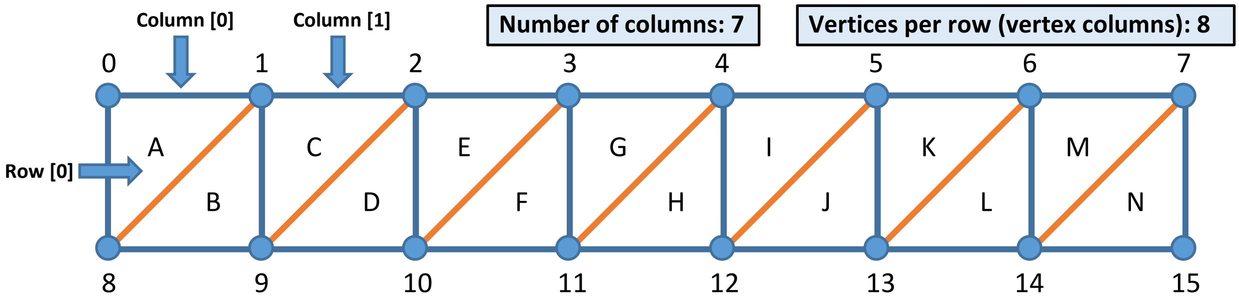
```
triangles.add(5);
triangles.add(0);
triangles.add(4);

triangles.add(4);
triangles.add(0);
triangles.add(3);
```

Adding the next quad

Four triangles worth of data, no problem! What about more than this? What about hundreds, or thousands? What about a row of triangles for, say, a procedurally generated terrain?

Number of columns: 7

Vertices per row (vertex columns): 8



For triangle A, what should its indices be?
0, 1, 8?
8, 0, 1?

Triangle B?
1, 9, 8?

Instead of hard-coding a particular value, we need to calculate that value from other variables. For example, can the numbers 0, 1 and 8 be derived from anything?

It is helpful to define a starting index, and then base other values off that.

startIndex = row * verticesInARow + currentColumn

With a starting index calculated, our triangle indices can be set easily:

Triangle A:	**Triangle B:**	**Triangle G (startIndex == 3)**
startingIndex (0)	startingIndex + 1 (1)	startingIndex (3)
startingIndex + 1 (1)	startingIndex + verticesInARow + 1 (9)	startingIndex + 1 (4)
startingIndex + verticesInARow (8)	startingIndex + verticesInARow (8)	startingIndex + verticesInARow (11)

This will also work for subsequent rows. Test it out with just a few quads at a time, so you can slowly step through the process! Every time you calculate an index, add it to your ArrayList (3 per triangle, 6 per quad).

SIDE QUESTION: Does it matter what order the indices are, for any given triangle? Does it matter if the first triangle is [0, 1, 8] or [8, 1, 0]?

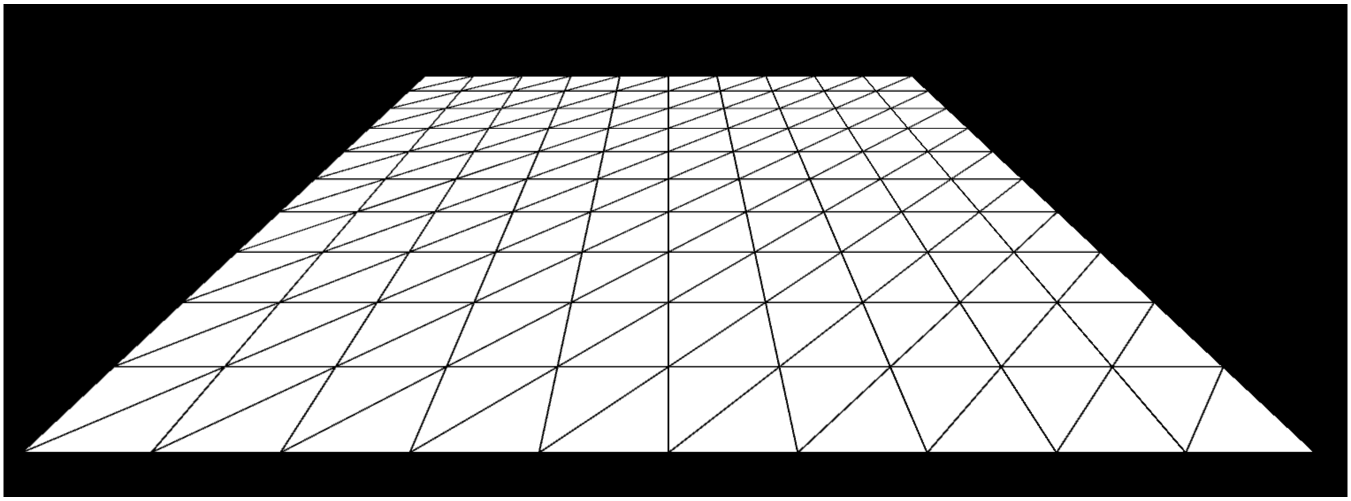
No, the order doesn't matter, with a few caveats:

1. It COULD matter, due to something called **back-face culling**. This is an optimization technique that avoids rendering polygons that are facing away from the camera (essentially: don't draw a polygon if we only see the back of it). Determining the "direction" of a face is due to what's called the winding order of a polygon. Its vertices will be in a clockwise, or counter-clockwise order, and the culling settings will use one order, and "ignore" the others.

   So [0, 1, 2] might be a correct clockwise order, but [2, 1, 0] might be an incorrect counter-clockwise order. This option is **off** in Processing by default, and so any winding order will still display correctly. It is all but guaranteed that you will have to deal with this in other rendering environments. Learn what the "correct" order is for that environment, and plan your algorithm accordingly.

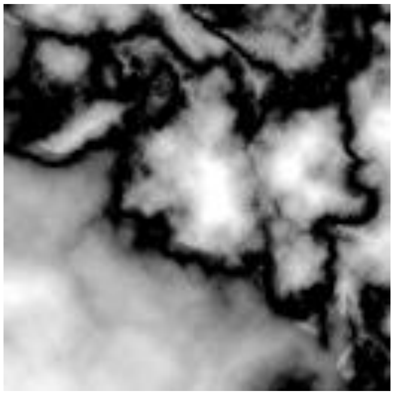
2. The order you choose will likely affect how you write your algorithm. Your first triangle might be [0, 1, 2], and your second triangle is [3, 0, 2]. No problem. The THIRD triangle… will need to know how the second triangle was built (i.e. how YOU chose to build it).

If you implement it correctly (and draw them correctly), you should see a nice grid of triangles:



The next step is to actually DO something with them!

Raising the terrain



For this step you'll use something called a **heightmap** to determine the elevation of each vertex. A heightmap is just a grayscale image where a darker value is a lower elevation, and a brighter value is a higher elevation.

Try to imagine the image on the right as an overhead shot of an area. The dark spots could be rivers or canyons, and the lighter areas could be hills or mountains.

The next step will be to load this image, read in pixel values, and then based on how bright a pixel is, adjust the height of a vertex in the terrain.

Loading an image

Images in Processing can be loaded very simply. There is a class called **`PImage`**, and an instance of that class can be set to the return of the function, **`loadImage(filename)`**;

The images you load will have to be in a "data" folder wherever you sketch lives, and all you need to provide is the filename. The filename you'll retrieve from the textfield you created with the ControlP5 library.

There are 7 different images you can use for testing your program, named **terrain0.png** → **terrain6.png.** Be sure to try them all out, some of them will give very different results!

Once an image is… what do we do? Make heavy use of the **`map()`** function!

Mapping vertices to pixels

On the one hand, a 2D grid of vertices. On the other hand, a 2D grid of pixels…

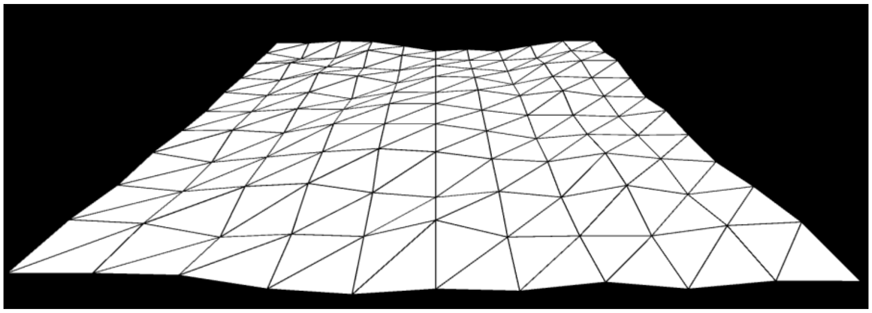
For every vertex (which we can consider as having a spot in a 2D array, even if it's a 1D ArrayList)

- Map the row/column index of that vertex to a row/column value of the image, based on the image's width and height
- Use those index values to look up a color from the image, using the **.get(x, y)** function

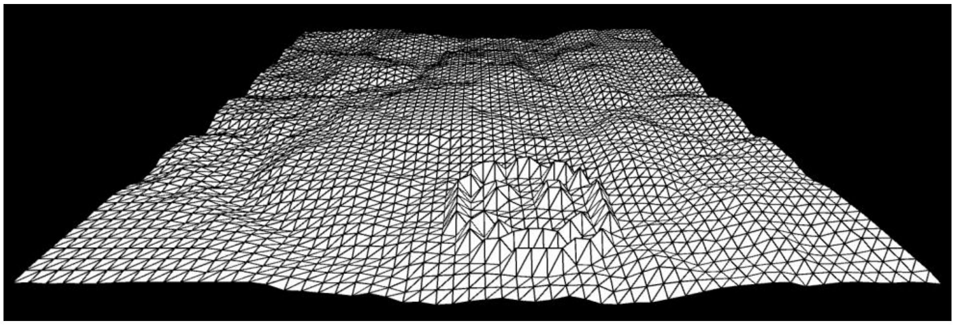
Pseudocode for this algorithm:

```
for (i <= rows)
{
    for (j <= columns)
    {
        // rows/columns +1 because there are more vert columns than polygon columns
        x_index = map j from 0 → columns+1 to 0 → imageWidth
        y_index = map i from 0 → rows+1 to 0 → imageHeight)
        color = image.get(x_index, y_index)

         // red/green/blue will all be the same, as the image is grayscale
        heightFromColor = map red(color) from 0 → 255 to 0 – 1.0f

        vertex_index = i * (columns + 1) + j
        vertices[vertex_index].y = heightFromColor
    }
}
```

After that has been completed, you should have terrain that looks a bit like a slightly wrinkled sheet of paper (depending on the image loaded).

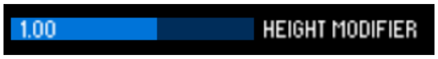


The detail seems minimal because the polygons themselves are fairly large. With a default 10x10 grid and a 30 unit terrain, each polygon is 3x3 units. Increasing the density of the terrain will highlight the details a lot more.



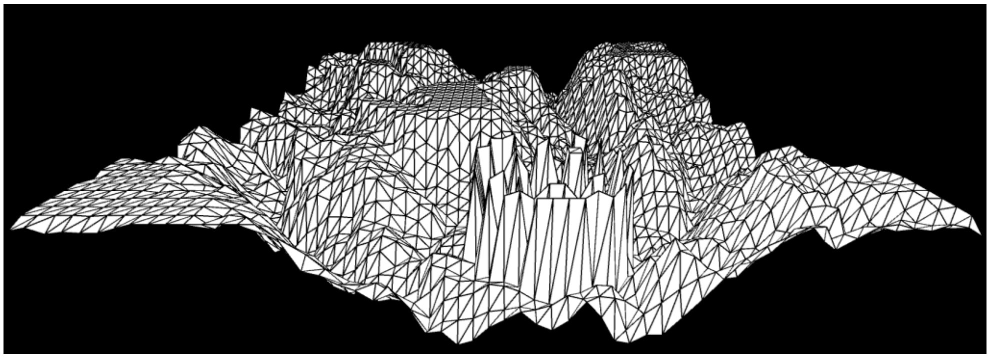
The same terrain, but with a 55x55 grid

The height modifier slider can also adjust the terrain's height. When drawing a vertex, simply use the value from this slider as a multiplier for the y value of a vertex.



1.00 HEIGHT MODIFIER

```
vertex(someVertex.x, someVertex.y * heightModifierValue, someVertex.z);
```



55x55 terrain, with a 5.0 height modifier

Coloring vertices

Plain white triangles are one thing, but what we really want is a nice looking terrain. This could be done in countless ways, and the approach here is by no means the only, nor the best. It is, however, relatively simple!

We start by calculating the relative height of any given vertex (this should be done when drawing the terrain, before a call to the vertex() function). For this, we'll make use of the "snow" slider. As the y-value of a vertex approaches (or exceeds) this value, its color will be white, like a snow-capped mountain.

5.00 SNOW THRESHOLD

The relative height of a vertex:

relativeHeight = absolute value of vertex.y * heightModifier / -snowThreshold

Why a negative threshold? Because Processing is a bit odd and uses negative Y as an up axis, that's why!

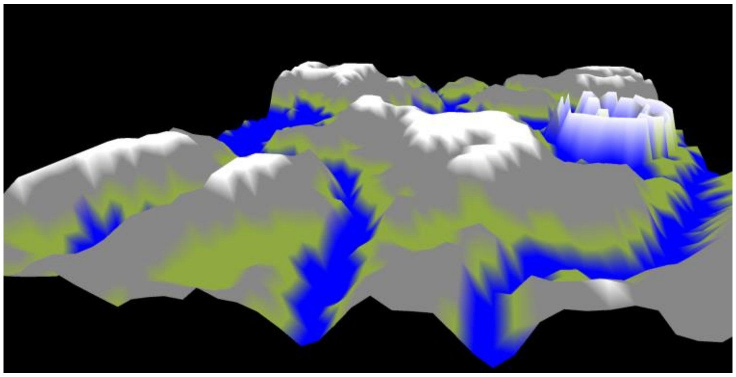
Then, based on the value of that relativeHeight…

First, let's define some colors:

Snow (255, 255, 255)	Rock (135, 135, 135)	Grass (143, 170, 64)	Dirt (160, 128, 84)	Water (0, 75, 200)
⬜	◼	🟩	🟫 (This will come into play later)	🟦

```
If the height is between 1.0 and 0.8
        It's a white (snow) vertex – use the snow color
If the height is between 0.4 and 0.8
        It's a gray (rock) vertex – use the rock color
If the height is between 0.2 and 0.4
        It's a green (grass) vertex – use the grass color
Otherwise…
        It's a blue (water) vertex – use the water color
```

The ranges here are arbitrary, as are the color values. They give a decent approximation, but a more complex calculation, one that takes the adjacent colors into account might be a bit better.



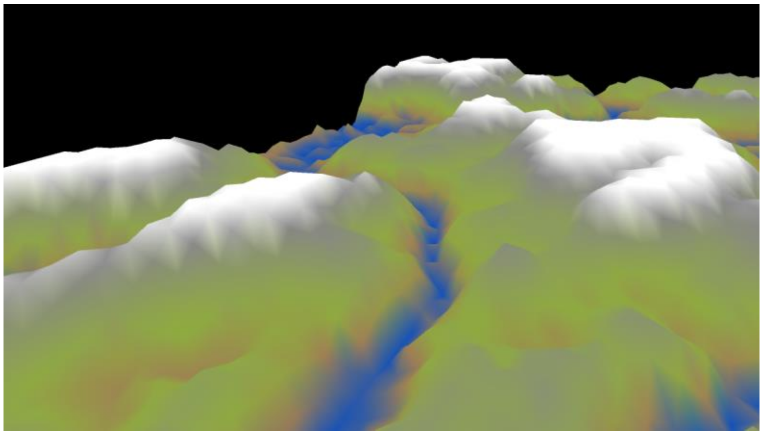
Looking better! But still a bit lacking…

Linear interpolation could be used here! Instead of just setting a color, we calculate one! Fortunately, Processing has a function called **lerpColor(colorA, colorB, ratio)** that will do just that for us. It would be helpful to define the various colors as actual variables, otherwise this code can turn into a jumbled mess.

```
color snow = color(255, 255, 255);
color grass = color(143, 170, 64);
color rock = color(135, 135, 135);
color dirt = color(160, 126, 84);
color water = color(0, 75, 200);
```

So now, the previous algorithm can turn into this:

```
If the height is between 1.0 and 0.8
    If we're blending colors…
        ratio = (heightValue – 0.8) / 0.2f         // 0.2f because that's the range of "snow" values, 0.8 to 1.0
        color = lerpColor(rock, snow, ratio)       // "lower" color first
    else
        color = snow
If the height is between 0.4 and 0.8
    If we're blending colors…
        ratio = (heightValue – 0.4f) / 0.4f
        color = lerpColor(grass, rock, ratio)
    else
        color = rock
If the height is between 0.2 and 0.4
    If we're blending colors…
        ratio = (heightValue – 0.2f) / 0.2f
        color = lerpColor(dirt, grass, ratio)
    else
        color = grass
Otherwise…
    If we're blending colors…
        ratio = heightValue / 0.2f
        color = lerpColor(water, dirt, ratio)      // This allows for a "bank" or "coastline" transition
    else
        color = water
```



Much nicer!

Going beyond this

There is a lot that can be done with procedural content of any flavor, and this is just one example. The concept of generating data through an algorithm instead of manually placing a single point or group of values at a time is very powerful.

With more variables, more data you could generate specific types of features like rivers, island chains, craters, and more. Multiple terrains could be created and strung together, multiple images could be blended together based on program data, and new terrains generated from that.

Rendering styles could change as well. You could add in a wireframe mode, change the color blending patterns, add the ability to set custom elevation colors—maybe an unusual color scheme for an alien world!

Tips while working

Tackle one piece at time, and verify that that part works before moving on to the next! Especially when getting started. Draw one or two polygons from your vertices, then try handle an entire row.

Break your code into separate pieces! Create functions or even classes to help you separate data where applicable. Generating the triangles is one step. Creating indices for those triangles could be a separate step. Setting colors for vertices based on their height could be a third, etc. Don't write monolithic, 500-line functions.

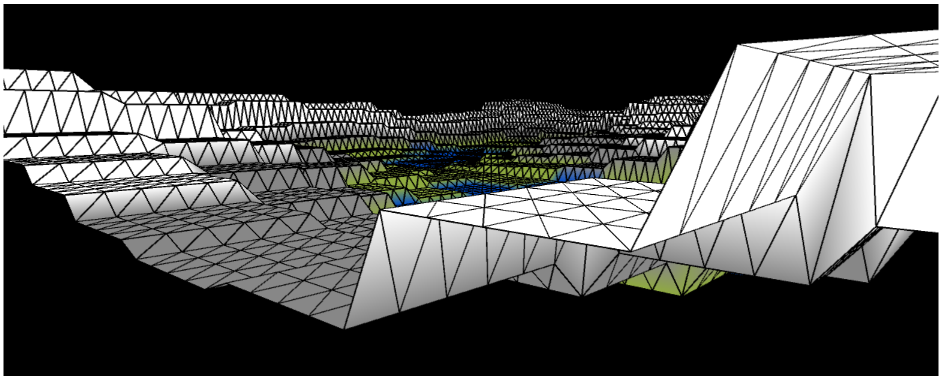
In some cases, given the nature of some operations, it might be more sensible to keep parts in the same function or loop.

Grading

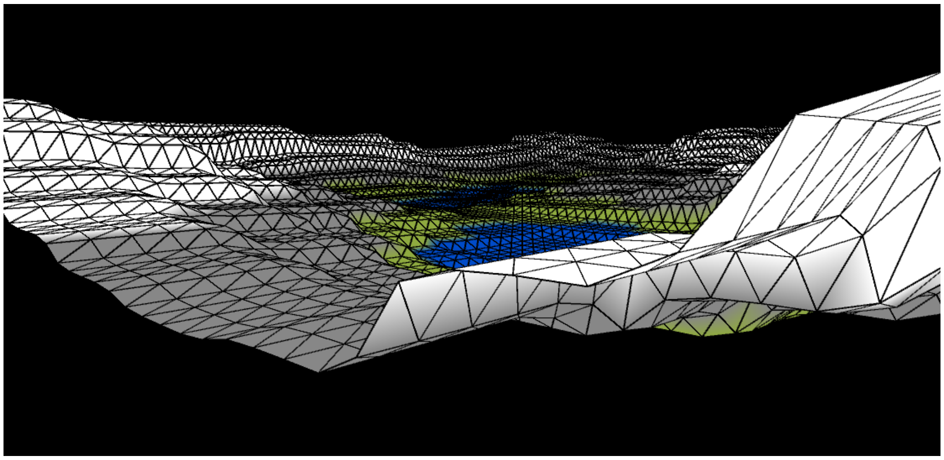
Item	Description	Max Points
Grid Resizing	Grid can be resized using the Rows, Columns, and Terrain Size sliders, and the Generate button	20
Loading files	Terrain can be modified by loading a heightmap entered in the textfield	30
Camera	Orbit camera implemented, moves when the user clicks and drags their mouse Don't reinvent the wheel, use your previous implementation as a base, and modify it!	10
Height modifier	Height modifier slider implemented and functional: Terrain can be raised or lowered based on the value of the slider (up for positive values, down for negative values)	10
Snow threshold	Snow threshold slider implemented and functional. "Snow" can be increased or reduced appropriately according to the snow threshold	10
Wireframe toggle	Toggle drawing wireframes over the terrain	5
Color toggle	Toggle between white polygons and colored terrain. Colored terrain is draw with 4 "levels" according to the project spec - water, grass, rock, snow	5
Blending implemented, can be toggled	Switch between basic color and interpolated color. Interpolated color "lerps" between color levels, and adds a dirt/mud later between water and grass	10
	Total	100
Extra Credit	Implement a smoothing function to even out rough parts of the terrain (see below). Add a "Smooth" button to execute this algorithm. Note: Multiple executions of the algorithm (i.e. button clicks) will be necessary to completely smooth something, as each "pass" will only smooth a little.	+10
	Total with Extra Credit	110

Extra Credit (+10 points)

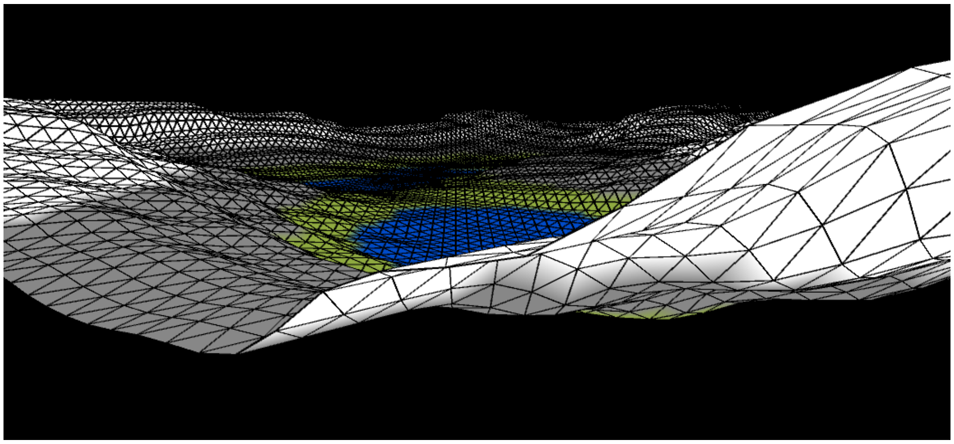
If everything else is completed, an additional feature you can implement is a smoothing algorithm. Depending on the terrain, you may end up with some jagged lines. This is particularly noticeable with small input textures and high-resolution grids. In these cases, multiple vertices sample the same pixel from an image. The same color input results in the same Y-value as output.



Interesting, and maybe exactly what you want! Or… not at all



After one “step” of smoothing, harsh edges start to soften



After two more steps, hard edges all but gone!

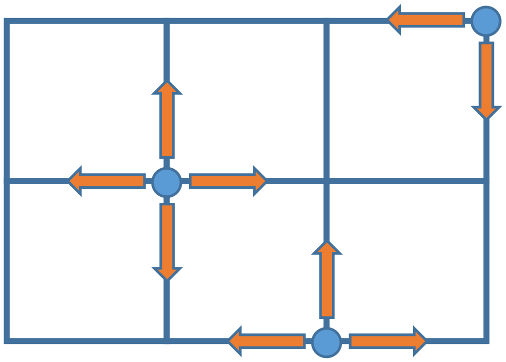
Smoothing Algorithm

The algorithm for smoothing is fairly simple, but implementing it may be tricky as there are a lot of edge cases.

For each vertex:

- Sample the position of the vertex and its neighboring vertices
- Take the average of all of the y-values s of each of those vertices
- Assign the average value to the y-value of the current vertex

A vertex and its neighbors



Vertices that are “inside” the grid have 4 neighbors, while others on the edges may only have 2 or 3.

The algorithm is the same though:

Take the average of a vertex and all its neighbors (however many there are)

For this algorithm, neighbors will be in the 4 cardinal directions: 1 above, 1 below, 1 to the left, and 1 to the right (assuming we aren’t going off the grid).

This algorithm could also be implemented by taking the “diagonal” vertices into account as well, which would give different results. For this assignment, only the 4 neighbors are required.

There’s a small twist: You should only sample the values of the vertices as they were **before** the algorithm was executed. If you change a vertex in the middle of this algorithm, this new value will skew the average of subsequent vertices. Which will skew the average, which will skew the average…

It may be necessary to store a copy of the vertices and update that copy just after creating the terrain, before executing this algorithm, just after executing this algorithm… Sample the “before” vertices, then apply to the results to the “now” vertices.

The most difficult part of an algorithm like this is typically in dealing with the edge cases. Structuring a loop to handle vertices with only 2 or 3 neighbors can often be a bit tricky, though sometimes the results might be interesting, even desirable (if you were going for some other effect…)