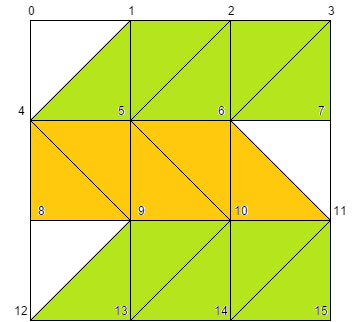
# Heightmap

Before we can generate a heightmap, we first need a grid of triangles. The problem is rendering each triangle itself is pretty performance intensive. It’s better to render all triangles of the grid with one “triangle strip”.

**How to generate a grid with one TRIANGLE\_STRIP**

If we consider a n\*n-matrix we do have (n+1)^2 vertices points.  
In a short 3x3 example we will show how this algorithm works:



In each row we switch the direction of the diagonal lines, because otherwise many indices would be used multiple times and it wouldn’t end up in such a nice index distribution:

0, 4, 1, 5, 2, 6, 3, 7, 11, 6, 10, 5, 9, 4, 8, 12, 9, 13, 10, 14, 11, 15

At the transition from one row to another (what means the diagonal direction changes) there are no real triangles (as you can see at the indices 3,7,11 or at 4,8,12). There no triangles are rendered, but most of the time you won’t even notice that they are missing.

To form code out of these indices we need to find the pattern in there. Rearranging these indices helps a lot:

0,  
4, 1, 5, 2, 6, 3, 7,  
11, 6, 10, 5, 9, 4, 8,  
12, 9, 13, 10, 14, 11, 15

The resulting algorithm (in pseudocode) looks like this:

Add 0 to the list  
Iterating through each row   
 if (row is odd)  
 start at column 0  
 add the bottom index (at column 0) and then the top index (of the next column) to   
 the list in an alternating manner going to the right until the right side is reached

Else (row is even)  
 start at the last column  
 add the bottom index (from the last column) and then the top index (from the   
 previous column) to the list in an alternating manner going to the left until the left   
 side is reached

**Shaders of the heightmap**

For a heightmap the y-values are the important ones, because these create the high “mountains” of the map. To create the y-values from an “black/white”-image we first give the vertex shader the image via an uniform of type sampler2D.  
  
With the function “texture2D(imageTexture, vec2(x, y)) you get the color value of this specific point of the image texture. We use this color as the y-position (instead of the attribute “a\_position.y”). With this, we have a vector, where the y-coordinate depends on the color of the texture image.  
  
To give the heightmap a nice texture we do the same as before. Give the fragment shader an image with an uniform of type sampler2D and use get the right color with the function “texture2D(…)”.

# Particle System

A particle system is a collection of a huge number of objects. The most common objects are triangles, rects and points. These objects all have different properties, for example different velocities, colors, lifespan and so on. To realize this one has to indicate different variances. The particle system itself also has some properties like how many new objects should be created at once or can destroyed objects be recreated. Finally you can make fire or smoke with particle systems.

I decided to us points for the objects. The disadvantage of rects and triangles are that they have no depth. So you only can see it from a distinct viewing angle. Solutions of course would be billboarding or 3D objects (but this would be a rendering overkill). When using points you have to ensure that you had set the gl\_Point property to something greater than zero, otherwise you won’t see anything.

There are also different kinds of calculating the final positions and colors of the objects. First you can compute everything in JavaScript, e.g. add the velocity to the base position. Then you only need buffers for all positions and colors, which you can connect to the shader program. But this method means that every position and color is computed on the CPU. The better approach is that you have buffers for all properties. One for positions, one for colors, one for velocities, one for the rest of the lifetime of the object and so on. So you are able to calculate the final position in the shader program, which means that it is processed on the GPU, which means that the processing of all objects is strongly parallel.

My properties a particle system can have are: max particles, spawn number, spawning position, variance of spawning position, color, variance of color, velocity, variance of velocity, lifespan and the ability of recreation. For example, to calculate the real spawning position of an object is very easy:

Math.random() creates a number between 0 and 1. When you subtract 0.5 you have a value between -0.5 and 0.5. Finally you just have to multiply it with the variance and add it to the base.