REPORT

MOUNTAIN HIKING

Kiousis Panagiotis, ID: 1092647

TASK 1: Creation of the 3D Terrain

To create the terrain, I used a heightmap (a top-down grayscale image of a region). In a heightmap, the brightness of each pixel corresponds to the elevation of that point (the whiter it is, the higher the altitude). As a heightmap, I used the region of Attica. (Source: Tangram Heightmapper)

Using the function **SOIL_load_image()**, I stored the image data in the 1D array *data*, and stored the height and width of the image in *height* and *width*, which also correspond to the terrain's dimensions along the z and x axes.

By storing the image data in a 1D array, the element located at (z, x) in the image has index:

index = z * width + x

Vertex creation

Using a double for-loop (z < height, x < width), I generated each vertex as follows:

- vx = -width/2 + x (This offset centers the terrain at (0,0) in the x–z plane.)
- vz = -height/2 + z
- vy = heightValue = data[z * width + x] * yScale
  (yScale controls the terrain's maximum height, knowing data values are max 256.)

I stored each vertex's coordinates in the **vertices** array.
Additionally, I created a **heights[]** array to easily retrieve the terrain height at any (x,z):

heights[z * width + x] = heightValue;

Normal generation

For each vertex v0 with coordinates (x, z), I find the vertices at (x+1, z) and (x, z+1). From these, I define two vectors: v1 - v0 and v2 - v0.

The normalized cross product of these vectors gives the normal vector to the triangle's surface.

This vector is added to the total normal of each of the three vertices.

Because each vertex can participate in several triangles, the vertex's final normal is the normalized sum of the individual normals from every triangle.

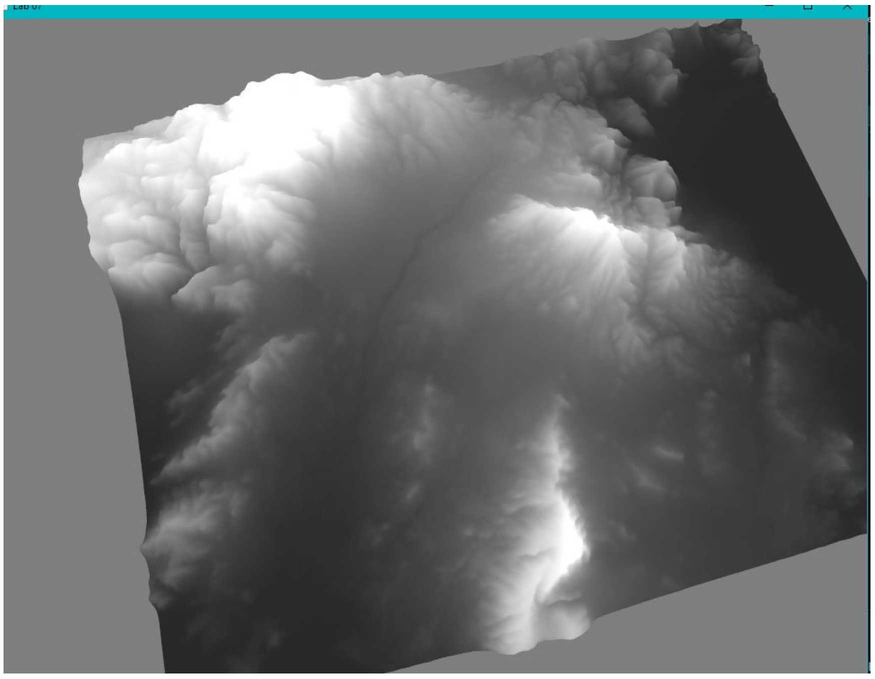
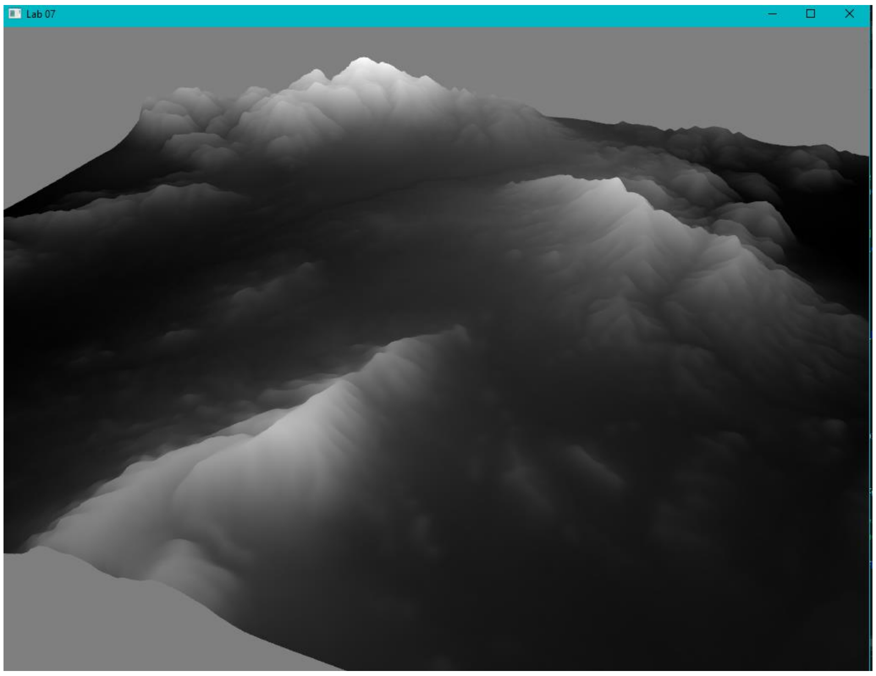
Creation of the Indices:

The use of indices is significant because this method (and the use of IBO) leads to more efficient rendering as we do not need to store each vertex multiple times depending on the triangles it belongs to. We define pointers in the vertices array to group the rendering into triangles, using these pointers to access the vertices for each triangle.

- For each vertex with coordinates (x, z), I find the vertices at (x+1, z), (x, z+1), and (x+1, z+1). The square formed is then split into two triangles, and the indices of their vertices are pushed back into the Indices array.

After the necessary shader operations, the world is drawn in the mainLoop using the glDrawElements() function. Initially, I defined the color in the fragment shader to depend on the height of the point.

The result is shown below:

TASK 2 : Texture the terrain

Initially, I had to create **UV coordinates** for the terrain. The starting idea was to make the texture repeat a number of times (textureRepeatFactor) across the terrain:

- float u = (float)x / width * textureRepeatFactor;
- float v = (float)z / height * textureRepeatFactor;

However, I wanted a different repetition frequency depending on the texture, so I repeated the process for a second textureRepeatFactor2. Also, to make the repetition of the image less noticeable, I applied **random rotation** to the UV coordinates. To avoid having a unique random rotation for every single point, I grouped the points into **blocks** based on the repetition factor and applied a random rotation to each block using one of the pre-defined random angles I had previously specified.

- float uRotated = u * cosTheta - v * sinTheta;
- float vRotated = u * sinTheta + v * cosTheta;

In each case, I pushed back the UVs to the textcoords or textcoords2 array.

I used three different textures for the terrain:

1. A **terrainTexture** (ground with stones and grass).
2. A **snowTexture** (for snow).
3. A **waterTexture** (for water).

To make the water appear to move, I used a mix of two textures of the same water image, with the UVs of one of them changing over time.

I defined which textures would be used based on the height. The case of high altitude (>50) is interesting, where a mix of snowTexture and terrainTexture occurs:

- float blendFactor = (Height - 50.0f) / (100.0f - 70.0f);
- textureColor = mix(groundColor, snowColor, blendFactor);

I want the snow blend to reach full intensity (100%) before reaching the maximum height so that the transition appears relatively abrupt. This is why '70' is present in the blendFactor instead of '50'.

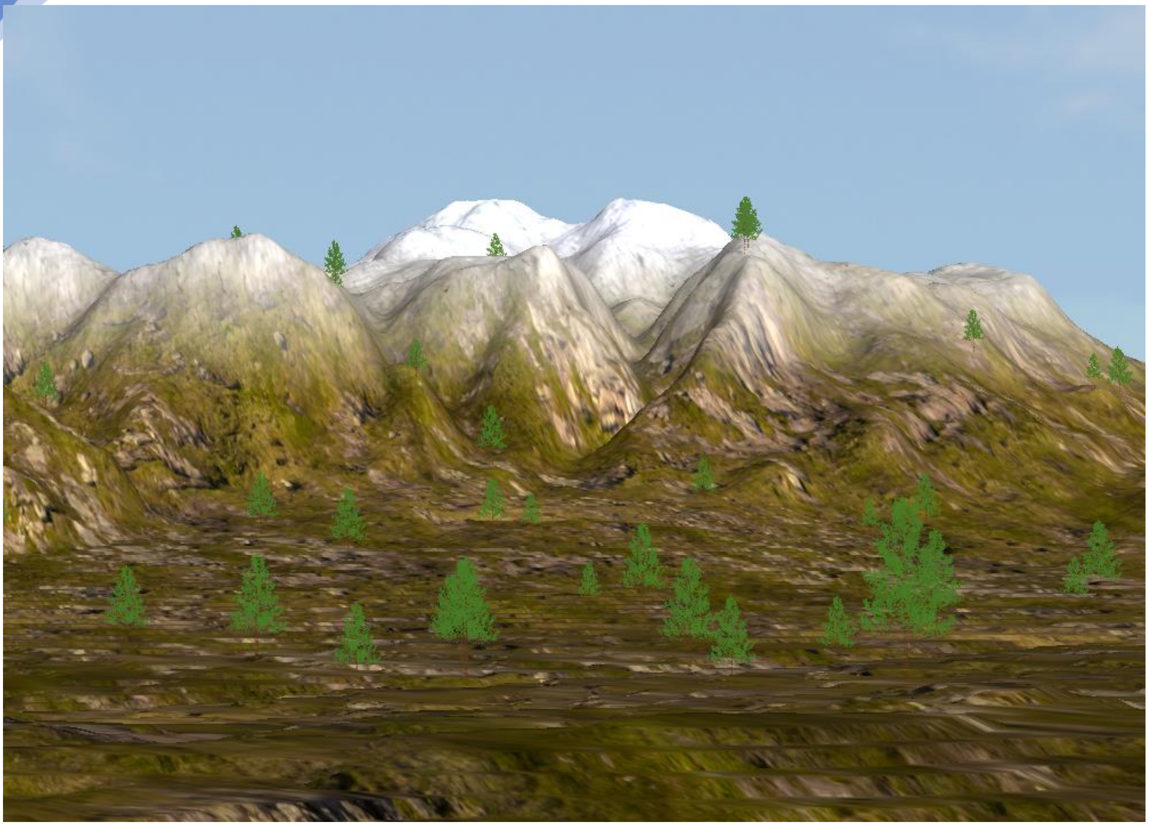
TASK 2 : Vegetation

For the trees, I loaded two separate **.obj files** (MapleTreeStem.obj for the trunk and MapleTreeLeaves.obj for the leaves). I created separate VAOs and VBOs for each. I used a flag so the fragment shader decides the correct color between the trunk and the leaves.

The next step was to create many trees. To avoid loading the tree model many times, I created an array of **800 random model matrices**.

- To generate the random model matrix, I chose random x, z coordinates within the terrain boundaries, and for y, I set the terrain height at that point.
- I also check if y > 5 and y < 80, as I do not want trees at sea level or on mountain peaks.
- Finally, to prevent all trees from looking identical, I apply **random scaling** to the model matrix.

Inside the mainLoop, I render the tree using all the generated model matrices.



TASK 3 : Camera that cannot fly

For the user's movement to be realistic, the camera movement must follow the terrain. To achieve this, I created a getHeight() function which returns the terrain height for given x, z coordinates.

Inside the mainLoop:

- float h = getHeight(camera->position.x, camera->position.z);
- camera->position.y = h + humanHeight;

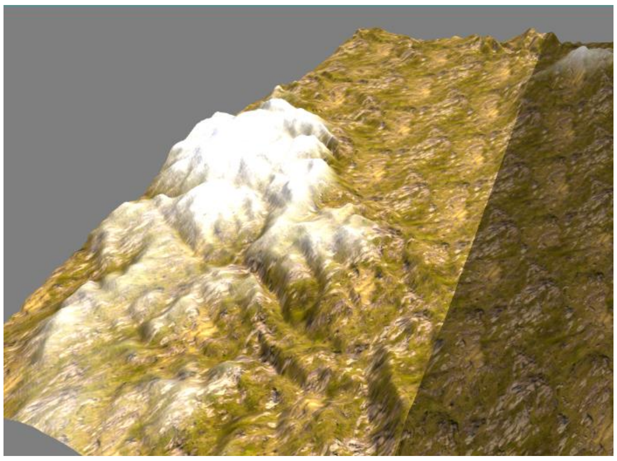
Where humanHeight is a constant simulating human height.

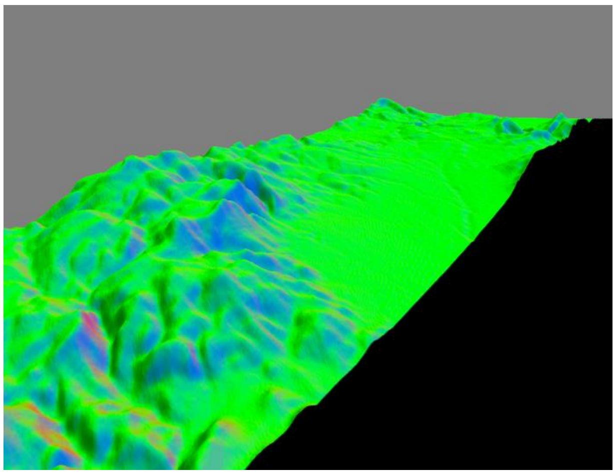
It is worth noting that as the user increased in altitude, the camera height changed discontinuously (jumps), which is logical. To achieve smoother movement, I modified the getHeight function to return the height using **bilinear interpolation**. The function takes x, z coordinates as arguments. After clamping the coordinates within the terrain boundaries, it rounds them to find the neighboring points and their heights. Depending on how close the position is to each point, it takes the corresponding percentage of each height.

TASK 4 : SUN

To simulate the light from the sun, the light source should theoretically be placed very far from the world, so the light strikes all points in the world with the same direction. Therefore, only the light's direction matters. I defined a dirLight structure which contains the components La, Ld, Ls, the lightDirection, and the light's intensity.

For lighting, I followed the **Phong model**, similar to the lab. The only difference is that I pass the light direction vector as a uniform to the vertex shader, so it is not calculated for every fragment.

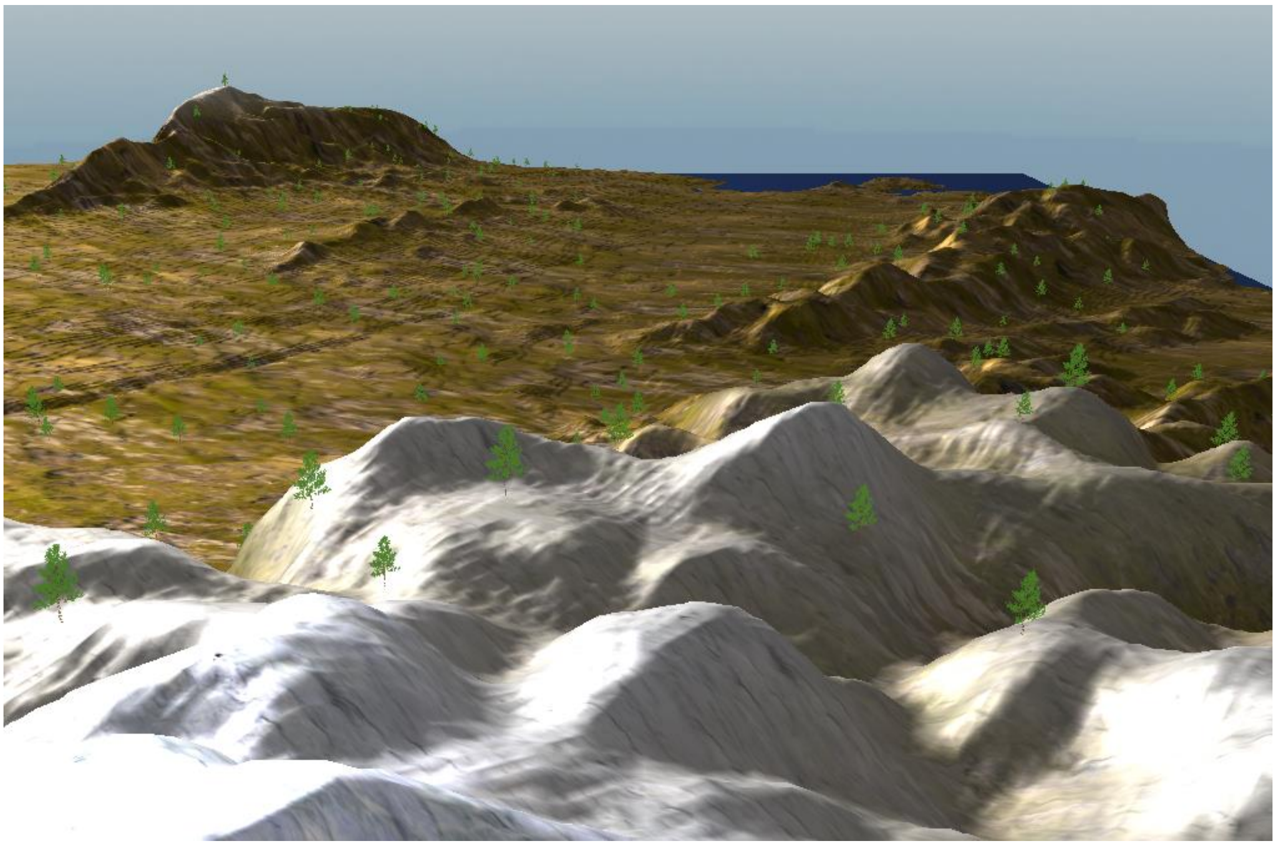




Initially, there was a mistake with the normal assignment, which I realized when I set the color equal to the normal of each vertex. I also made the direction vector change sinusoidally with time to simulate the sun's cycle:

- sun.lightDirection = normalize(vec3(sin(t), cos(t), 0.0f));

The final result is shown below.



TASK 5 : Skybox

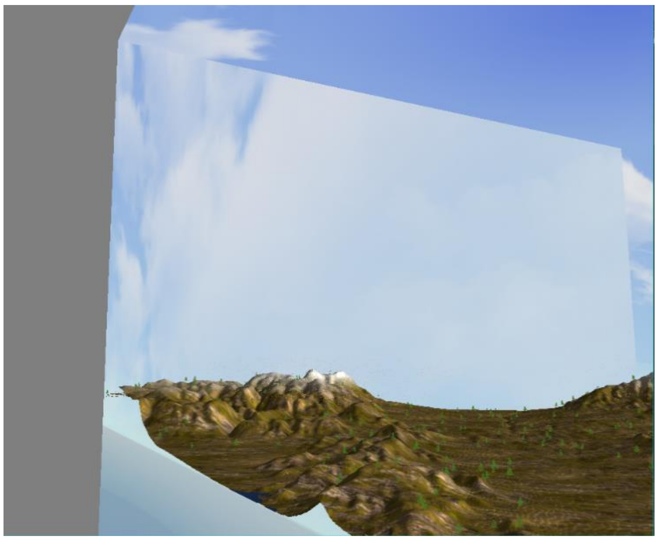
The skybox is essentially a cube onto which a spherical image of the sky is projected.

First, I define the cube's vertices in the skyboxVertices array. I define the vertices for each triangle they are a part of. For simplicity, I define the 2x2x2 cube around the origin and then apply scaling.

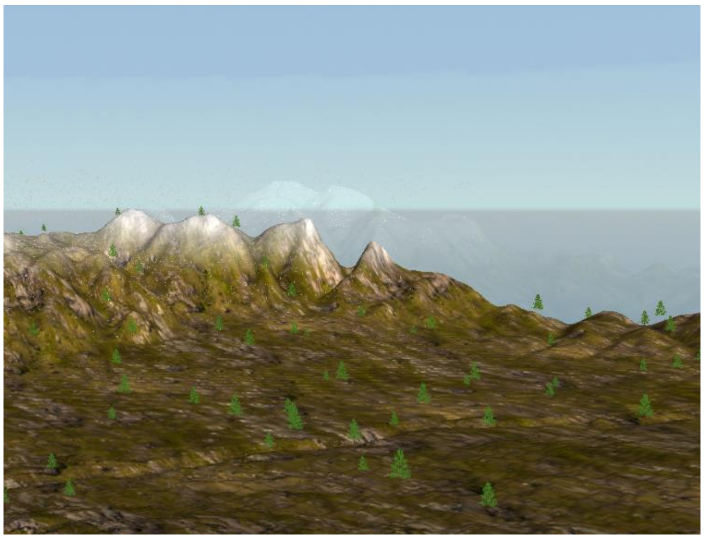
I also need to load the appropriate images. After finding images that make up a skybox, I loaded them using the loadCubemap() function.

- The loadCubemap() function loads a **cubemap texture** using the **SOIL** library.
- First, a texture ID is generated and bound (glGenTextures() and glBindTexture(GL_TEXTURE_CUBE_MAP)).
- Then, for each of the six faces of the cube, the image is loaded from faces (a vector<string> of file paths) via SOIL_load_image().
- The image is stored with glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, ...) , where i corresponds to the six cubemap faces (right, left, top, bottom, front, back).
- The GL_CLAMP_TO_EDGE parameter is used to avoid seams between the faces.
- Finally, the texture ID is returned for use in the program.

For displaying the skybox, I initially used separate shaders. I applied scaling to make it appear outside the world:



However, it was not ideal that the user had the ability to move outside the skybox. Generally, for realism, we should not be able to move relative to the skybox. Therefore, I next removed the translation part from the **View Matrix**. This ensures that the camera's orientation defines the View Matrix, and the camera's position (wherever it is in space) will appear to be at the center of the skybox.



This solution had the drawback that if the user moved to the edges of the terrain, the skybox (being forced to follow the camera) would enter the terrain and cover part of it. Therefore, I significantly increased the **scaling factor** so that this problem does not occur in reasonable positions.

TASK 6 : Reduce the speed of the camera and add sound effect depending on the altitude

In the mainLoop, I have the variable h (the terrain height where the player is located) from Task 3. After defining a maximum limit for the speed, I set the camera speed as follows:

- camera->speed = camspeedmax * (1 - 0.9f *h / 100.0f);

This makes the speed vary linearly from camspeedmax (minimum height) to camspeedmax / 10 (maximum height).

As for the sound : I use the built-in Windows function **PlaySound()** to play audio files.

- PlaySound("wind.wav", NULL, SND_ASYNC | SND_FILENAME | SND_LOOP);
- "wind.wav" is an audio file of wind blowing.
- SND_ASYNC: The sound plays asynchronously, meaning the code continues to execute without waiting for the sound to finish.
- SND_LOOP: The sound plays repeatedly until interrupted.

The volume is set using the height h:

- DWORD newVolume = (h * 0xFFFF) / 100.0f; (h/100 is the percentage of the maximum possible volume, where the maximum value is 0xFFFF)
- waveOutSetVolume(0, (0x1111 << 16) | newVolume);

In this way, the wind the user hears becomes progressively stronger as they ascend to higher altitudes.

TASK 7 : View bobbing

Inside the mainLoop, I create a **bobbing amount** that is added to the camera's position:

- vec3 bobbing = vec3(0.05f * sin(50 * t), 0.005f * sin(55 * t), 0.05f * sin(45 * t));
- camera->position += bobbing * (h / 100);

I use different frequencies because I do not want synchronous oscillation; I want it to appear random. Also, since the bobbing is caused by the wind (which is stronger at high altitudes), I multiply the bobbing amount by the height normalized to one.

BONUS 1 : Sway the vegetation

To implement the swaying, I worked in the **vegetation.vertexshader**.

```
float attitude = 0.01*Mo[3][1]; // *0.01 because max height=100
float factor = vertexPosition_modelspace.y; // because i dont want to sway the lowest part of
the plant
factor *= 0.05*factor; // because i want the tree to bend from swaying
vec3 swaying = vec3(0.08f * sin(3 * time), 0.0f, 0.02f * sin(5 * time));
vec3 SvertexPosition_modelspace = vertexPosition_modelspace + attitude*factor*swaying;
```

This swaying is intended to move the tree in the x, z plane, which is why its y-component is zero. Through experimentation, I found specific amplitudes and frequencies (which are different to prevent synchronous oscillation) to make the movement realistic.

With this method, the whole tree sways. However, I want the lowest part to remain stationary and the swaying amplitude to gradually increase moving upwards in the tree. The initial thought was to multiply the swaying by the local height of each vertex in model space, but this only makes the tree tilt without bending. To make it more realistic, I wanted the tree to bend, so I used a **quadratic dependence** on height.

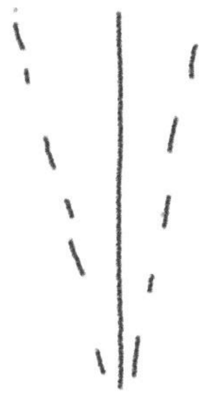


Figure 1 linear dependence on height.

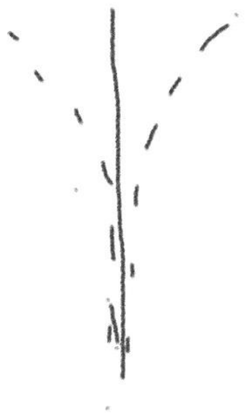
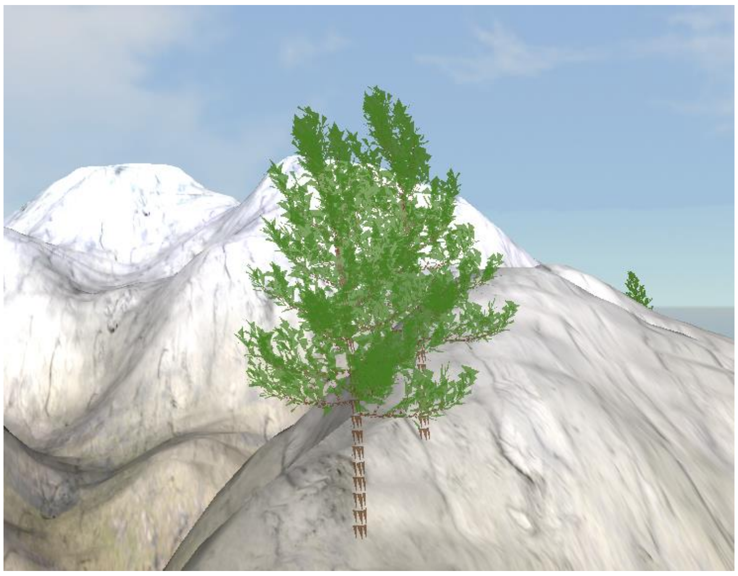


Figure 2 quadratic dependence on height.

Finally, I wanted the swaying to depend on the altitude where the tree is located, which I extracted from the model matrix, normalized it to one, and used it as a factor for the swaying. The final result is shown below:



<u>BONUS 2 : Blur in the direction of the wind</u>

To create a blur effect based on whether the camera is looking into the wind, I used **post-processing** with a framebuffer texture, into which I load the rendering result.
First, I create the framebuffer texture and a quad consisting of two triangles that covers the entire screen. The framebuffer texture will be projected onto this quad. I use separate shaders for rendering the blurred image.
In the mainLoop, I first bind the blur framebuffer object and then execute the scene rendering. Afterward, I use the blur_shaderProgram and bind quadVAO and textureColorBuffer.

The intensity of the blur (blur factor) is maximum when the camera faces the wind direction and decreases when turning away:

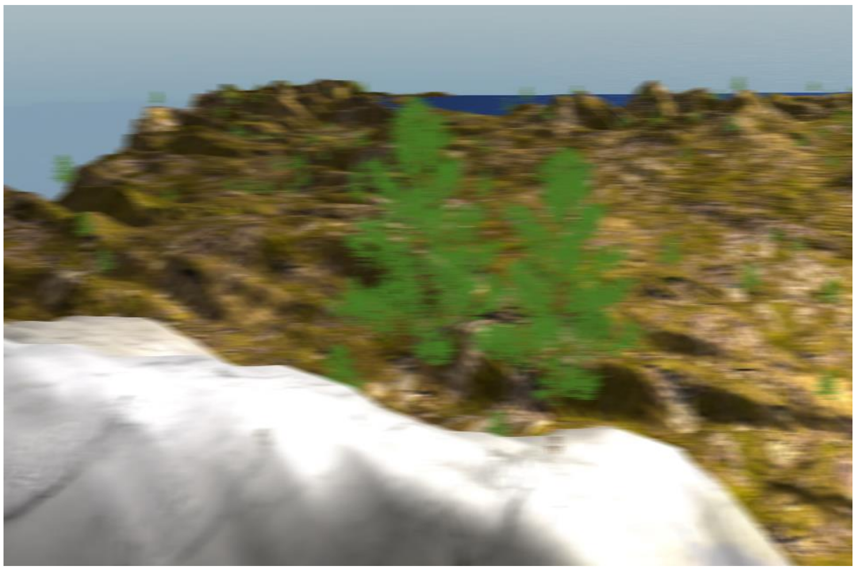
```
float blurfactor = dot(normalize(-winddirection), normalize(camera->direction)); // i want the max blur=1 when the camera is facing the wind
blurfactor = std::max(0.0f,blurfactor); // because i dont want blur when i turn my back on the wind
```

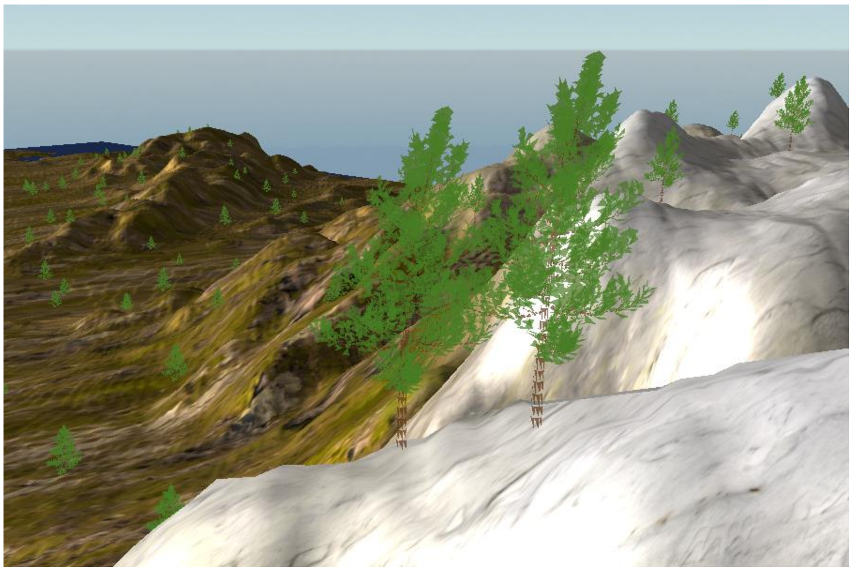
I also wanted the blur to be affected by the strength of the wind (how high the user is), so I pass the normalized height as a uniform to the shader in addition to the blurfactor.

The color of each fragment is then calculated in the fragment shader by taking the **average of neighboring fragments**. The number of neighboring fragments is predetermined. However, how spread out or concentrated they are around the current fragment is determined by the factor:

```
blurOffset = offset * blurFactor * windSpeed * 0.0003; // the last factor found expirementally
```

Where offset is a vec2 that defines the direction on the texture where the points used for the average will be spread out. Specifically, I set it to [1, 0] so that the contributing points are on the same horizontal line.





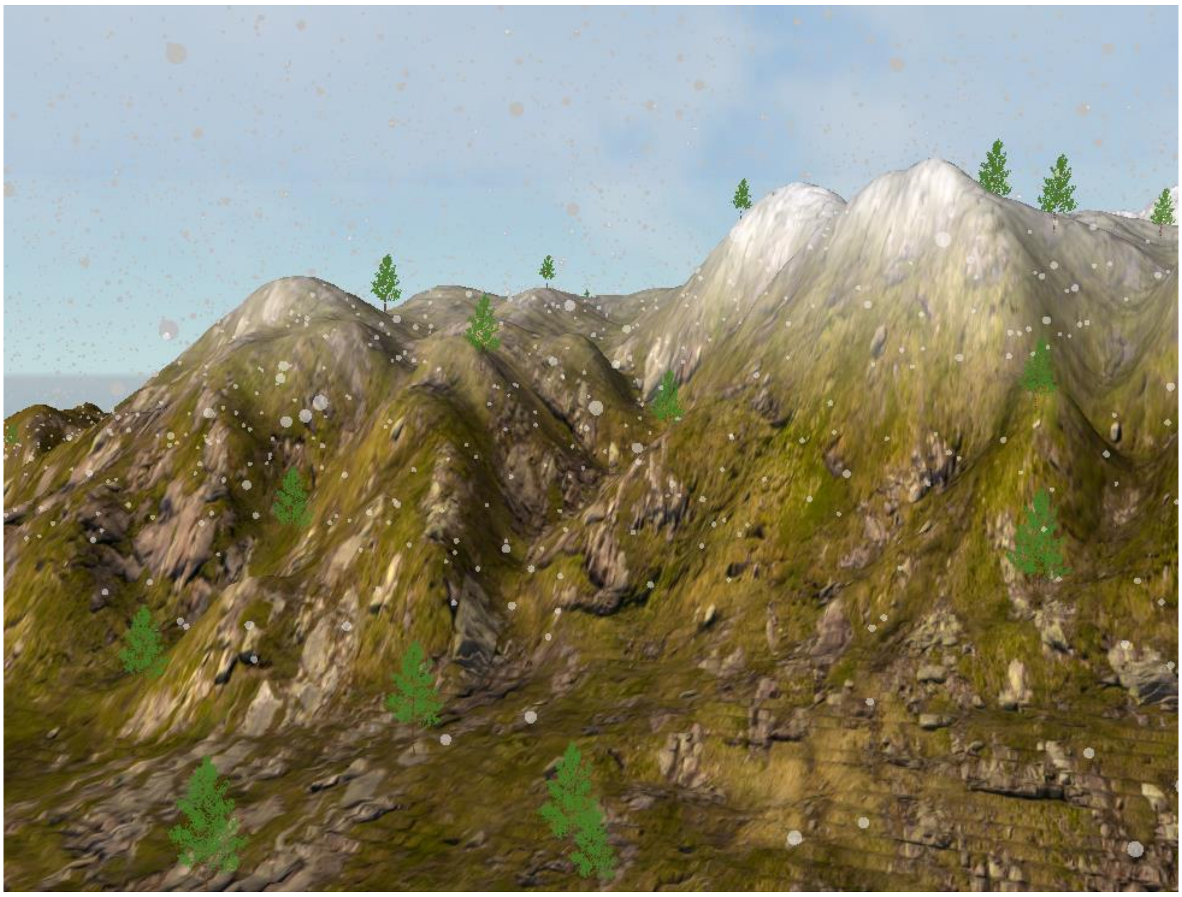
The images on the left show the two same trees when looking against the wind (top) and when looking in the same direction as the wind (bottom).

TASK 9 : Particles

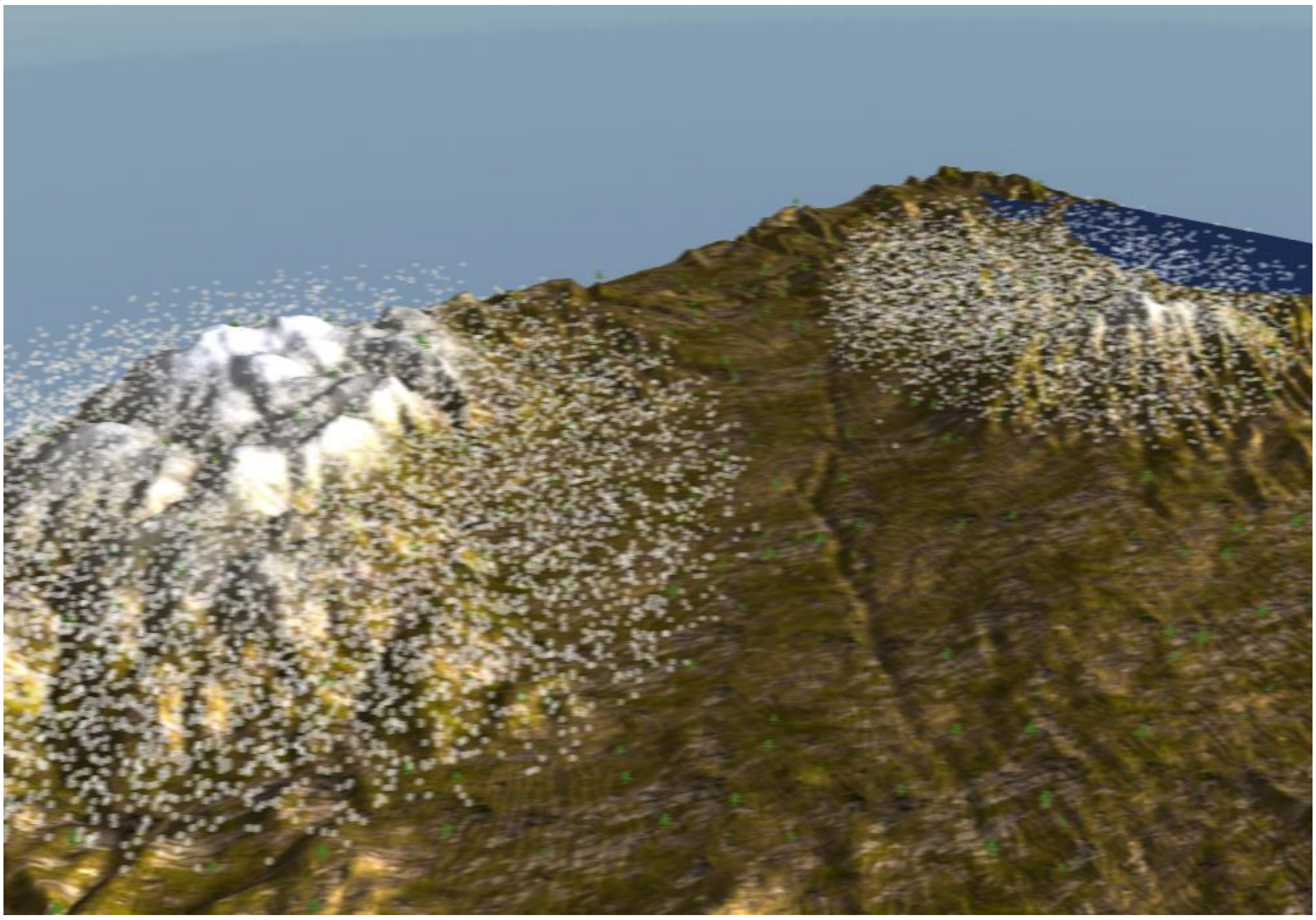
Based on the extra lab regarding particles, after including the relevant files in my project, I created the **DustEmitter** class, which inherits from IntParticleEmitter. Instead of dust, due to the mountainous terrain, I decided to use **snow**. The goal was to create two clouds of snow particles in the regions of the two highest mountains. I also want the particle movement to depend on the wind direction.

- I modified the constructor to accept the radius of the cloud and its center point (center_position) as additional arguments.
- In createNewParticle(), I set the initial position according to the predefined region in the x, z plane and set y to a random height in the range [center_position.Y, 100]. The particle's velocity and size are also initialized.
- In updateParticles(), I increase the velocity by a **constant multiplied by the wind direction**. To make it look realistic, I add a small component in a random direction. I also check if the particle has moved out of bounds, and if so, it is regenerated.

In the main program, I create two snow dust_emitters using a sphere as the object.



The two clouds are shown from above (with increased particle size).

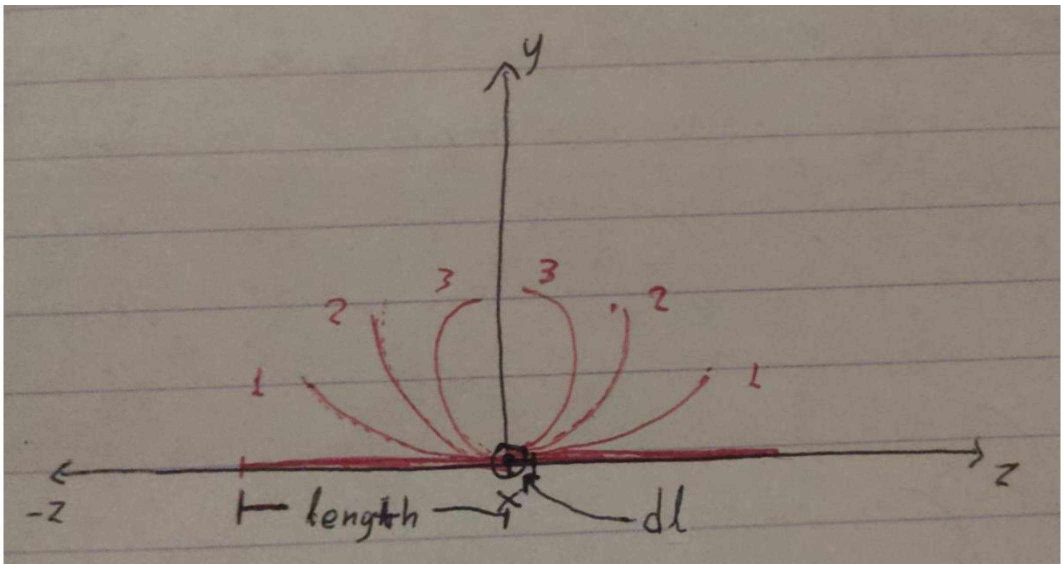


TASK 10 : Rotational world distortion !!!

I apply this within the **vertex shaders** of the terrain and the trees.

The main idea is that the world will curve/curl/close with respect to a fixed axis.

Assuming this axis is x, viewing the y, z plane would look like the illustration below.



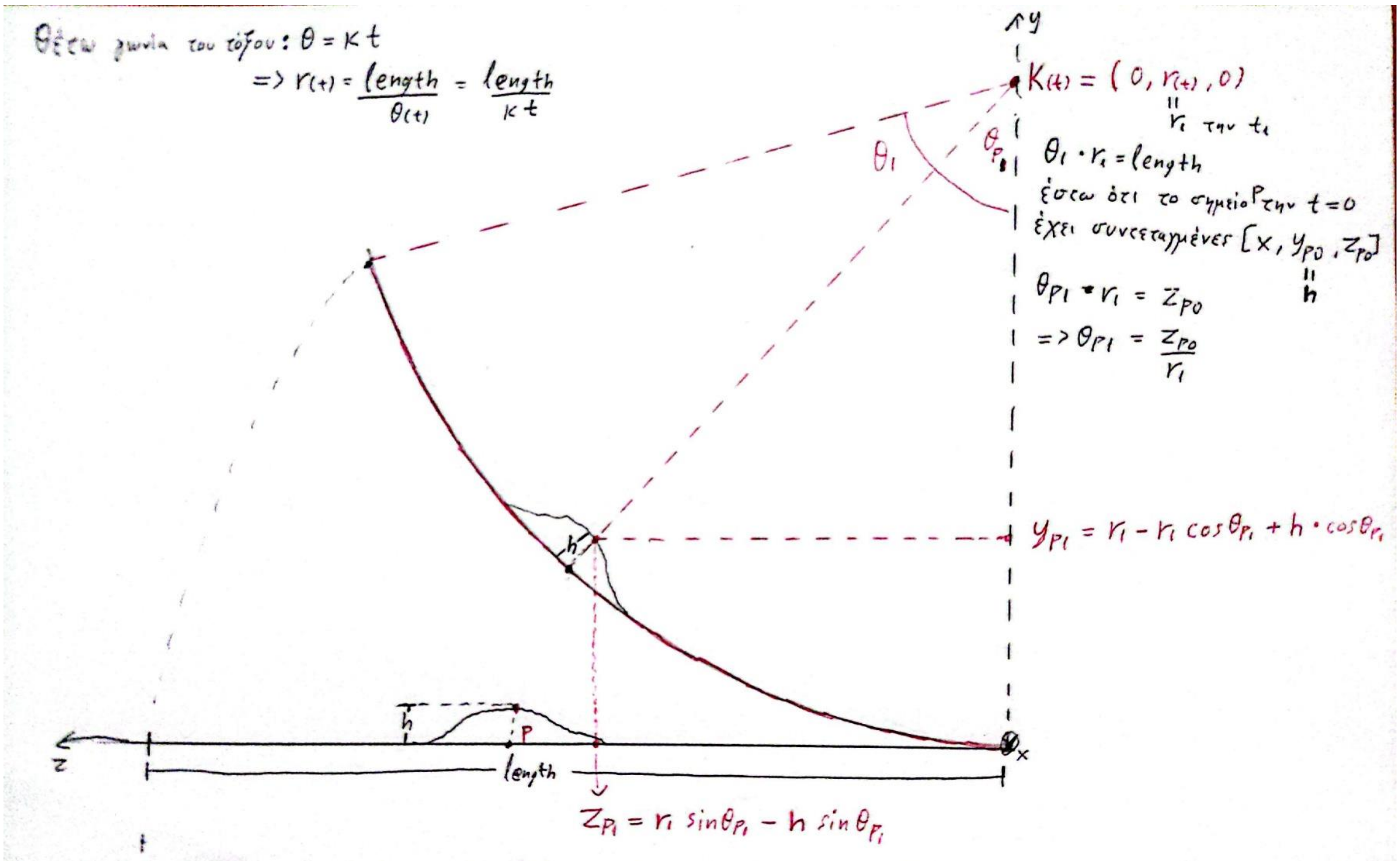
Where 1, 2, 3 show the curvature of a slice at increasing time instants.

I want the length to remain constant over time and the curvature to be spatially constant but temporally increasing.

Therefore, I want the slice at any given time to be a circular arc of continuously decreasing radius.

I also want the infinitesimal dl (as shown in the above picture) to remain on the z axis (otherwise, a fold would appear on the x axis), so the circle center must be on the y axis.

To solve the problem, I need to calculate the new coordinates of a point given its coordinates for t=0 and the time elapsed. The x coordinate is not affected by the rotation, so I focus on the y, z plane.


Θέτω γωνία του τόξου: $\theta = \kappa t$
$\Rightarrow r(t) = \frac{length}{\theta(t)} = \frac{length}{\kappa t}$
y
$K(t) = (0, r(t), 0)$
r_1 την t_1
θ_1
θ_{P_1}
$\theta_1 \cdot r_1 = length$
έστω ότι το σημείο P την $t=0$ έχει συνεταγμένες $[x, y_{P0}, z_{P0}]$
h
$\theta_{P1} \cdot r_1 = z_{P0}$
$\Rightarrow \theta_{P1} = \frac{z_{P0}}{r_1}$
$y_{P_1} = r_1 - r_1 \cos\theta_{P_1} + h \cdot \cos\theta_{P_1}$
h
h
P
z
length
x
$z_{P_1} = r_1 \sin\theta_{P_1} - h \sin\theta_{P_1}$

To implement the rotation in the code, I define the pos_after_Rot() function in the vertex shader, which implements the above procedure, and call it if the uniform indicating the rotational distortion is enabled. I also want the rotation to depend on x, with the rotation evolving faster at points with smaller x.

```
float xfactor = (pos.x+w)/1222.0f ;
```

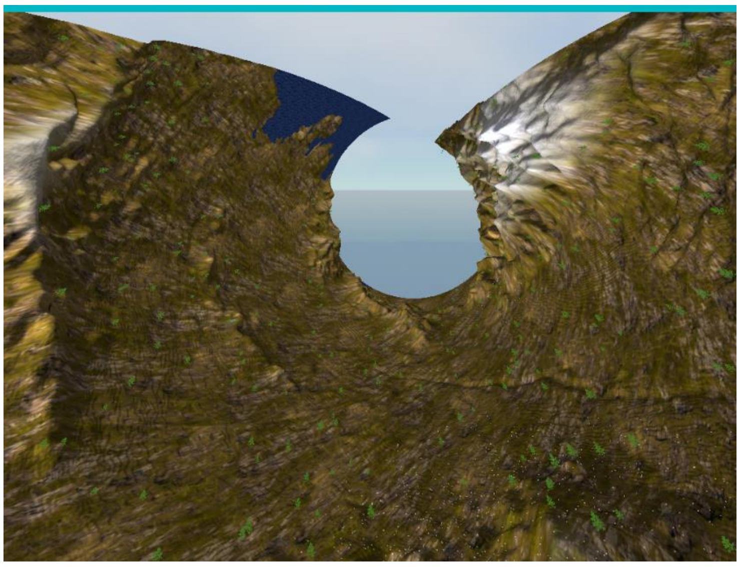
```
theta = theta - 0.8*xfactor * theta;
```

This way, points with a larger x have a slower but non-zero rotation.

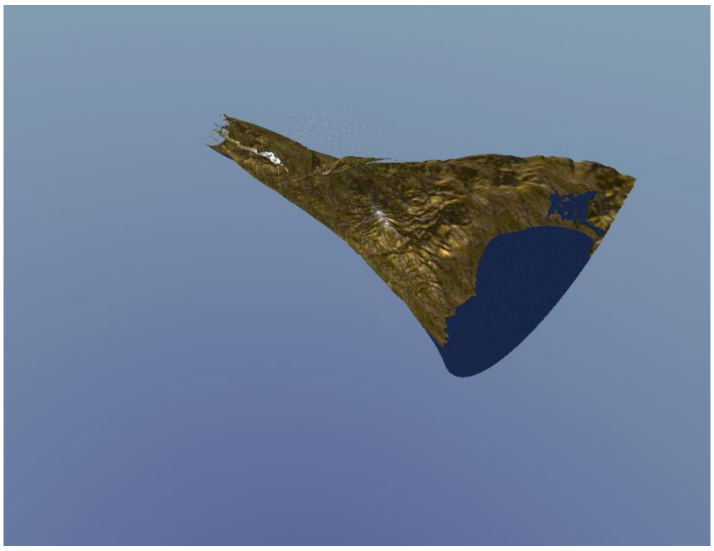
In the mainLoop, I define that rotation will occur if the 'z' key is pressed.

Finally, I want the rotation to start from the initial state whenever the user presses 'z'. Thus, I defined a new time variable rottime that is used for the rotation and which the user can stop or start by pressing 'x'.

Some results are shown below!



MOUNTAIN HIKING



TASK 11 : Voxelizing and quantizing the scene

I apply this within the **vertex and fragment shaders** of the terrain and the vegetation.

Since I had two ideas for how the Voxelizing/quantizing would be done, I defined the uniform variable voxelizing to determine the Voxelizing operation.

Voxelizing	Operation
0	The world remains as is.
1	Voxelizing occurs: Vertices are rounded to certain quantized values, so the world appears in small cubes.
2	Same as the previous, but the cubes are separated from each other.
3	Here I don't round the world into cubes. I "break" (separate) it into square pieces, which are moved away from each other by a defined distance gap.

Implementation Details;

Voxelizing = 1)

To round the vertices to specific points on a grid, I define a gridSize, which is the distance between neighboring points on the grid and determines the size of the cubes the world consists of.

```
newPos.x = floor(aPos.x / gridSize) * gridSize;
```

With the above method, if aPos.x is the initial x coordinate of the point, the new x will be the initial x rounded down to the nearest multiple of gridSize. By applying this to x, y, z, all points end up on a 3D grid. However, cubes appear in the world (not isolated points) due to the interpolation performed for the fragments.

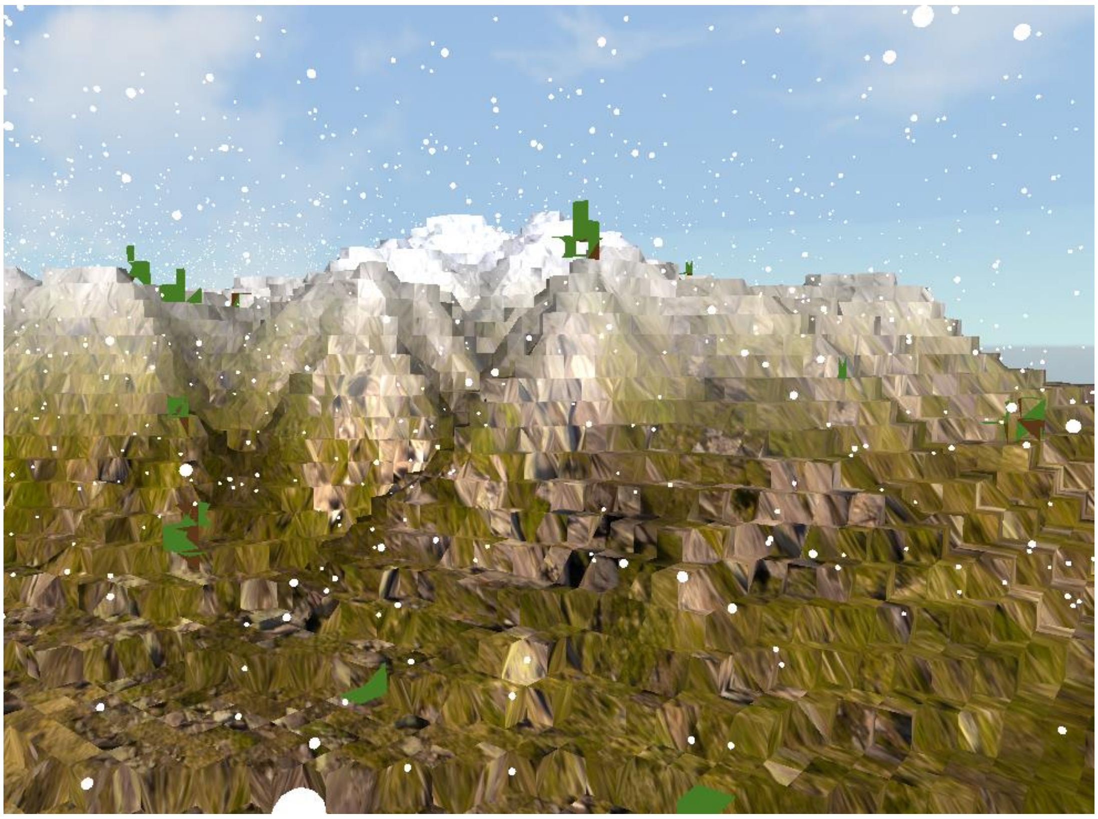
Voxelizing = 2)

Continuing from the previous case, I now want to introduce a gap between the voxels. To create gaps, I need to discard some fragments, so I apply the following in the fragment shader:

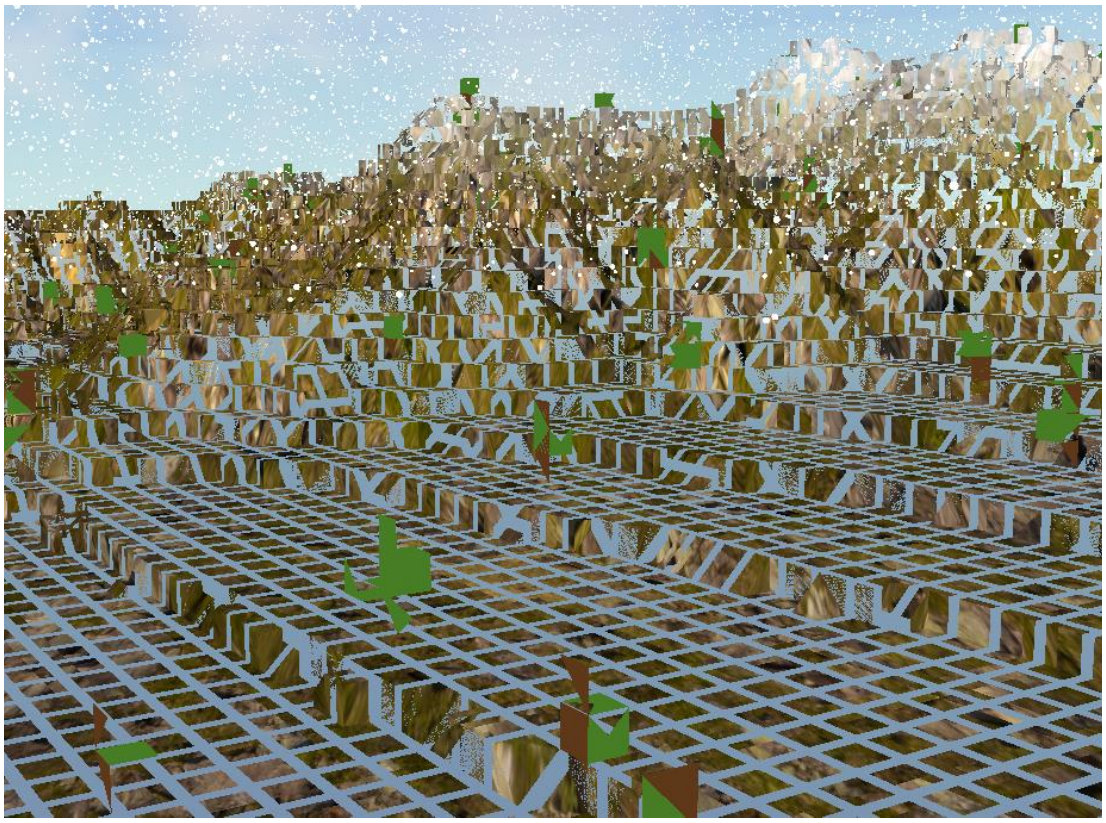
```
vec3 localPos = mod(posModelspace, gridSize + gap);

// If the fragment is inside the gap, discard it
if (localPos.x > gridSize || localPos.z > gridSize || localPos.z > gridSize) {
    discard;  // This makes the gap fully transparent
}
```

The mod() function returns the local position of the point within the local voxel (size gridSize + gap) it belongs to. If the local position is outside the local voxel of size gridSize, I discard the fragment.

Voxelizing=1



Voxelizing=2



Voxelizing = 3)

In this method, I group the vertices into cells. I want these cells to be displaced from their original position so that the distance between them is constant. Therefore, each cell must move away from its original position by gap + gap * (number of cells intervening between the origin and the current cell). So:

```
vec3 voxelIndex = floor(aPos / gridSize); // calculates the grid index
newPos = aPos + voxelIndex * gap;
```

The method for discarding the fragments between the cells is the same as in the Voxelizing=2 case.

Voxelizing=3