Bachelorarbeit

Outdoor Editor implemented with Vienna Vulkan Engine

Verfasser

Stefan Eger

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2021

Studienkennzahl lt. Studienblatt:	UA 033 521
Fachrichtung:	Informatik - Medieninformatik
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

Abstract

This bachelor's thesis extends the Vienna Vulkan Engine by providing it with a graphical user interface for creating scenes, without the currently required knowledge of importing models and creating a terrain via code. The Outdoor Editor is provided with a limited amount of nature models and enables the user to design simple nature scenes. The process of importing models has been streamlined and simplified in order to make the action of placing a model easier for future programmers and user alike. Furthermore the terrain is interactive, because it is based on Voxels and uses the Marching Cubes algorithm to generate a mesh. This editor also serves as a foundation for future projects in the VVE.

1 Introduction

Vulkan is one of many APIs to create a game, but the problem with Vulkan is that it is very difficult to understand, debug and requires a lot of effort to even render a single triangle, but it also provides a significant performance increase. The Vienna Vulkan Engine [6] simplifies the use of Vulkan enormously, but still requires code and knowledge to create a scene. The aim of the Outdoor Editor is to simplify this process even further and give the user instant visual feedback while creating a nature scenery. The question is, what does the Outdoor Editor need to implement exactly, in order to create an engaging scenery.

1.1 Implementation Goals

1.1.1 Voxel Terrain

One goal is to create a terrain, which is built by the Marching Cubes algorithm. This enables the user to not just create hills and valleys, but also caves and overhangs.

1.1.2 Interactive Terrain

Modifying a terrain sounds simple, but as any 3D artist and programmer knows, it requires a lot of mathematical knowledge to get decent results. For this reason the Outdoor Editor has to provide some user friendly way to modify a scene, this includes knowing where something is placed, what area is affected and which actions are available.

1.1.3 Models

Another goal is the ability to add different models to a scene. These nature models are provided by the Outdoor Engine and will be created in Blender. This enables the user to create entire nature scenes without previous knowledge of Vulkan or the Vienna Vulkan Engine and is able to focus on designing nice landscapes.

1.1.4 Reusability

With all these components in mind the Outdoor Editor should be written in a polymorphic way, that allows its integration into later projects, which involve other components and environments than trees and grass. Another point that falls into this category is the ability to save and load an already created scene.

2 Related Work

An Outdoor Editor can be seen as a simplification of a terrain editor implemented in game engines such as Unity or the Unreal Engine. These engines have been developed by multiple teams over a long time and the Outdoor Editor can not compare in quality to those engines, but they can be used to illustrate different concepts which the Outdoor editor is based on and what the Outdoor Editor does differently.

One of the similarities between these game engine terrain editors and the Outdoor Editor is that models and textures can be placed via a simple click. This is enabled by brushes which the Outdoor Editor also provides. These brushes help to edit the terrain, but the brushes in game engines like Unity are much more sophisticated than the one the Outdoor Editor provides. Which leads to what the Outdoor Editor does differently than the terrain editors of other game engines.

The main difference is the terrain. Most of these scene editors use height textures and/or simple planes with a fixed amount of vertices. These height maps are grayscale images, which store the height of a landscape and pass this value on to a vertice plane. Another approach is to just store the Plane Mesh itself and modify it with the given tools, but the problem with these approaches is the inability to create caves. The standard tools of these programs do not support this. The Outdoor Editor uses Voxels, to create a terrain, which allows caves, overhangs and more creativity. The Voxel approach allows more freedom but also adds another dimension of data, since a texture only needs 2 dimension, the Voxel terrain stores 3 dimensional data. Normally this approach is only seen in games like Minecraft, but not in scene editors within a game engine, although there are many implementations of Voxel terrains in the aforementioned mentioned game engines. However there are also different ways to render Voxel data which allow even more freedom in displaying and modifying the scene appearance. The Outdoor editor focuses on the Marching Cube algorithm, but this algorithm could be replaced by others. Another difference is the fact that those engines are not built upon Vulkan to create those terrains.

3 Architecture and Design Decisions

In this chapter each package of the class diagram and its design decisions will be explained A.2.

3.1 Voxel Data and Voxel Presentation

As mentioned before the terrain is built upon Voxel data, but before explaining how it was implemented, it is necessary to understand what exactly a Voxel is. A Voxel is a 3 dimensional pixel, which holds a certain Isovalue. This Isovalue represents a certain density at a given 3D coordinate. In order to create a terrain, a grid of 3D pixels is needed.

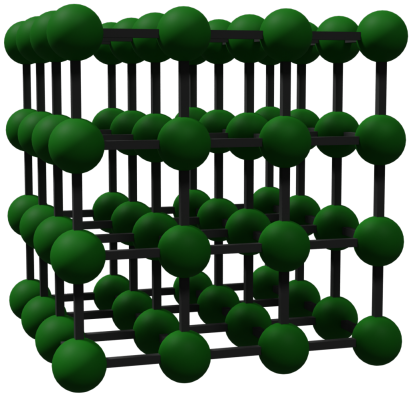

Figure 1: Voxel grid (green sphere represents a Voxel)

The aforementioned information is used to generate a mesh around these points of data. The algorithm chosen for this task is called Marching Cubes [9]. For the task the class diagram A.2 implemented the VoxelData and TerrainGenerator package. The Voxel data and mesh generation was separated, to allow different presentation methods of the Voxel data. Marching Cubes implements a strategy pattern with the TerrainGenerator class. This way other algorithms to create a mesh around Voxels is still possible. Since rendering an almost infinite 3D array is difficult to maintain, change and render, it is wrapped up in so called chunks. These VoxelChunks hold multiple Voxels at a time, which makes it easier to create a terrain. It also opens up the possibility for multi threaded terrain generation, endless worlds and many other features in the future. These chunks are also managed in the VoxelManager which makes access easier by addressing individual chunks with just a 3D coordinate in the World Coordinate System. The same approach can be seen in the Voxel presentation with the classes TerrainManager and TerrainMeshChunk. The TerrainMeshChunk is a simple data structure which holds the actual mesh

of a chunk on a cube by cube basis with the MeshCell class, but it does not actually create the vertices and indices of a chunk. This Task is handled by the MarchingCubes class, but the TerrainMeshChunk is responsible for finally displaying the mesh via the Vienna Vulkan Engine.

3.2 Models

Creating models by code always has the danger of a shotgun surgery, if the Vienna Vulkan Engine changes its functions. To avoid searching for each class, which imports a model via a file path, it is necessary to collect each component of this process in one class. This class is called NatureEntityDatabase, which stores all the information needed to import a model. This information consists of the file-path and the filename and also includes scaling for quick readjustments and aiFlags to import the model correctly. All of the data just mentioned is wrapped up in NatureEntity_t class. After knowing where to import models from, the question arises how a certain model should be imported. All different kind of models are imported via a the NatureEntity and NatureEntity_t class. These classes make use of the Type Object Pattern, which reduces the ability to add new variables to the model importing process. In this case this does not matter, because there won't be a variation of variables in the import process anyway and this simplifies the process of importing all kinds of models with a single class dramatically, because the NatureEntity_t class holds every information needed [10]. But there is more than one way to import a model e.g. leaves and billboards have to be imported differently, because they need different render passes and different approaches to create a model with the VVE. These variations of the NatureEntityClass still use NatureEntity_t, but the NatureEntityLeafs class, which has the ability to cut off leaf objects and billboards, only accepts a path to a picture which is used as a texture on a billboard plane. Another variation of the NatureEntity class is named NatureEntityModel, which implements another pattern called Composite Pattern, which allows hierarchical tree structures. This permits the combination of different importing techniques into a single NatureEntity, which comes in handy for models like a tree, which usually consists of a tree trunk and multiple leave planes. The ModelManager class is tying it all together and acts as a facade that showcases available models and can be seen as an interface between the EntityDatabase and NatureEntity classes.

3.3 Brushes

Brushes can be seen as a tool to extract an affected area from the current mesh. They take into account the position where something has been hit and around this position a sphere is formed with a given radius. There are multiple mathematical functions which can be useful to determine how much a position, within a certain radius and depending on its distance to the center, is affected. This is useful to edit the terrain, because Voxels have a certain Isovalue, which determines the shape of the terrain. Now editing a terrain can be seen as adding and

subtracting spheres from a terrain. This is implemented via a Strategy Pattern, because there are many mathematical functions to determine an affected area. With the help of the Strategy Pattern, the list of brushes can be extended if a future programmer wishes to do so.

3.4 The Editor

The Editor Package contains the OutdoorEditor class, which is the centerpiece of the editor and acts as a facade to VVE classes, which applies to the OutdoorEditorEngine and the Event Listeners. Those classes have the ability to change the Outdoor Editor. To enable Event Listeners and the OutdoorEditorEngine to modify a single Outdoor Editor it was necessary to save the object as global value in OutdoorEditorInfo which acts as an interface between the VVE classes and the Outdoor Editor. The Outdoor Editor takes input from those classes, but how this input is processed depends on which state (oeEditingModes) the OutdoorEditor currently is. The Outdoor Editor also provides access to all the Sub-Managers that it uses to avoid unnecessary bloating of the Outdoor Editor class, with functions that only propagate a command to the correct manager. The Outdoor Editor itself handles with its functions mainly actions which use more than one manager class. This decision ensures that the outside (VVE) classes do not have to care about synchronising these manager classes.

3.5 Event Listeners

The Event Listeners take user input and translate this input for the Outdoor editor. The EventListenerUser class takes keyboard, mouse buttons and mouse movement as input. The keyboard is used for controlling the camera and mouse clicks for interacting with the terrain. On the other hand the GUI Event Listener provides options to change the state of the OutdoorEditor and also provides options about models, materials and brushes. To communicate with the Outdoor Editor both use the OutdoorEditorInfo class.

4 Implementation

4.1 General

This chapter deals with how to use the Outdoor Editor and then goes into depth how important parts of the editor work. The editor was implemented on top of the Vienna Vulkan Engine [6] and uses Visual Studio 2019 with cpp20.

4.2 VoxelData

Lets begin with Voxel data, which creates a 3D Grid of Isovalues at its junctions (1). These Voxels are stored in an array in the VoxelChunkData class. The problem is these Voxels can only be accessed via local chunk coordinates. This becomes problematic at a certain number of chunks, because the world coordinate has to be translated into the local chunk space. (B.1) Every time a getVoxel or setVoxel is called it uses world2ChunkCoordinates and world2local to translate world coordinates to local chunk coordinates. This is why it is critical to use the manager to access chunks instead of using a chunk directly, otherwise editing a chunk becomes prone to errors. However editing the data of a chunk, does not result in constructing a mesh, this is the task of the marching cubes algorithm.

4.3 Marching Cubes

The Marching Cubes algorithm is based on the idea of a cube in a 3D Voxel grid, which marches through the grid cube by cube until the chunk is completed [9].

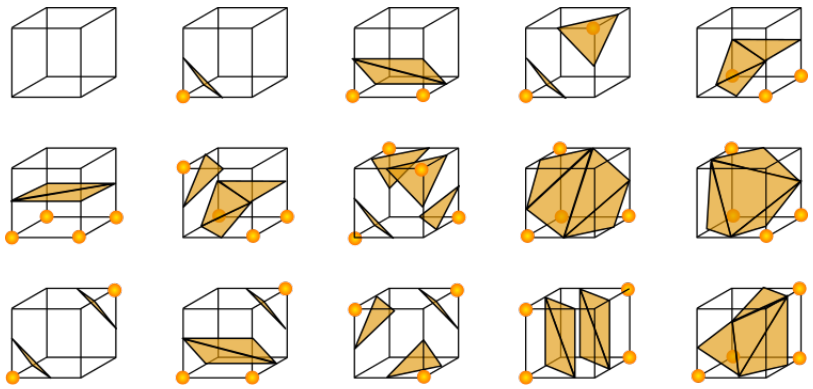

Figure 2: Marching Cubes cases [4]

The Marching Cubes algorithm has 256 different cases, but can be broken down to 15 basic cases as seen above 2. While a single abstract cube is marching through the grid it generates vertices according to these cases, but it does not actually render mesh yet

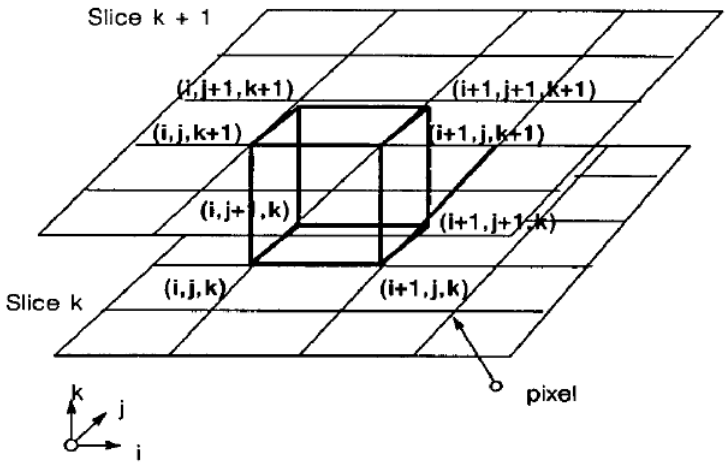

Figure 3: Marching Cubes walking through grid (x = i, y = k, z = j) [9]

A big part of the code (B.2) was copy pasted from this website [5] and has been adjusted to use the VoxelManager and TerrainMeshChunks in order to actually create the mesh with the VVE. There are only 2 methods, which allows to generate a mesh for a single cube or an entire chunk. The cubeIndex determines which of the 15 cases applies and then interpolates between the selected and unselected vertices in order to get final vertices. The vertex normals are a bit trickier, but can be calculated by looking at each neighbouring vertice with the following formula [9].

$$Gx(x,y,z) = \frac{D(x-1,y,z) - D(x+1,y,z)}{\Delta x} \tag{1}$$

$$Gy(x,y,z) = \frac{D(x,y-1,z) - D(x,y+1,z)}{\Delta y} \tag{2}$$

$$Gz(x,y,z) = \frac{D(x,y,z-1) - D(x,y,z+1)}{\Delta z} \tag{3}$$

G is the gradient and D is the density of a Voxel (isovalue). Δ x, y, z can be neglected, because the Outdoor Editor only uses cubes with the length of 1. The formula has been adjusted to calculate correct vertice normals. To better understand what it does it is helpful to think about an edge case, where an axis (lets take y axis) is observed with an Isovalue of 1 below and and Isovalue of 0 above. Using the formula it calculates the opposite direction upward for Gy away from the surface. Taking all direct neighbours into account then the vector represents the vertex normal.

```
MarchingCubes::GridCube MarchingCubes::nextCube(const VoxelCoordinates& chunk,
    const VoxelCoordinates& offset) const
{
        GridCube cube;
        for (int i = 0; i < 8; i++) {
                VoxelCoordinates cubeVerticesShifted =
                    VoxelManager::local2World(cube.vertices[i] + offset, chunk);
                //Lookup Voxel
                cube.voxelVal[i] = voxelManager->getVoxel(cubeVerticesShifted);

                //Calculate Surface Normal of with gradient
                    if there is no neighbor then it is empty
                cube.voxelNormals[i].x = cubeGradientDir(cubeVerticesShifted,
                    VoxelCoordinates(1, 0, 0));
                cube.voxelNormals[i].y = cubeGradientDir(cubeVerticesShifted,
                    VoxelCoordinates(0, 1, 0));
                cube.voxelNormals[i].z = cubeGradientDir(cubeVerticesShifted,
                    VoxelCoordinates(0, 0, 1));

                cube.voxelNormals[i] = (cube.voxelNormals[i] != glm::vec3(0,0,0)) ?
                    glm::normalize(cube.voxelNormals[i]) :
                    cube.voxelNormals[i];
        }
        return cube;
}

//Direction only (1,0,0), (0,1,0) or (0,0,1)
float MarchingCubes::cubeGradientDir(const VoxelCoordinates& currentVertice,
    const VoxelCoordinates& direction) const
{
        float d1 = voxelManager->getVoxel(currentVertice - direction).density;
        float d2 = voxelManager->getVoxel(currentVertice + direction).density;
        return (d1-d2);
}
```

4.4 Terrain Mesh Chunks

TerrainMeshChunks is a simple data structure which stores vertices and indices on a cube by cube basis. The reason, why such a data structure is needed, is Vulkan, which only returns handles to a vertices, indices buffer, but the actual vertice and indices data is unavailable at this point. This data is necessary for interacting with the terrain, which is explained later 4.5. As this code snippet demonstrates B.4, the chunk is not saved as a coherent single mesh, but is instead saved as a chunk consisting of many cube objects. This is a design decision since every object needs a material with a shader to render and it would otherwise limit the variety of applicable textures severely. Four to be

exact which is the upper limit of the VVE. Another problem with a voxel terrain is that it can be looked at in every possible direction and the Marching Cubes algorithm does not provide UV-coordinates for a vertice. To the rescue comes a technique called tri-planar texturing, which calculates a UV-coordinate from the position of a vertice. Furthermore it also solves the problem of distorting a texture in a 3D environment by applying a texture three times from each axis and then blends them together depending on the vertice normals. All of this is processed in the fragment shader [12].

```
void main() {
...
//parameters
uint resIdx     = iparam.x % RESOURCEARRAYLENGTH;
vec3 normalW    = fragNormalW;  //to be consistent with DN
...
/*
https://gamedevelopment.tutsplus.com/articles/use-tri-plan
ar-texture-mapping-for-better-terrain--gamedev-13821 (08.11.21)
*/
//Trilinear Blending
vec3 blending = abs(normalW);
//Just care about if something is on the axis not if it is minus or plus
blending = normalize(max(blending, 0.00001f));
//Force blending to be between 0 and 1
float b = (blending.x + blending.y + blending.z);
blending = blending / vec3(b,b,b);
float texScale = 0.5f;
vec3 fragColor;
//Sampling the Texture 3 times in each direction with the current position of the vert
vec3 Xdir = texture(texSamplerArray[resIdx], fragPosW.yz * texScale).rgb;
vec3 Ydir = texture(texSamplerArray[resIdx], fragPosW.xz * texScale).rgb;
vec3 Zdir = texture(texSamplerArray[resIdx], fragPosW.xy * texScale).rgb;
fragColor = Xdir * blending.x + Ydir * blending.y + Zdir * blending.z;
...
}
```

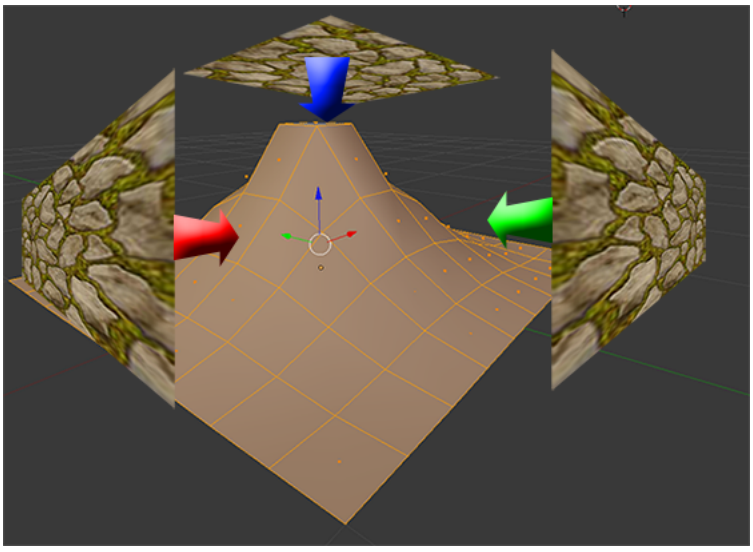

Figure 4: Tri-planar Texturing visualisation [12]

The material for the terrain just uses a diffuse texture, which is blended onto the terrain with the just described technique. All textures have been retrieved from the internet except the grass texture which was provided by the VVE itself [1].

4.5 Terrain interaction

A terrain without interaction would be boring, therefore it is necessary to interact with the terrain in a meaningful way. Firstly it is crucial to hit the mesh of the terrain at any given angle and position. This requires the ability to shoot a ray through a pixel and then search for a ray-terrain intersection. This code snippet (B.6) resides in the EventListenerUser and gives the user the ability to cast a ray from the camera center of projection (COP) through the pixel. The ray itself has only two properties, the origin and the direction. Shooting a ray through a pixel has to take almost every camera attribute into account. The center of projection equals the position of the camera. On the other hand the direction needs to translate the pixels from Raster Space to Screen Space[2].

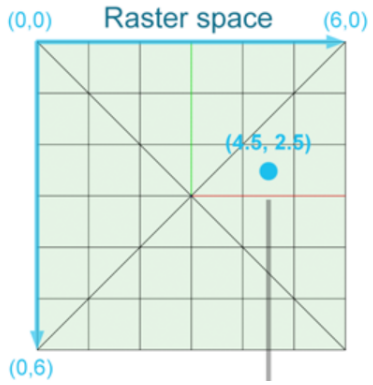

Figure 5: Raster Space [2]

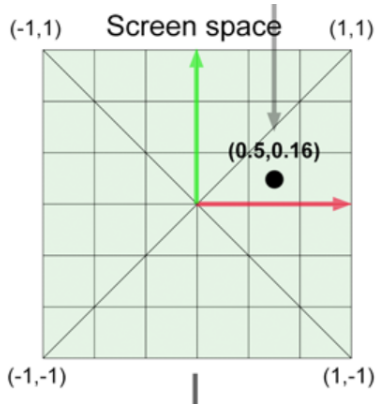

Figure 6: Screen Space [2]

This is achieved by the xx and yy variables. These variables also take the camera field of view (FOV) into account and the screen aspect ratio. The final direction is calculated from the local camera CS and Screen Space coordinates. This is needed to take the actual camera angle into account and not just the pixel direction from a camera with a fixed direction. Since the camera is looking directly into the z+ axis, the camera direction can be taken directly from the camera matrix (the third column). This ray is propagated to the Outdoor Editor, which can trace the ray and outputs a boolean if it has hit something and a parameter where it has hit B.5. Multiple chunks have to be checked it the ray has hit something. To avoid endless checking the ray has a maximum distance it can travel and checks each chunk only once if it has hit something. But now the question arises how does a chunk check if a ray has actually hit the terrain. Each triangle is extracted from each cell in the TerrainMeshChunk class (B.7) and tested if the ray has hit something with the ray-triangle-intersection

test [3]. The test (B.3) checks whether the dot product of the surface normal and ray direction are parallel or not and if the ray comes from behind.

$$a * x + b * y + c * z + d = 0 \tag{4}$$

$$O + \alpha * R \tag{5}$$

If not, then the plane equation is used to check if the ray would hit an infinite plane by calculating the alpha part of the ray, which is accomplished by substituting x, y, z of the plane equation with the ray equation. If yes, then the cross product is used to determine if the hit was inside the triangle. The product uses the edge from a vertice to the hitpos and and an edge of of the triangle. Then the test checks if the hitpos was right of the triangle edge 3 times. If yes, then the ray has hit the terrain successfully [3].

4.6 Brushes

Now that the terrain can be hit per mouse click, the question arises how should the editing of the terrain work exactly. The decision was using brushes to tell the editor how much the area, around the position where the terrain was hit, is affected. A brush consists of the three parts

- radius,
- strength,
- and mathematical falloff function.

A brush is implemented via the strategy pattern with a concrete maths function which returns a vector of the Voxel points, affected within the radius, called VoxelPoints. The Outdoor Editor has 3 brushes implemented, the smooth sphere brush, the full sphere brush and the drill brush, which is basically the same as the full brush, but with a fixed radius. These brushes are abstract in nature, but their functionality can be visualised with the following picture.

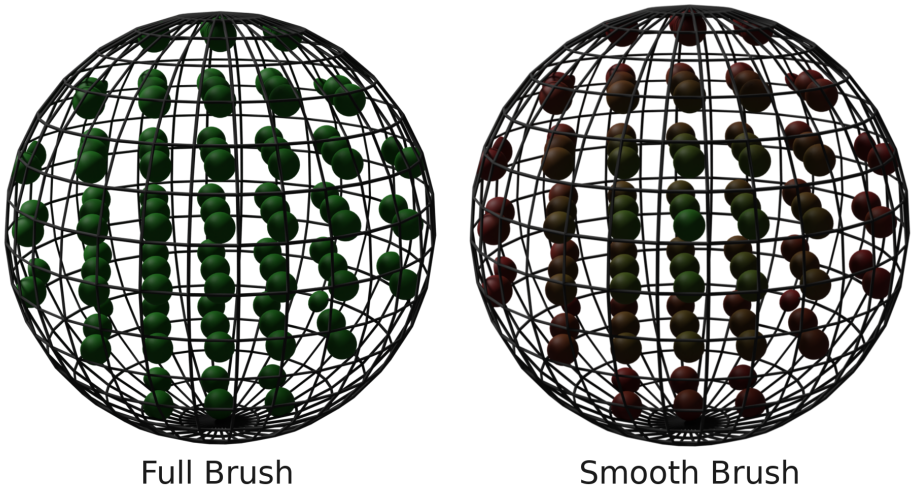

Figure 7: Brush affected areas

The spheres represent junction points in the 3D Grid (Voxels) and how much a sphere is affected (green completely, red unaffected). The radius of a brush is symbolised by the wireframe sphere. Noteworthy is that the Voxel grid is cut in half inside the sphere to better visualise the effect. The full brush just takes every Voxel inside the radius and applies its strength to the Voxel. The Smooth Brush uses a cosine function to create a better sphere effect. At first glance this might look like a sphere inside a sphere, but the results differ a lot from each other, when using it to edit the terrain A.1 A.2. There are many falloff implementations, but these two suffice for a simple terrain editing tool B.8.

4.7 Nature Entities - Importing Models

Models are imported with the help of the type object pattern, but some resources have to be imported in a different way than a solid model. The best example for this is the NatureEntityLeafs class. Normally a model is simply imported via a .obj file, which contains a single object. The NatureEntityLeafs class (B.9) however is able to process more than one object or leaves in this case and then cuts off a certain amount of leaf branches (objects) to give a tree for example more variety without having to create multiple trees. The branch cutoff is implemented via a bernoulli distribution, which is used to create random booleans, which can be controlled stochastically by providing it with the desired cutoffRatio between 0.0 and 1.0. This was a particular difficult task, because the VVE uses the assimp importer to load models and needs to extract each object from the assimp tree manually, in order to achieve the branch cutoff. Another way to import models is using the NatureEntityBillboard(B.10),

which only takes a picture as input and projects this picture onto a plane, which rotates according to the camera angle and position [7]. The plane itself is generated in the geometry shader, which uses just a single vertice as a position. At end the question arises, can all of these different entities be combined into a new entity. The answer is yes, by using a composite pattern which delegates the createEntity task down to each leaf node, which is achieved by the NatureEntityModel class. The NatureEntityManager decides which NatureEntity is best suited for importing a certain resource and how these entities should be combined to create a new model.

4.8 The Outdoor Editor and its Sub-Managers

The Outdoor Editor ties everything together (B.11), but delegating a task to the correct manager function by function would bloat the class unnecessarily. To counter this issue, the class provides a single function called handleInput(...), which takes only a position where something is happening, probably the hitPos (4.5), and a direction. There is also the parameter to invert a certain operation if the given state has the ability to do so. In order to control what is happening, when this method is invoked, it uses an enum called oeEditingModes, which represents the current editing mode. The volume mode allows editing the terrain, the texture mode allows giving the terrain a different texture and the object mode allows the user to place different models. The NatureEntityManager uses a similar approach with the oeEntityModel (A.2), each enum represents a model which can be placed by the editor.

If the terrain was modified, multiple managers have to be synchronised and work together. The first design decision is the removal of NatureEntities, if the entity is within the soon to be modified area (brush radius). When the terrain is going to be modified, the VoxelData has to be changed first. Then the Outdoor Editor uses a method called refresh, which decides if the terrain has to refresh an entire chunk or refresh just a cell and its neighbours (two methods in the TerrainManager). It decides this by looking at how many voxels have been changed and if it crosses the CHUNKS_CHANGED_VOXELS_THRESHOLD then the entire chunk has to be refreshed. Otherwise only the cells around the voxel and its neighbours will be refreshed. This has been implemented to increase the performance for editing the terrain.

After the scene has been set up, it can be saved as a .json file. This is achieved by an external library [8] (B.12). The .json file saves Voxel data and nature entities. The save and load functions can be found in the NatureEntityManager and the VoxelDataManager, which only saves and loads the part these managers and its components are responsible for. In case of the NatureEntityManager all information needed to recreate model is already in the manager, but the VoxelManager delegates the task further to a VoxelChunk, to save the parts it is responsible for. This allows future programmers to add packages without having to know about the other packages. They just have to invoke the save and load function of the respective package/class since every package/class knows how to save and load itself.

4.9 Transparency Subrender

The Transparent Subrenderer is not listed in the class diagram, because it has to be included directly into the Vienna Vulkan Engine. This subrender is needed to actually make use of the alpha channel of leaves and other textures. To enable this feature the color blending of pixels has to be modified. The color in the frame buffer (destinationColor) will be blended together with the new color from the fragment shader (sourceColor). How exactly this will be processed is defined by the following piece of code while building the graphicsPipeline [11].

```
...

VkPipelineColorBlendAttachmentState colorBlendAttachment = {};
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT |
VK_COLOR_COMPONENT_A_BIT;

colorBlendAttachment.blendEnable = VK_TRUE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;

colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;

...
```

The new color of the pixel will be calculated in the following way [11]

$$finalColor = (sourceAlpha * newColor) + ((1 - sourceAlpha) * oldColor); \quad (6)$$

$$finalAlpha = (1 * newAlpha) + (0 * oldAlpha) \quad (7)$$

But while correctly doing so a problem was encountered. The alpha parts had a greyish semi transparent color. After further analysis of the problem, it was discovered that the light passes have trouble handling transparency, because normally the solid objects are separated from the transparent objects. The code is unable to fix the underlying problem, but disallowing the depthWriteEnable the grey semi transparent color disappears, but with the drawback that transparent objects are now drawn inversely to the order they were placed.

```
...
VkPipelineDepthStencilStateCreateInfo depthStencil = {};
depthStencil.sType =
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_FALSE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.stencilTestEnable = VK_FALSE;
...
```

4.10 Decoupling from VVE

One of the tasks is to decouple the Outdoor Editor as much as possible from the VVE. These following classes are coupled with the VVE.

- MarchingCubes ⇒ uses vhVertex
- TerrainMeshChunk ⇒ creates materials, meshes, scene nodes and entities
- NatureEntity ⇒ loads models
- NatureEntityLeafs ⇒ uses loadAssets
- NatureEntityBillboard ⇒ creates a material with one diffuse texture, rntity and a mesh with one vertice (vhVertex)
- NatureEntityManager ⇒ creates SceneNodes
- EventListenerGUI, EventListenerUser, OutdoorEditor ⇒ inherit from VVE classes

5 Technology Stack

- Visual Studio 2019 with c++20 ⇒ for creating the Outdoor Editor code and importing the VVE
- Blender 2.9 ⇒ to create sample models used by the Outdoor Editor
- Visual Paradigm Community Edition ⇒ in order to design the class diagram (A.2)
- Gimp ⇒ to adjust textures for the models built in blender.

6 Evaluation and Discussion

In this chapter the discussion revolves around whether the Outdoor Editor has fullfilled its task and how it compares to the stand alone Vienna Vulkan Engine. At the end some snapshot will be presented to see what the Outdoor Editor is currently capeable of.

6.1 Outdoor Editor goals

The goals described in the introduction 1 have all been fullfilled. The Outdoor Editor is able to create a Voxel based terrain which is rendered with the help of the Marching Cubes algorithm as the pictures can demonstrate (A.1, A.2). The terrain is also interactive and gives instant feedback what area is currently affected by a red circle which represents the radius of the current brush. An intuitive GUI has also been provided which shows the options the user is able to use to create the scene of his choosing. Modifying the terrain can be achieved by simply clicking on the application window and the process will be handled according to the selection of the current GUI. The same process also applies to the object placement by the combination of the GUI selection and a simple click on the terrain. The question whether the Outdoor Editor is reusable for other projects can be answered with yes, because the code base is open ended e.g. the Marching Cubes algorithm can easily be switched out by another algorithm. Available models can also be switched out by simply changing two classes, the NatureEntityManager and the EnitityDatabase by adding the path to the model. A different project for example could be a planet editor. To achieve this some models would have to be adjusted and another set of Voxel data, instead of a plane, would have to be used. In the end the main goal was achieved, which was the visualisation of setting up a scene in the Vienna Vulkan Engine without any need to set it up by code and having to imagine the whole scene while coding it. The Outdoor Editor does not perform better against an engine like Unity, which provides a lot more features and has a huge team of developers behind it, but the Outdoor Editor still can hold its ground for setting up a simple Voxel based scene.

6.2 Comparison to simple VVE

Compared to the VVE the model importing is now streamlined and just requires little changes in the NatureEntityManager and EnitityDatabase. This is advantageous, because now not every model has to be imported via different classes, but can be instantiated by simply using the NatureEntityManager. This disincentives other programmers loading models in their own classes and thus prevents a shotgun surgery if the VVE is being adapted or rewritten. Another advantage over the base VVE is that a complex terrain can be edited without having to code the terrain. A simple look at the class diagram (A.2) will suffice to see, that a lot of the complexity is handled by the editor and the user does not have to bother with the terrain anymore. And finally the feedback is instant while setting up the scene and the Outdoor Editor and no code to create a scene is required, except the NatureEntity_t data to import the desired models. A disadvantage would be of course that there is a certain loss of freedom, because the Outdoor Editor restricts the user to use Voxel terrains and also restricts the choice of models provided in the NatureEntityManager, which could be generated by code, but the user would also be able to extend the variety of models.

6.3 Outdoor Editor in Action

Here we can see what kind of scenes the Outdoor Editor is capable of setting up.

Figure 8: The stag hunt

Figure 9: Stag cult

Noteworthy about the picture is is that the animals are placed in the direction away from the user.

6.4 Encountered Issues

The issues mentioned here have been found in VVE while working on the Outdoor Editor. Some have already been fixed, while others proved to be more

difficult.

6.4.1 Alpha Blending

As mentioned before (4.9) the transparency of a model is impaired by a suspected problem with the render-passes, which do not handle transparency separately from solid objects. Thus a cheap fix has been implemented which sadly reverses the drawing order of transparent objects.

6.4.2 Vulkan Mesh Resources

While creating the TerrainMeshChunk class, an error occurred while adding and deleting sceneNodes. It was found out that the problem was an error was caused by the VESubrenderFW.

```
void VESubrenderFW::removeEntity(VEEntity *pEntity) {

...

//Before
m_descriptorSetsResources.resize(m_entities.size());//remove descriptor sets

//After
m_descriptorSetsResources.resize(m_entities.size() /
    m_resourceArrayLength);//remove descriptor sets

...

}
```

The error was an edge case which created NULL VkDescriptorSet in m_descriptorSetsResources and this would cause trouble later in VESubrenderFW::addMaps(...), which tried to access NULL values.

6.4.3 loadAssets problem

Another problem that occured while creating the NatureEntityLeafs class, was that the aiScene pointer returned by the loadAssets(...) function, pointed to an already deleted object. After reading the assimp documentation in more detail the complication could be resolved by using the importer getOrphanedScene() function.

```
aiScene* VESceneManager::loadAssets(std::string basedir, std::string filename,
    uint32_t aiFlags, std::vector<VEMesh*> &meshes, std::vector<VEMaterial*> &materials) {
    ...
    //Before
    return pScene;

    //After
    return importer.GetOrphanedScene();
}
```

The problem was that the importer held the aiScene and when the importer calls the destructor the aiScene will be destroyed, which the pScene pointer actually pointed at. The GetOrphanedScene() function returns the aiScene, but the outside is now responsible for deleting the scene object.

6.4.4 Mouse Movement no cursor coordinates

This is not an error but was a necessity for creating the Outdoor Editor. The problem was the Mouse movement event did not have access to the cursor coordinates.

```
void VEWindowGLFW::mouse_button_callback (GLFWwindow* window,
    int button, int action, int mods) {
        ...
    //Before
    veEvent event(veEvent::VE_EVENT_SUBSYSTEM_GLFW, veEvent::VE_EVENT_MOUSEBUTTON);
    event.idata1 = button;
    event.idata3 = action;
    event.idata4 = mods;
    event.ptr = window;
    app->processEvent(event);

    //After
    double xpos, ypos;
    //getting cursor position
    glfwGetCursorPos(window, &xpos, &ypos);
    veEvent event(veEvent::VE_EVENT_SUBSYSTEM_GLFW, veEvent::VE_EVENT_MOUSEBUTTON);
    event.fdata1 = static_cast<float>(xpos);
    event.fdata2 = static_cast<float>(ypos);
    event.idata1 = button;
    event.idata3 = action;
    event.idata4 = mods;
    event.ptr = window;
    app->processEvent(event);
        ...
}
```

This was fixed by adding 4 lines in the VEWindowGLFW::mouse_button_callback(...) function, which extract the cursor position and pass on the information in the event object (fdata1 and fdata2). This information is later needed in the onMouseMove event.

7 Conclusion

The main goal of the Outdoor Editor was to create the first interactive scene editor on top of the Vienna Vulkan Engine. The editor provides the user with the option to edit the terrain and the ability to place models onto the terrrain. Thus the time needed to set up a scene can be significantly reduced by just adding the desired models to the Outdoor Editor. This allows future programmers to focus more on models and level design, instead of taking care of a huge amount of code in order to create a new scene. The Outdoor Editor is also open ended and provides a lot of possibilities for improvements in performance and also allows the implementation of new features.

The next step to make the editor more interesting would be to increase the performance for creating a chunk. Parallelisation of this process will probably provide a huge performance boost. Another performance increase would be the loading of the mesh, which could be coupled with what the camera is currently looking at, to reduce the sceneNodes in the VVE. In the model area a feature could be added to use level of detail (LOD) to further increase performance. And at last many small features could be added such as using new brushes to give the user more tools to edit the terrain. In the end the Outdoor Editor does not have to revolve around nature, it could be used as a basis for different projects just by providing new models and Voxel data.

References

[1] 3d textures. https://3dtextures.me/(07.02.22).

[2] Ray-tracing: Generating camera rays. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays(05.02.22).

[3] Ray tracing: Rendering a triangle. //https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution(05.02.22).

[4] Marching cubes cases. https://en.wikipedia.org/wiki/File:MarchingCubes.svg(03.02.2022), (accessed 03.02.2022).

[5] Bourke, P. Marching cubes algorithm. https://paulbourke.net/geometry/polygonise/(03.02.22), 1994.

[6] Hlavacs, H. Viennavulkanengine. https://github.com/hlavacs/ViennaVulkanEngine, (accessed 06.01.2022).

[7] Lapinski, P. *Vulkan Cookbook Work through recipes to unlock the full potential of the next generation graphics API-Vulkan, ISBN 978-1-78646-815-4*. Packt, 2017, pp. 601–607.

[8] Lohmann, N. Json for modern c++. https://github.com/nlohmann/json(06.02.22), (accessed 06.02.22).

[9] Lorensen, W., and Cline, H. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics 21* (08 1987), 163–.

[10] Nystrom, R. *Game Programming Patterns, ISBN: 978-0-9905829-2-2*. online(https://gameprogrammingpatterns.com/ (31.01.2022)), 2014, pp. 193–212.

[11] Overvoorde, A. Vulkan tutorial. https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Fixed_functions(06.02.22), 2020.

[12] Owens, B. Use tri-planar texture mapping for better terrain. https://gamedevelopment.tutsplus.com/articles/use-tri-planar-texture-mapping-for-better-terrain--gamedev-13821(04.02.22), 2014.

A Appendix-Pictures

A.1 Brush Full in Action

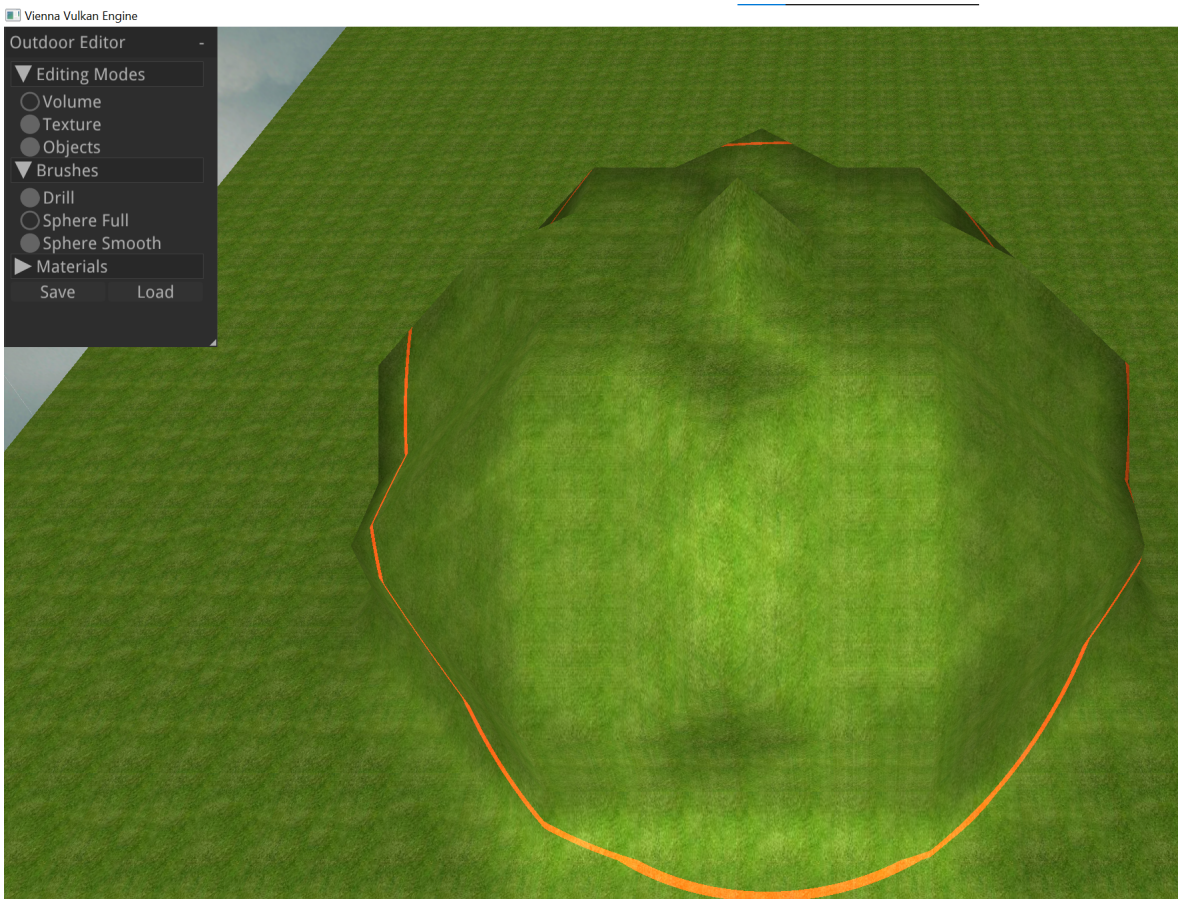

Figure 10: Full Sphere Brush in Action

A.2 Brush Smooth in Action

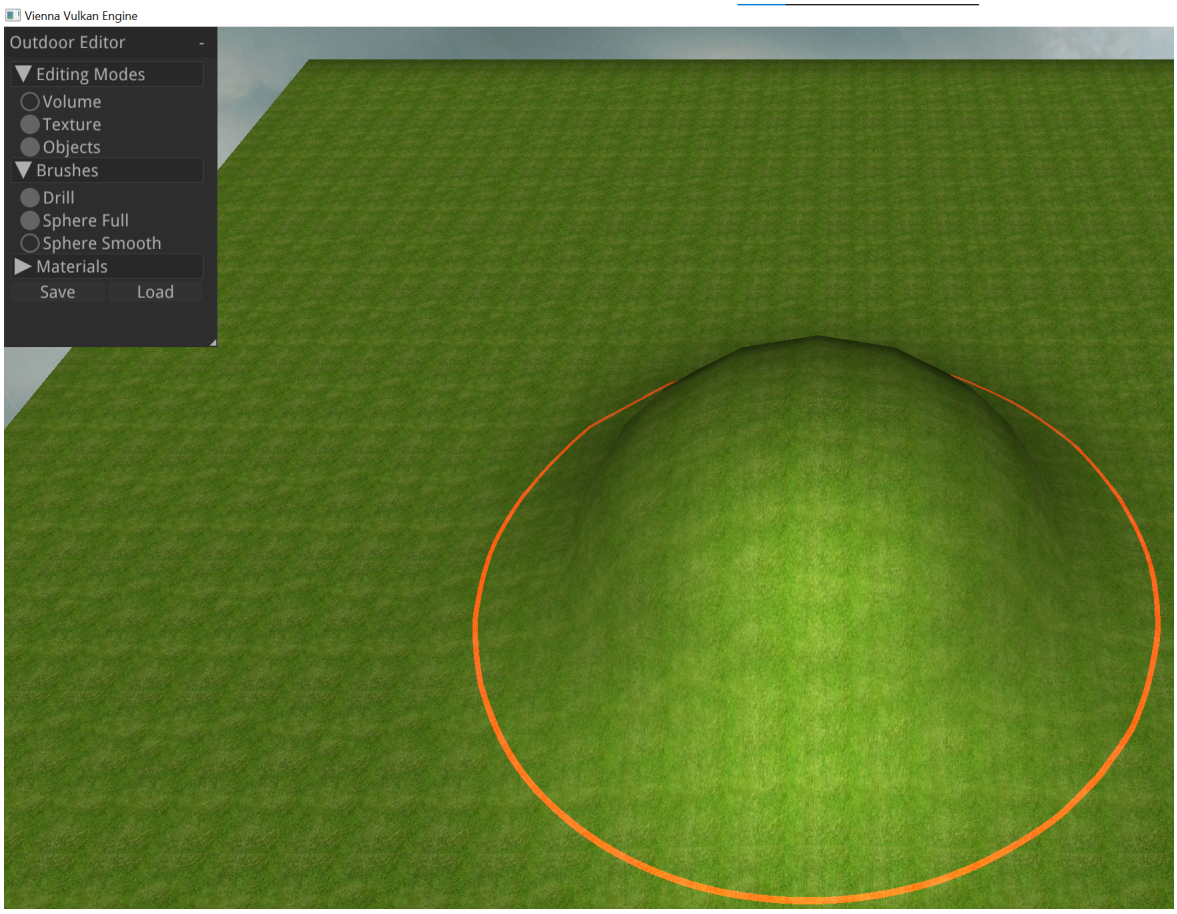

Figure 11: Smooth Sphere Brush in Action

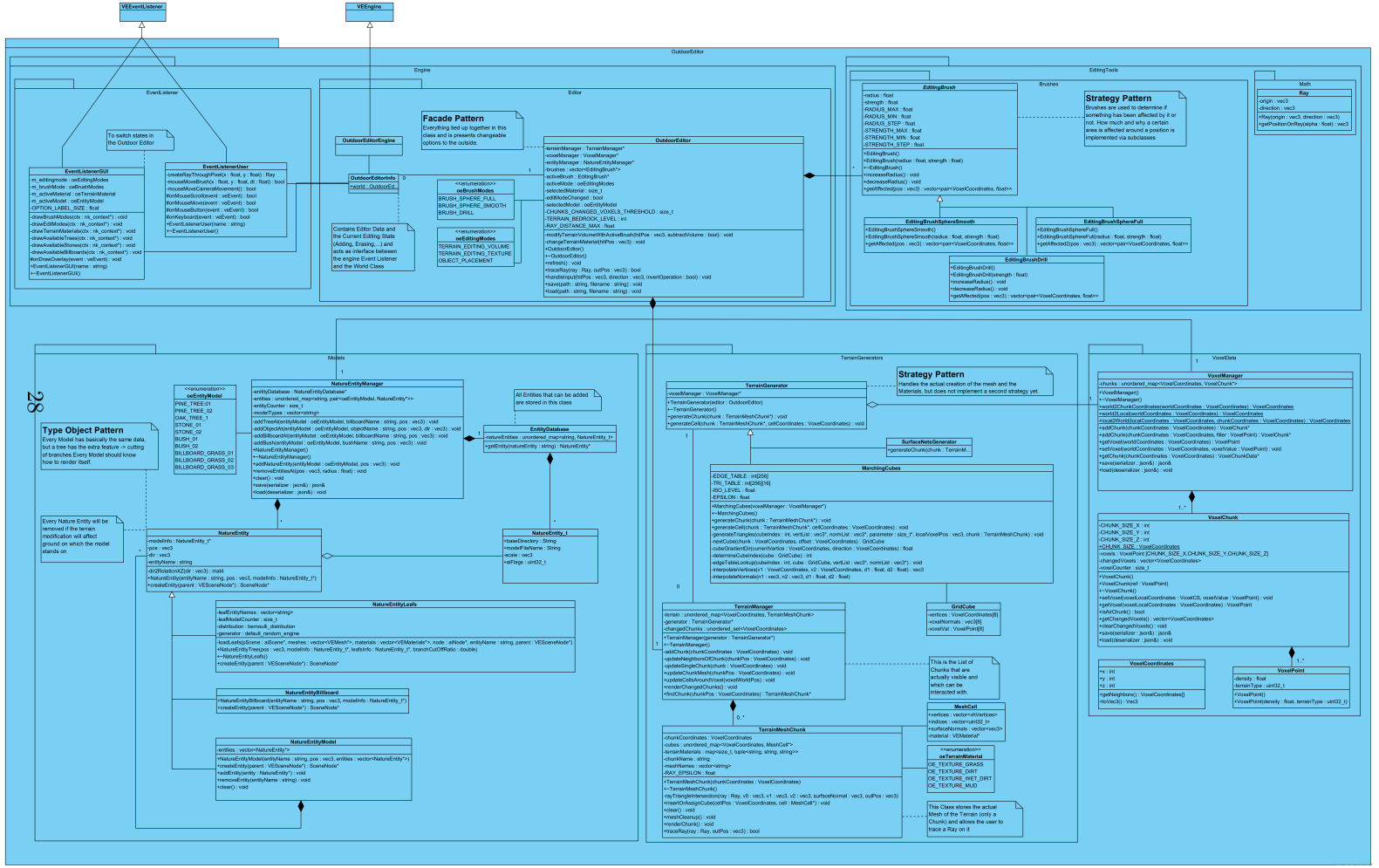

Figure 12: Outdoor Editor Class Diagram

B Appendix-Code

B.1 VoxelData CS transformation

```
VoxelCoordinates VoxelManager::world2Local
    (const VoxelCoordinates& worldCoordinates)
{
        auto x = worldCoordinates.X < 0 ?
        (VoxelChunkData::CHUNK_SIZE.X - 1) +
        ((worldCoordinates.X + 1) % VoxelChunkData::CHUNK_SIZE.X) :
        worldCoordinates.X  % VoxelChunkData::CHUNK_SIZE.X;

        auto y = worldCoordinates.Y < 0 ?
        (VoxelChunkData::CHUNK_SIZE.Y - 1) +
        ((worldCoordinates.Y + 1) % VoxelChunkData::CHUNK_SIZE.Y) :
        worldCoordinates.Y % VoxelChunkData::CHUNK_SIZE.Y;

        auto z = worldCoordinates.Z < 0 ?
        (VoxelChunkData::CHUNK_SIZE.Z - 1) +
        ((worldCoordinates.Z + 1) % VoxelChunkData::CHUNK_SIZE.Z) :
        worldCoordinates.Z % VoxelChunkData::CHUNK_SIZE.Z;

        return VoxelCoordinates(x, y, z);
}
VoxelCoordinates VoxelManager::local2World
    (const VoxelCoordinates& localCoordinates,
    const VoxelCoordinates& chunkCoordinate)
{
        return localCoordinates + VoxelChunkData::CHUNK_SIZE * chunkCoordinate;
}
VoxelCoordinates VoxelManager::world2ChunkCoordinates
    (const VoxelCoordinates& worldCoordinate)
{
        auto x = worldCoordinate.X < 0 ?
        -1 + (worldCoordinate.X + 1) / VoxelChunkData::CHUNK_SIZE.X :
        worldCoordinate.X / VoxelChunkData::CHUNK_SIZE.X;

        auto y = worldCoordinate.Y < 0 ?
        -1 + (worldCoordinate.Y + 1) / VoxelChunkData::CHUNK_SIZE.Y :
        worldCoordinate.Y / VoxelChunkData::CHUNK_SIZE.Y;

        auto z = worldCoordinate.Z < 0
        ? -1 + (worldCoordinate.Z + 1) / VoxelChunkData::CHUNK_SIZE.Z :
        worldCoordinate.Z / VoxelChunkData::CHUNK_SIZE.Z;

        return VoxelCoordinates(x, y, z);
```

```
}

void VoxelManager::setVoxel
    (const VoxelCoordinates& worldCoordinates,
    const VoxelPoint& voxelValue)
    {
        VoxelCoordinates localCoordinates =
            world2Local(worldCoordinates);
        VoxelCoordinates chunkCoordinates =
            world2ChunkCoordinates(worldCoordinates);

        ...
}

VoxelPoint VoxelManager::getVoxel
(const VoxelCoordinates& worldCoordinates) const {
        VoxelCoordinates localCoordinates = world2Local(worldCoordinates);
        VoxelCoordinates chunkCoordinates =
            world2ChunkCoordinates(worldCoordinates);

        ...
}
```

B.2 Marching Cubes Mesh generation

```
void MarchingCubes::generateChunk(TerrainMeshChunk* mc) const
{
        if (mc == nullptr) return;

        for (auto x = 0; x < VoxelChunkData::CHUNK_SIZE.X; ++x) {
        for (auto y = 0; y < VoxelChunkData::CHUNK_SIZE.Y; ++y) {
        for (auto z = 0; z < VoxelChunkData::CHUNK_SIZE.Z; ++z) {
                int cubeIndex = 0;
                GridCube cube = nextCube(mc->getChunkCoordinates(),
                    VoxelCoordinates(x, y, z));

                cubeIndex = determineCubeIndex(cube);

                glm::vec3 vertList[12] = {};
                std::size_t terrainMaterial[12] = {};
                glm::vec3 normList[12] = {};
                /* Cube is entirely in/out of the surface ->
                nothing will be drawn next Cube */
                if (EDGE_TABLE[cubeIndex] == 0) continue;

                edgeTableLookup(cubeIndex, cube, vertList,
                    terrainMaterial, normList);

                generateTriangles(cubeIndex, vertList,
                    terrainMaterial, normList,
                    VoxelCoordinates(x, y, z), mc);
    }
    }
    }
}

void MarchingCubes::generateCell(TerrainMeshChunk* mc,
const VoxelCoordinates& cellCoordinates) const
{
        if (mc == nullptr) return;

        int cubeIndex = 0;

        GridCube cube = nextCube(mc->getChunkCoordinates(), cellCoordinates);

        cubeIndex = determineCubeIndex(cube);

        glm::vec3 vertList[12] = {};
        std::size_t terrainMaterial[12] = {};
```

```
        glm::vec3 normList[12] = {};

        edgeTableLookup(cubeIndex, cube, vertList,
            terrainMaterial, normList);

        generateTriangles(cubeIndex, vertList,
            terrainMaterial, normList, cellCoordinates, mc);
}
```

B.3 Ray Triangle Intersection

```
/*
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray
-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution
(04.11.21)
*/
//Implements and Explains the Geometric intersection
//algorithm with a plane equation and left right tests to
//ensure the hit point is in the triangle
bool TerrainMeshChunk::rayTriangleIntersection(const Ray&
    ray, const glm::vec3& v0, const glm::vec3& v1,
    const glm::vec3& v2, const glm::vec3& surfaceNormal,
    glm::vec3& outPos) const
{
        float surfaceNormalDOTRayDirection = glm::dot(surfaceNormal, ray.getDirection());

        //Check if Direction of Ray and Surface normal are
        //perpendicular (Direction of ray
        //is parallel to triangle)

        if (std::abs(surfaceNormalDOTRayDirection) < RAY_EPSILON)
                return false;

        //Direction and Surface normal should oppose each other
        //so if it bigger than 0 then they point in
        //the same direction
        if (0.0f < surfaceNormalDOTRayDirection)
                return false;

        //Calculate alpha to determine if triangle was hit by the ray
        //it uses the Ray Equation P = alpha * direction +
        //origin and the plane Equation: A*x + B*y + C*z + D = 0
        //(x = Px, y = Py, z=Pz)
        float D = -glm::dot(surfaceNormal, v0);
        float alpha = -(glm::dot(surfaceNormal,
            ray.getOrigin()) + D) / surfaceNormalDOTRayDirection;

        //Ray hits a triangle behind the origin -> no hit
        if (alpha < 0.0f)
                return false;

        //Successful hit (it has hit the plane not the triangle yet)
        glm::vec3 hitPos = ray.getPositionOnRay(alpha);

        //Test if hit was inside Triangle (right is below 0.0f so return false if left)
```

```
        glm::vec3 e01 = v1 - v0;
        glm::vec3 v0toHit = hitPos - v0;
        glm::vec3 c0 = glm::cross(e01, v0toHit);
        if (glm::dot(surfaceNormal, c0) > 0.0f)
                return false;

        glm::vec3 e12 = v2 - v1;
        glm::vec3 v1toHit = hitPos - v1;
        glm::vec3 c1 = glm::cross(e12, v1toHit);
        if (glm::dot(surfaceNormal, c1) > 0.0f)
                return false;

        glm::vec3 e20 = v0 - v2;
        glm::vec3 v2toHit = hitPos - v2;
        glm::vec3 c2 = glm::cross(e20, v2toHit);
        if (glm::dot(surfaceNormal, c2) > 0.0f)
                return false;

        //Hit is inside Triangle return hitPos and true for hit
        outPos = hitPos;
        return true;
}
```

B.4 Terrain Mesh Chunk generation

```
void TerrainMeshChunk::renderChunk(){
        meshCleanup();

        //Create Chunk SceneNode
        auto pScene = getSceneManagerPointer()->getSceneNode("Scene");
        VESceneNode* pChunkNodeParent;

        VECHECKPOINTER(pChunkNodeParent =
            getSceneManagerPointer()->createSceneNode
            (chunkName, pScene, glm::mat4(1.0)));

        glm::vec3 nextChunkPos = (chunkCoordinates *
            VoxelChunkData::CHUNK_SIZE).toVec3();

        pChunkNodeParent->multiplyTransform(
            glm::translate(glm::mat4(1.0f), nextChunkPos));


        //Create Mesh for each cube stored in cubes
        for (auto& cube : cubes) {
                std::string cubeName = chunkName + "_Cube[" +
                    std::to_string(cube.first.X) + "," +
                    std::to_string(cube.first.Y) + "," +
                    std::to_string(cube.first.Z) + "]";

                meshNames.push_back(cubeName);

                //Actual Engine Mesh generation
                    //(Creates only a mesh if there are vertices)
                VEMesh* cube_mesh = nullptr;
                VECHECKPOINTER(cube_mesh =
                    getSceneManagerPointer()->createMesh
                    (cubeName + "_Mesh", cube.second->vertices,
                    cube.second->indices));

                VEMaterial* cube_material = getTerrainMaterial(cube.second->material);

                //CreateCube Entity
                VEEntity* entity;
                VECHECKPOINTER(entity =
                    getSceneManagerPointer()->createEntity
                    (cubeName + "_Entity",
                    VEEntity::veEntityType::VE_ENTITY_TYPE_VOXEL_TERRAIN,
                    cube_mesh, cube_material, pChunkNodeParent));
```

```
                entity->multiplyTransform(glm::translate
                    (glm::mat4(1.0f), cube.first.toVec3()));

            }

        }
```

B.5 Outdoor Editor traceRay

```
bool OutdoorEditor::traceRay
    (const Ray& ray, glm::vec3& outPos) const
{
        TerrainMeshChunk* oldMeshChunk = nullptr;
        float distance = 0.0f;
        while (distance < RAY_DISTANCE_MAX) {

                auto rayPos = ray.getPositionOnRay(distance);
                auto rayChunkCoordinates =
                    VoxelManager::world2ChunkCoordinates
                    (VoxelCoordinates((int)rayPos.x,
                    (int)rayPos.y, (int)rayPos.z));


                TerrainMeshChunk* chunk =
                    terrainManager->findChunk(rayChunkCoordinates);

                if ((chunk != nullptr) && (chunk != oldMeshChunk)) {
                        oldMeshChunk = chunk;

                        bool hit = chunk->traceRay(ray, outPos);
                        if (hit) {
                                return hit;
                        }
                }
                distance += 1.0f;
        }
        return false;
}
```

B.6 Ray casting - shoot ray through a pixel

```
Ray EventListenerUser::createRayThroughPixel
    (const float& cursorX, const float& cursorY) const
{

        VkExtent2D window = getEnginePointer()->getWindow()->getExtent();
        VECameraProjective* pCamera =
            dynamic_cast<VECameraProjective*>
            (getSceneManagerPointer()->getCamera());

        //Helper Variables
        float aspectRatio = pCamera->m_aspectRatio;
        float invWidth = 1.0f / window.width;
        float invHeight = 1.0f / window.height;

        float fovVertical = std::tan(glm::radians(
            pCamera->m_fov * 0.5f));

        //Screen Space
        float xx = (((cursorX + 0.5f) * invWidth)
            * 2 - 1) * aspectRatio * fovVertical;
        float yy = (1 - 2 * ((cursorY + 0.5f) * invHeight))
            * fovVertical;


        //Camera CS to calculate the direction of the ray
        //(if the camera Rotates then the Ray has to adjust accordingly)
        glm::vec3 u = pCamera->getWorldTransform() *
            glm::vec4(1.0f, 0.0f, 0.0f, 0.0f);
        glm::vec3 v = pCamera->getWorldTransform() *
            glm::vec4(0.0f, 1.0f, 0.0f, 0.0f);
        glm::vec3 camDir = pCamera->getWorldTransform() *
            glm::vec4(0.0f, 0.0f, 1.0f, 0.0f);

        //creating Ray
        glm::vec3 cameraPos = pCamera->getWorldTransform()[3];
        glm::vec3 direction = u * xx + v * yy + camDir;

        Ray r(cameraPos, direction);
        return r;
}
```

B.7 trace Ray in TerrainMeshChunk

```
bool TerrainMeshChunk::traceRay
    (const Ray& ray, glm::vec3& outPos) const
{

        //Trace All Triangles and
        //if a Triangle is hit return true
        for (const auto& cube : cubes) {
                const auto vertices = cube.second->vertices;
                const auto indices = cube.second->indices;

                for (std::size_t i = 0, j = 0;
                    i < indices.size(); i += 3, ++j) {
                        auto chunkOffset = (VoxelChunkData::CHUNK_SIZE *
                            chunkCoordinates).toVec3();

                        auto cubePosOffset = cube.first.toVec3();

                        auto v0 = vertices.at(indices.at(i)).pos +
                            chunkOffset + cubePosOffset;
                        auto v1 = vertices.at(indices.at(i + 1)).pos +
                            chunkOffset + cubePosOffset;
                        auto v2 = vertices.at(indices.at(i + 2)).pos +
                            chunkOffset + cubePosOffset;

                        auto surfaceNormal =
                            cube.second->surfaceNormals.at(j);

                        bool hit = rayTriangleIntersection(ray, v0, v1, v2,
                            surfaceNormal, outPos);
                        if (hit) {
                                return hit;
                        }
                }
        }
        return false;
}
```

B.8 Full and Smooth Brush

```
//Full Brush
std::vector<std::pair<VoxelCoordinates, float>> EditingBrushSphereFull::getAffected
    (const glm::vec3& pos) const
{
        std::vector<std::pair<VoxelCoordinates, float>> ret;
        VoxelCoordinates midPoint(std::lround(pos.x), std::lround(pos.y),
            std::lround(pos.z));

        long iRadius = lround(radius);
        long invRadius = lround(-radius);
        for (auto x = invRadius; x <= iRadius; ++x) {
                for (auto y = invRadius; y <= iRadius; ++y) {
                        for (auto z = invRadius; z <= iRadius; ++z) {
                                float length = glm::length(glm::vec3(x, y, z));
                                if (length > radius) {
                                        continue;
                                }
                                VoxelCoordinates nextPos = midPoint +
                                    VoxelCoordinates(x, y, z);

                                ret.push_back(std::make_pair(nextPos, strength));
                        }
                }
        }
        return ret;
}

//Smooth Brush (same as above but different falloff function)
...
float length = glm::length(glm::vec3(x, y, z));
if (length > radius) {
        continue;
}
VoxelCoordinates nextPos = midPoint + VoxelCoordinates(x, y, z);
//We know length is  0 < length <= radius
//cos(2 * pi / f) * amplitude
//We want cos from 0 to pi / 2
//this is f by varying radius and calculated length in order to get pi / 2
// 2*pi* 1/f  * radius * length == pi / 2
// => 4*radius*length and a shortcut by using PI
//instead of 2*Pi results in 2 * radius * length
// so in the end we want f = 2*r^2 because this is the maximum the
//length we can achieve
float frequency = (2.0f * radius * radius);
```

```
float nextStrength =  std::cos((static_cast<float>(M_PI) / frequency) *
    radius*length)*strength;
ret.push_back(std::make_pair(nextPos, nextStrength));
...
```

B.9 NatureEntityLeafs - branch cutoff

```
void NatureEntityLeafs::createEntity(VESceneNode* parent)
{
        std::vector<ve::VEMesh*> leafMeshes;
        std::vector<ve::VEMaterial*> leafMaterials;

        NatureEntity_t* leafsInfo = getModelInfo();
        glm::vec3 pos = getPos();

        aiScene* scene =
            getSceneManagerPointer()->loadAssets(leafsInfo->baseDirectory,
            leafsInfo->modelFileName, leafsInfo->aiFlags, leafMeshes,
            leafMaterials);
        aiNode* pRoot = scene->mRootNode;

        loadLeafs(scene, leafMeshes, leafMaterials,
            pRoot, getEntityName(), parent);

        parent->multiplyTransform(glm::scale(leafsInfo->scale));
        parent->setPosition(pos);

        delete scene; // We are now responsible for destroying the imported scene
}

//copied from copyaiNode but with slight adjustments to enable
//branch building with probabilities
void NatureEntityLeafs::loadLeafs(const aiScene* pScene,
    std::vector<VEMesh*>& meshes, std::vector<VEMaterial*>& materials,
    aiNode* node, const std::string& entityName, VESceneNode* parent)
{
    //go through the meshes of the Assimp node
        for (uint32_t i = 0; i < node->mNumMeshes; i++) {

                if (distribution(generator)) { continue; }

                VEMesh* pMesh = nullptr;
                VEMaterial* pMaterial = nullptr;

        //get mesh index in global mesh list
                uint32_t paiMeshIdx = node->mMeshes[i];
                //use index to get pointer to VEMesh
                pMesh = meshes[paiMeshIdx];
                //also get handle to the Assimp mesh
                aiMesh* paiMesh = pScene->mMeshes[paiMeshIdx];
        //get the material index for this mesh
```

```
                uint32_t paiMatIdx = paiMesh->mMaterialIndex;
                //use the index to get the right VEMaterial
                pMaterial = materials[paiMatIdx];

                glm::mat4* pMatrix = (glm::mat4*)&node->mTransformation;

                std::string leafName = entityName +
                    "_Leaf" + std::to_string(leafModelCounter++);

                VEEntity* pEnt = nullptr;
                VECHECKPOINTER(pEnt =
                    getSceneManagerPointer()->createEntity(leafName,
                    VEEntity::VE_ENTITY_TYPE_NORMAL_WITH_ALPHA,
                    pMesh, pMaterial, parent, *pMatrix));
                leafEntityNames.push_back(leafName);
        }

    //recursivly go down the node tree
        for (uint32_t i = 0; i < node->mNumChildren; i++) {
                loadLeafs(pScene, meshes, materials, node->mChildren[i], entityName, parent)
        }
}
```

B.10 NatureEntityBillboards

```
void NatureEntityBillboard::createEntity(VESceneNode* parent)
{
        std::vector<vh::vhVertex> vertices;
        std::vector<uint32_t> indices;
        NatureEntity_t* billboardInfo = getModelInfo();

        vh::vhVertex billboard_vertice;
        billboard_vertice.pos = glm::vec3(0.0f,0.0f,0.0f);
        uint32_t index = 0;

        vertices.push_back(billboard_vertice);
        indices.push_back(index);

        std::string entityName = getEntityName();

        VEMesh* billmesh;
        VECHECKPOINTER(billmesh = getSceneManagerPointer()->createMesh(
            entityName + "_Mesh", vertices, indices));

        //Material with texture for billboard
        VEMaterial* billMaterial;
        VECHECKPOINTER(billMaterial =
            getSceneManagerPointer()->createMaterial(
            entityName + "_Material"));
        billMaterial->mapDiffuse = getSceneManagerPointer()->createTexture
            (entityName + "_Texture", billboardInfo->baseDirectory,
            billboardInfo->modelFileName);

        //getRenderer()->removeEntityFromSubrenderers(x);
        //Extend Entity Type enum for further subrenderes
        //getRenderer()->addEntityToSubrenderer(x);
        VEEntity* entity;
        VECHECKPOINTER(entity =
            getSceneManagerPointer()->createEntity(entityName,
            VEEntity::veEntityType::VE_ENTITY_TYPE_BILLBOARD,
            billmesh, billMaterial,
            parent));

}
```

B.11 Outdoor Editor

```
void OutdoorEditor::handleInput(const glm::vec3& hitPos,
    const glm::vec3& direction, bool invertOperation)
{
        switch (activeMode)
        {
        case oeEditingModes::TERRAIN_EDITING_VOLUME:
                entityManager->removeEntitiesAt(hitPos,
                    activeBrush->getRadius());
                modifyTerrainVolumeWithActiveBrush(hitPos, invertOperation);
                break;
        case oeEditingModes::OBJECT_PLACEMENT:
                if (invertOperation) {
                        entityManager->removeEntitiesAt
                            (hitPos, activeBrush->getRadius());
                        break;
                }
                entityManager->addNatureEntity(selectedModel, hitPos, direction);
                break;
        case oeEditingModes::TERRAIN_EDITING_TEXTURE:
                changeTerrainMaterial(hitPos);
                break;
        default:
                std::cout << "Warning: Unknown Editing Mode" << std::endl;
                break;
        }
}

void OutdoorEditor::modifyTerrainVolumeWithActiveBrush(const glm::vec3& hitPos,
    bool subtractVolume)
{
        float strength = subtractVolume ? 0.0f : 1.0f;
        activeBrush->setStrength(strength);

        auto affected = activeBrush->getAffected(hitPos);

        for (const auto& a : affected) {
                VoxelPoint oldVoxel = voxelManager->getVoxel(a.first);

                if (!subtractVolume && (oldVoxel.density < a.second)) {
                        VoxelPoint voxel = VoxelPoint(a.second, selectedMaterial);
                        voxelManager->setVoxel(a.first, voxel);
                }
                if (subtractVolume && (a.second < oldVoxel.density)) {
                        VoxelPoint voxel = VoxelPoint(a.second, selectedMaterial);
```

```
                        voxelManager->setVoxel(a.first, voxel);
                }
        }
        refresh();
}

void OutdoorEditor::modifyTerrainVolumeWithActiveBrush(const glm::vec3& hitPos,
    bool subtractVolume)
{
        float strength = subtractVolume ? 0.0f : 1.0f;
        activeBrush->setStrength(strength);

        auto affected = activeBrush->getAffected(hitPos);

        for (const auto& a : affected) {
                VoxelPoint oldVoxel = voxelManager->getVoxel(a.first);

                if (!subtractVolume && (oldVoxel.density < a.second)) {
                        VoxelPoint voxel = VoxelPoint(a.second, selectedMaterial);
                        voxelManager->setVoxel(a.first, voxel);
                }
                if (subtractVolume && (a.second < oldVoxel.density)) {
                        VoxelPoint voxel = VoxelPoint(a.second, selectedMaterial);
                        voxelManager->setVoxel(a.first, voxel);
                }
        }
        refresh();
}
```

B.12 save and load Outdoor Editor

```
void OutdoorEditor::save(const std::string& path,
    const std::string& filename) const
{
        //Open File
        std::ofstream ofs(path + "/" + filename + ".json", std::ios::out);
        //JSON Serializer
        nlohmann::json serialize;
        nlohmann::json voxelData;
        nlohmann::json entityData;

        auto timeNow = vh::vhTimeNow();

        std::cout << "Saving in Progress..." << std::endl;
        std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
            << "]" << std::endl;
        //Write VoxelData
        serialize["VoxelData"] = voxelManager->save(voxelData);
        std::cout << "Voxel Data Saved" << std::endl;
        std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
            << "]" << std::endl;

        //Write Models
        serialize["EntityData"] = entityManager->save(entityData);
        std::cout << "Nature Entities Saved" << std::endl;
        std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
            << "]" << std::endl;

        //Write JSON Data to File
        ofs << std::setw(2) << serialize;

        //Close File
        ofs.close();
        std::cout << "Save Finished "<< std::endl;
        std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
            << "]" << std::endl;
}

void OutdoorEditor::load(const std::string& path,
    const std::string& filename)
{

        auto timeNow = vh::vhTimeNow();
        std::cout << "Loading in Progress..." << std::endl;
        std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
```

```
        << "]" << std::endl;
    std::cout << "Import File..." << std::endl;
    //Import Data from file
    std::ifstream file(path + "/" + filename + ".json", std::ios::in);
    nlohmann::json deserialize;
    file >> deserialize;
    file.close();

    std::cout << "File Data Imported" << std::endl;
    std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
        << "]" << std::endl;
    std::cout << "Update Voxels..." << std::endl;

    //Update with Imported VoxelData
    nlohmann::json voxelData;
    voxelData = deserialize["VoxelData"];
    voxelManager->load(voxelData);
    terrainManager->clear();


    std::cout << "Voxels Updated" << std::endl;
    std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
        << "]" << std::endl;
    std::cout << "Update Models..." << std::endl;


    //Update with Imported Entities
    nlohmann::json entityData;
    entityData = deserialize["EntityData"];
    entityManager->load(entityData);


    std::cout << "Models Updated" << std::endl;
    std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
        << "]" << std::endl;
    std::cout << "Refresh Terrain" << std::endl;

    refresh();

    std::cout << "Loading Finished " << std::endl;
    std::cout << "Time Duration[" << vh::vhTimeDuration(timeNow)
        << "]" << std::endl;
}
```