

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Eduard Hopper

Smooth Minecraft terrain rendering

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Tomáš Iser

Study programme: Computer Science

Study branch: IPP6

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date
Author's signature

I would like to thank my friends for always keeping food delivery prices low, and distracting me just the right amount, so that I could still finish this thesis on time. I would also like to thank my family for their continued support throughout my studies. Last but not least, I thank my supervisor, Tomáš Iser, for being cool and providing valuable feedback.

I dedicate this thesis to all people burning with passion for creating things.

Title: Smooth Minecraft terrain rendering

Author: Eduard Hopfer

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Tomáš Iser, Department of Software and Computer Science Education

Abstract: Interesting terrain is the backbone of any virtual world. To make features like caves, overhangs, and arches possible, we need to capture the internal structure of the terrain. In other words, we need a volumetric terrain model. Minecraft is known for its rich, procedurally generated worlds. Internally, these worlds use a volumetric representation - a 3D grid of voxels. While these blocky worlds look impressive already, continuous terrain is often more desirable. The goal of this thesis is to create a tool that can visualize Minecraft worlds as smooth surfaces. We explore ways of using Minecraft as a source of volumetric terrain data. The smooth scene is created by extracting a field function and polygonizing the implicit surface it defines. The user can adjust the appearance of the smooth scene through several parameters. The application also supports the exploration of the infinite Minecraft worlds in real-time.

Keywords: terrain voxel implicit surface Minecraft rendering

Contents

Introduction	3
1 Analysis	6
1.1 Related work	6
1.1.1 Terrain representation	6
1.2 Smoothing the voxels	8
1.3 Implicit surface rendering	8
1.3.1 Ray tracing	9
1.3.2 Mesh extraction	10
2 Minecraft as a data source	12
2.1 How are minecraft worlds generated	12
2.1.1 Noise	14
2.1.2 Terrain	14
2.1.3 Arches and overhangs	15
2.1.4 Caves	15
2.1.5 Biomes	15
3 Our approach	18
3.1 Extracting the smooth surface	18
3.2 Creating the mesh	18
3.2.1 Sharing vertices	19
3.2.2 Avoiding duplicate vertices	19
3.3 The floating tree problem	19
3.3.1 Smooth minimum	19
3.4 Shading	20
3.4.1 Normals	22
3.4.2 Materials	23
3.4.3 Textures	23
3.4.4 Lighting	24
3.5 Controlling the smoothing	25
3.5.1 Smoothing parameters	25
3.5.2 Multiple smoothing passes	25
3.5.3 Rigid structures	26
4 Implementation	27
4.1 Technologies	27
4.2 Renderer architecture	27
4.2.1 Render loop	27
4.2.2 User input	27
4.2.3 Components	28
4.2.4 Draw call abstraction	28
4.3 Model	28
4.3.1 Smooth chunk loading	29
4.3.2 Discrete scene	29

4.3.3	Smooth scene	29
4.4	Pixel art textures	30
4.5	Controls	30
5	Performance	32
5.1	Caching and distribution of work	32
5.2	Discrete scene	32
5.3	Smooth scene	32
5.4	Scene update	34
5.5	Memory	34
	Conclusion	35
	Bibliography	36
	List of Figures	37
A	Attachments	38
A.1	User guide	38
A.1.1	Controls	38
A.2	Material types	38

Introduction

Nice-looking terrain is a desirable feature of many 3D games and renders. There are many ways to approach creating and rendering terrain (section 1.1). One of the simplest is using a **heightmap** - a 2D texture where each pixel value represents the height of the terrain at that point. A flat grid of vertices is then rendered with each vertex having its height offset based on the value of the heightmap. A noise-generated heightmap already produces interesting-looking and detailed terrain. We are interested in rendering terrain with overhangs and caves, however, which is not possible with this approach.

To represent interesting features like this, we need a **volumetric** representation of the terrain. Manually creating volumetric data is cumbersome and generating data that can be used for terrain rendering is a complex problem. In this thesis, we avoid generating volumetric data entirely and focus solely on the rendering part.

We obtain our data from Minecraft the video game. Minecraft is set in an infinite, procedurally generated voxel world. The worlds generated by Minecraft have infinite variations and contain interesting features like caves, overhangs, ravines, mountains, and lakes.

Minecraft worlds will serve as perfect input data for our purposes. We explore ways of extracting a smooth surface from discrete volumetric data and implement a solution using **implicit surfaces**.

Our solution is able to support the infinite worlds of Minecraft by loading the necessary parts on demand. We are able to render both the original blocky version of the world as well as a smoothed-out version that is fully textured and shaded. While in the discrete version, every voxel has exactly one material assigned to it, the smooth version blends materials together, allowing for smooth transitions between them.

Additionally, we implement a way to keep some parts of the world as they are, since making them smooth wouldn't make sense. This is useful for trees and buildings for example. Our approach allows us to make the terrain smooth while keeping these objects intact and having them connect seamlessly to the smooth terrain.

This thesis proposes a read-only solution, meaning custom terrain generation or terrain editing are not in the scope of our thesis. We also do not aim for supporting the enormous amount of different blocks that Minecraft defines and limit ourselves to only a small subset. The final product is a graphical application that works with data generated by Minecraft and renders it as a smooth surface. The user can then explore this smooth world interactively in real-time.

Figures 1 and 2 showcase the terrain rendered by the final product.

Text structure

First, we examine all of the aspects of rendering smooth terrain while listing different existing approaches. In the second chapter, we explore how our data source - Minecraft - generates terrain and how we can use and control the data. Next, we cover in detail how we managed to extract and render the smooth

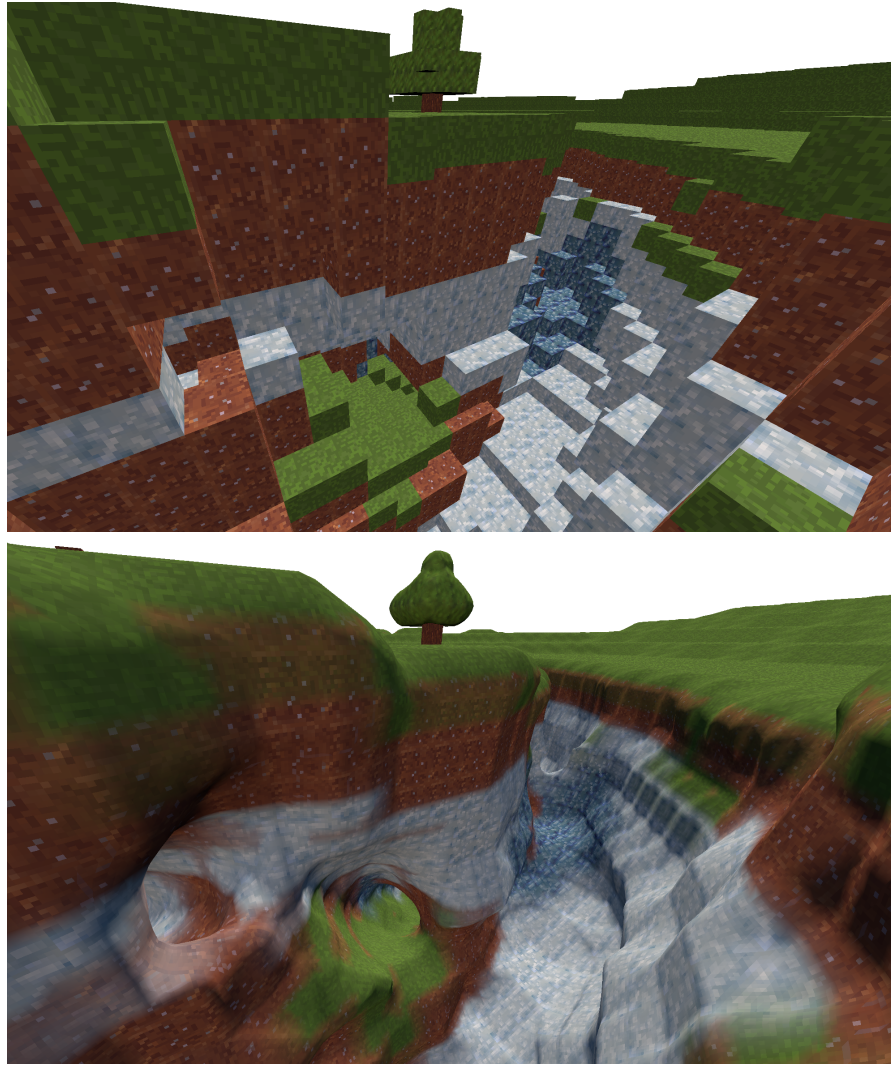


Figure 1: Example of terrain smoothed by our application.

implicit surface. After that, we provide a broad overview of how we implemented our solution and analyze the performance of the implementation. Lastly, we conclude by assessing the qualities and shortcomings of our solution and discuss future improvements.



Figure 2: Example of smooth terrain in combination with rigid blocks.

1. Analysis

1.1 Related work

There are many approaches when it comes to rendering smooth terrain [Galin et al., 2019]. We will now examine existing methods to find techniques applicable to our problem.

1.1.1 Terrain representation

How we represent the data that form a terrain is crucial. No representation is perfect and each has its uses and drawbacks.

Height based models

In terrain modeling, height-based representations are the most common. The terrain is represented as an elevation function $h(p) : \mathbb{R}^2 \rightarrow \mathbb{R}$, which maps points in the XZ plane to their heights. The terrain then consists of points in the form $(x, h(x, z), z)$.

The values of the function h can be generated procedurally or sampled from a closed-form expression. This representation has infinite precision and is memory efficient, but creating functions that generate realistic-looking terrain is difficult. Additionally, sampling the function can be computationally expensive.

The most common approach is to use a discrete **heightmap** (see figure 1.1). This is a 2D texture where the value of the pixel at coordinates (X, Z) is equal to the height Y. This approach is widely used for its simplicity, but it has its downsides. In contrast to procedural functions, it requires storage quadratic in world size. Due to its discrete nature, it is also inherently limited in resolution. The memory footprint can be reduced by quantization but this can lead to banding artifacts. Representing steep slopes is also problematic as the slope will always be undersampled.

Another representation, common in methods using erosion simulations is the **layered representation**. It consists of a set of elevation functions ($h_1, h_2, \dots$) where each corresponds to one material or layer of the model. The layers are usually in set order and the elevation functions return the thickness of the layer instead of its Y offset. The sum of the thicknesses of the layers is then the resulting height.

Height-based models are easier to work with, but they cannot represent the internal structure of the terrain. For terrain with caves and overhangs, a volumetric model is needed.

Volumetric models

Volumetric models are defined as a material function

$$\mu : \mathbb{R}^3 \rightarrow M$$

which maps every point in 3D space to a material index - the material present at that point.

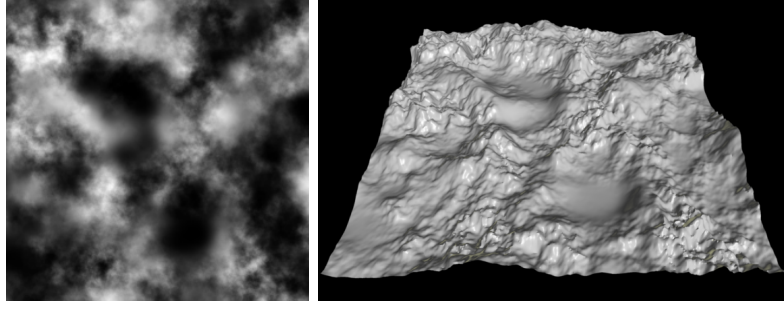


Figure 1.1: An example of a heightmap texture along with the terrain rendered from it.

Parallels exist with the height-based models, where some representations can be extended to work in 3D. Volumetric terrain can be represented as an **implicit surface** extracted from a field function f . The field functions $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ should be at least C^0 , but higher levels of continuity are preferred. Authoring implicit surfaces is complicated as is reconstructing the surface as a mesh that can be rendered. Additionally, as it was in the 2D case, evaluating the field function can be costly. For these reasons, implicit surface representations are often not preferred.

Voxels are an extension of the 2D heightmap to 3D. Every cell in the 3D grid contains information about the material present at that point in space. This allows us to represent complex geometry at the cost of memory, as we need $O(n^3)$ space where n is the size of the world. The material function can be evaluated efficiently in contrast to the function representation.

The memory footprint can be reduced by using optimized data structures such as **sparse voxel octrees**. This data structure partitions the voxel grid in a hierarchical manner and only subdivides regions if needed. This allows us to spend way less memory on regions with homogenous geometry, which was otherwise wasted. Another disadvantage of voxels is their discrete nature, which causes issues when trying to represent gentle slopes for example.

The layered representations described earlier can be used as volumetric representations directly if we allow inserting a layer of air between 2 solid layers.

Feature models

Work has been done to create a representation that is more abstract and can be used to model terrains with more control. The paper by Hnaidi et al. [2010] proposes a method of authoring complex terrains using feature curves represented as bezier splines. It then uses a diffusion method with added noise for details and procedurally generated textures to synthesize the terrain.

The technique described by G  nevaux et al. [2013] uses a sketch of a river network as input and generates terrain as if the river carved it.

Hybrid models

Some models use a combination of methods to represent the terrain. The approach described by Peytavie et al. [2009] is particularly interesting and is what our approach is based on. The terrain is stored in a layered voxel representation

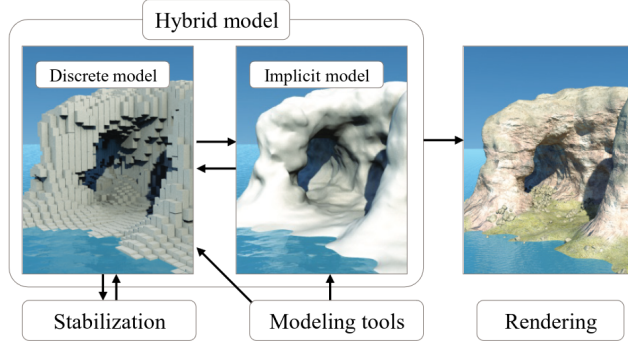


Figure 1.2: Overview of the hybrid model used by the technique described in Peytavie et al. [2009]

that allows layers of air, so it can be used to represent the internal structure of the terrain. This discrete model is used for sculpting the bedrock and large-scale features. A smooth implicit surface is then extracted from the discrete data through a convolution-based smoothing process. Details are then fine-tuned in this continuous representation.

1.2 Smoothing the voxels

We are using Minecraft to procure our terrain data, so we will be using a standard voxel representation. We want to render the data as a smooth surface, so we need a way to extract a smooth surface from the discrete data.

The method known as **Surface Nets**, first proposed by Gibson [1998] uses an iterative approach of smoothing discrete volumetric data. It works by assuming vertices have "gravity" and pull other vertices towards them. We can then examine the local or global neighborhood of each vertex, calculate the force applied to it and move it in that direction. This approach is nice, but it doesn't represent the smooth surface in any way as it just deforms the initial surface. We need to be able to do operations on the smooth surface to accomplish our goals.

Another approach is described in Peytavie et al. [2009] and is what our approach is based on. It is based on extracting a field function by using convolution over discrete data. The implicit surface defined by this function is then used as the smooth terrain surface.

1.3 Implicit surface rendering

First, let us define the implicit surface.

Definition 1. Let f be a function of 3 variables $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. The set of zeros

$$S = \{x \in \mathbb{R}^3 \mid f(x) = 0\}$$

is then called the implicit surface of f .

There are 2 major approaches when it comes to rendering implicit surfaces. we can either extract a mesh and render it using the traditional pipeline, or we can use ray tracing to render it directly.

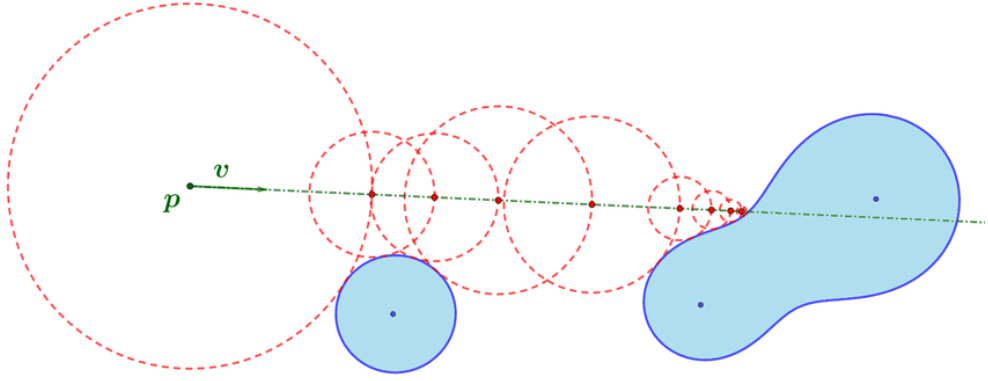


Figure 1.3: Visualization of the ray marching algorithm [Bálint et al., 2019]

1.3.1 Ray tracing

Ray tracing is the simpler of the 2 options. It involves shooting a bunch of rays into the scene and tracing their paths as they bounce off the surface. This model represents more closely how light would bounce in the real world. Ray tracing produces the highest quality images with the caveat that it is very computationally demanding. Real-time ray tracing is currently only possible on select high-end graphics cards.

A variation of ray tracing called **ray marching** is particularly interesting in the context of implicits [Hart, 1995, Liktó, 2008]. It requires the implicit surface to be a **signed distance field**.

Definition 2. Let $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. For each point f returns the **distance** to the surface of the scene, with points inside the scene resulting in **negative** distances. Then such a function is called a **signed distance function** or **sdf**.

Definition 3. Given an sdf f , $\alpha \in \mathbb{R}$ and its implicit surface defined as

$$S = \{x \in \mathbb{R}^3 \mid f(x) - \alpha = 0\}$$

We call this implicit surface the α -isosurface of f .

An isosurface is a set of points with equal distances to the surface, also known as a level set. Usually, we care about the 0-isosurface, but we can use the higher isosurfaces to effectively get a scaled version of our scene.

Since we always know the distance to the scene, we can safely advance the ray by this distance and be sure we won't intersect anything. An intersection is found when the step size (the distance) gets too small. This makes effects like soft shadows or ambient occlusion free since information about the surrounding geometry can be extracted just from the step sizes the ray took along its march. We can see an example of this in 1.3. The step size gets small in the middle of tracing the ray; this means the ray passes closely to the geometry and we can use this information to cast a soft shadow when the ray finally intersects the scene.

This Method is very elegant and works well, but we wanted to use a mesh extraction approach for this thesis.

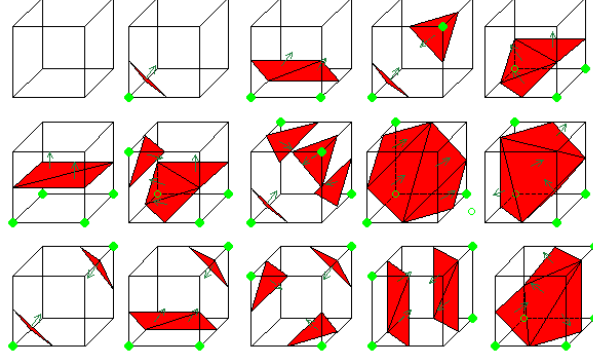


Figure 1.4: Marching cubes - the 15 unique cases [Cirne and Pedrini, 2013]. The highlighted vertices represent points within the volume.

1.3.2 Mesh extraction

The implicit surface can represent arbitrary geometry, so extracting a good polygonization is not an easy task. Moreover, since the mesh has limited geometry, it will always be a more crude approximation of the surface than we would get with ray tracing.

Marching cubes

Probably the simplest and most common method of mesh extraction is Marching cubes. The original paper dates back to 1987 - Lorensen and Cline [1987].

It works by marching a cube through the volume in a uniform grid. At every point, we sample the implicit function for each cube vertex. Each vertex can be either inside the volume or outside. The state of each vertex is used to create an index into a lookup table. We consult the lookup table to see which cube edges intersect the surface and should therefore contain a vertex of the resulting mesh. Another lookup table is used to see how to connect these vertices into triangles of the resulting mesh.

Remark. There are 2^8 possible cases for each cube - each vertex is either inside the volume or outside. Only 15 cases are unique, however, the rest are just symmetries. See 1.4 for insight on how they look like.

This is the method we decided to use for the purposes of this thesis. We chose it for its simplicity and relatively good quality of the resulting mesh.

Adaptive Marching triangles

Marching cubes work well, but it has its problems. Since we always sample in a uniform grid, we end up wasting triangles in flat regions, and get artifacts or deformed triangles in regions of high curvature. The method proposed in Rodrigues de Araújo and Armando Pires Jorge [2005] tries to remedy this by considering the hessian of the implicit function and adjusting the size of the triangles based on local curvature. It works by expanding the mesh from a single point, evaluating the curvature at each point, and choosing an appropriate triangle size and count. This results in a much better polygonization of the surface as can be seen in figure 1.5.

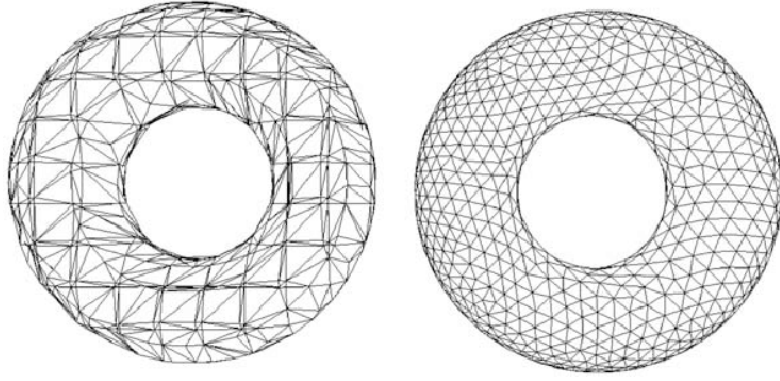


Figure 1.5: Comparison of meshes of a torus. The image on the left was created using a variation of marching cubes called marching tetrahedra, while the right mesh was created by the adaptive marching triangle algorithm [Rodrigues de Araújo and Armando Pires Jorge, 2005].

This is the method we initially wanted to use, but our method of extracting the implicit function produces a function of only class C^0 . Marching triangles requires the evaluation of the **hessian**. Since even the 1st-degree derivatives are not continuous, evaluating the 2nd-degree did not produce any meaningful values and the resulting hessian was unusable. For this reason, we are limited to polygonization techniques that do not require higher degrees of continuity.

2. Minecraft as a data source

Generating volumetric data is out of the scope of this thesis. Instead, we chose to use data generated by Minecraft.

Minecraft generates worlds with mountains, ravines, caves, and overhangs so we can easily find interesting features to test our renderer on. We can even manually edit details in-game or use a third-party tool to have more control over the world generation. Minecraft is extremely popular, so there is a ton of support for these kinds of things.

Remark. Voxels are called **blocks** in Minecraft, and we will use these terms interchangeably from now on.

To support an infinite world, Minecraft partitions it into **regions**. A new region is procedurally generated when you first enter it based on a seed value that is constant for the world. After it was generated once, a region is saved into its own file on disk. When entered at a later time, it is loaded from the disk, instead of being generated again. Regions partition the world in the XZ plane and they are areas of 512x512 blocks or 32x32 **chunks**. A chunk is a 16-block wide, 16-block long, 384 blocks high segment of the world.

When the player moves around, chunks are loaded and unloaded around them to create the illusion of an infinite world. Our implementation works this way too.

In Minecraft, each block represents 1 cubic meter of some material (dirt, stone, emerald, ...).

Remark. There are special blocks that are not displayed as just cubes (torches, stairs) and even blocks that take up multiple voxels worth of space (doors, beds). We do not consider these special blocks in our implementation, since they are mostly not part of the terrain and each of them would require individual care to integrate into the continuous terrain.

Currently, there are over 800 types of blocks in Minecraft, and supporting them all wouldn't add much value to our work while being extremely labor-intensive. This is why we decided to support only a handful of them and create a many-to-one mapping from Minecraft blocks to our block types. While Minecraft uses many types of wood and ore, we coalesce all of them into a single type of wood and ore respectively. We do the same thing for other types of blocks as well to support as many block types as possible with relatively little effort. We also wanted to keep the feel of the Minecraft art style in our visualization. As such we use pixel art textures similar to those that Minecraft uses for our block types. All of the textures were created from scratch for this project in order to avoid any licensing issues. A full list of supported materials, along with the mapping from Minecraft blocks and the corresponding textures can be found in the appendix A.2.

2.1 How are minecraft worlds generated

Minecraft is probably the biggest commercial product with a focus on large-scale and diverse procedurally generated terrains. Since this is where we get our input

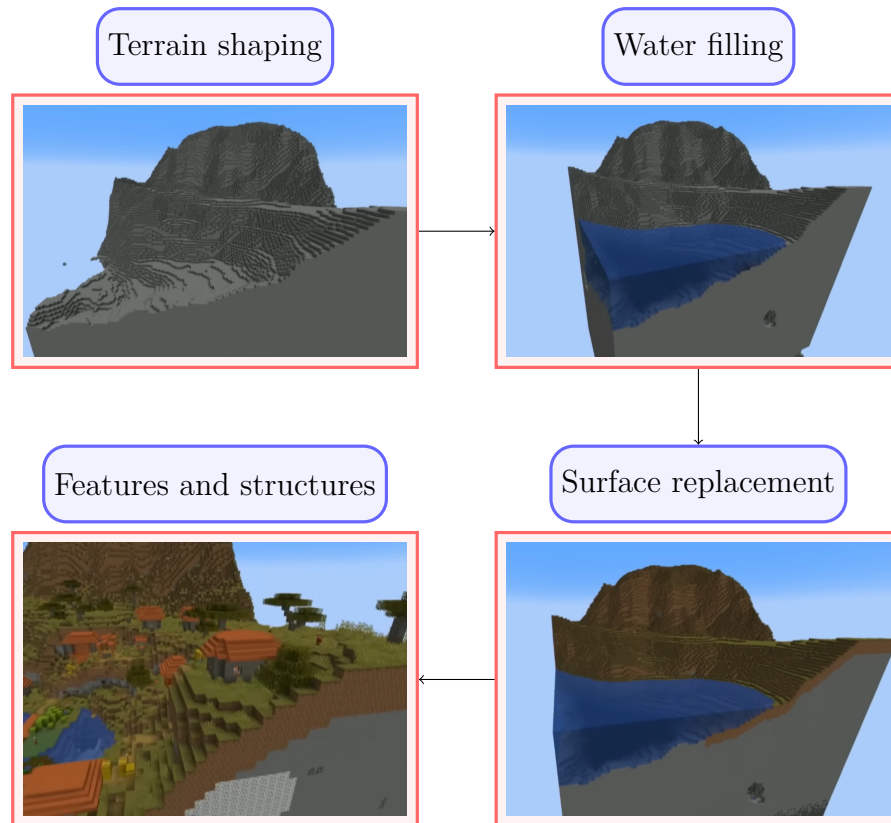


Figure 2.1: The Minecraft world generation pipeline [Kniberg, 2022]

data, it is important to understand how they are created. With this knowledge, we could even tweak the world generation to create data more suited for smoothing, which is our goal. Moreover, the techniques used in Minecraft are useful for terrain generation in general.

Minecraft uses a voxel terrain representation and the world is generated on demand per chunk [Kniberg, 2022]. The worlds are created in stages. Figure 2.1 showcases the pipeline of world creation. What happens in each stage is summarized below, in order:

Terrain shaping In this stage, the outline of the terrain is generated, complete with oceans, plains, mountain ranges, ravines, and caves. The output is a binary value for each voxel - is there a block or not?

Water filling Next, a water level is set and all parts of the world with a lower y value are submerged in water.

Surface replacement This is the stage where blocks are assigned different materials, based on the biome a chunk falls into. This is essentially painting the skeleton terrain with different colors.

Features and structures Lastly, features and structures are generated. This includes trees and villages as well as patches of different ores.

For our purposes, we are mostly interested in the **terrain shaping** stage. This stage is further subdivided. First, a heightmap representable version of the

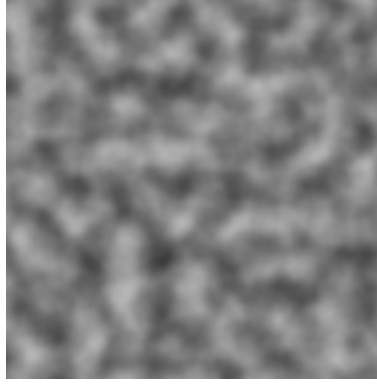


Figure 2.2: Example texture generated by 2D perlin noise. The color of each pixel represents the noise value.

terrain is created. Then overhangs and arches are added. Lastly, space is carved into the terrain to create caves.

2.1.1 Noise

To generate terrain that is not repetitive some randomness is required. A purely random distribution of blocks doesn't create anything that could pass as terrain, however. We need a kind of randomness that produces similar values in a local area. Minecraft uses 2D and 3D Perlin noise to generate this randomness as it satisfies these properties.

The terrain shaping algorithm cleverly combines multiple layers of noise to create realistic-looking terrain.

2.1.2 Terrain

First, a terrain without any volumetric features is generated. This is done by layering 3 different noise values, each of which influences the terrain height differently. The shape of each of the noise layers is different and the way they interact with each other and contribute to the final height of the terrain is nontrivial.

These values decide how the world is shaped and are named to suggest what they control:

Continentalness Controls whether a region contains an ocean, a shoreline, or is inland.

Erosion Controls the shape of mountains and the creation of ravines.

Peaks and valleys Controls the general height of a region.

At every point in space, we sample the respective noise values and use splines to map these values to terrain height. The shape of the splines decides how these noises influence world generation and changing them leads to wildly different worlds. Figure 2.3 illustrates the spline setup used in the base game.

Remark. Modifying the shape of the noise as well as the shape of the splines is the primary way of tweaking the world generation as it can be done without

affecting the code in any way. Note that this can and will give a completely different meaning to the noise layers.

2.1.3 Arches and overhangs

Next, arches and overhangs are generated by using 3D Perlin noise. Another 2 parameters are introduced to control the 3D noise: **squashing factor** for controlling the magnitude of the features and **height offset** to control at what height these features occur. Both of these parameters are derived from the values of the noise layers from the previous section. The way they are derived is integrated into the splines, also described in the previous section.

2.1.4 Caves

Caves are also generated using 3D noise. A noise with a low frequency is used to create cave systems containing large caverns. A chosen constant value is used to differentiate regions of blocks and air (noise value above/below the constant). The pockets of air are then used as the interior of the cave.

Another noise is then used to create long, narrow tunnels. This time we place a cave in regions where the absolute noise value is below some constant. This will create caves along the transition between the positive and negative noise values and is known as noise ridging.

Combining these 2 types of caves creates large and interesting cave systems (figure 2.4).

2.1.5 Biomes

Biomes dictate what types of blocks are present in a given region. Minecraft supports many different biomes; meadows, deserts, and jungles are some of them. They are distributed by using even more noise. Two more noise layers are introduced: **temperature** and **humidity**. These are used in combination with the 3 original noises to create a 5D space that controls the placement of biomes.

Biomes are discrete, so they are configured using a set of tables instead of splines. Noise values are split into a finite set of ranges and each range corresponds to some row/column in a table. This makes the biome space a 5D grid. When designing the tables, it is important to think about neighboring values in the biome space to avoid placing a snowy biome right next to a desert for example. The biome tables are designed with respect to the terrain splines so that snowy biomes get generated mostly in regions with mountains, swamps in lowlands, and more. Figure 2.5 showcases how different noise values influence the shape of the terrain and biome.

Remark. Note that biomes are distributed in the *surface replacement* stage. In contrast, all of the other generations we discussed were done in the terrain shaping stage.

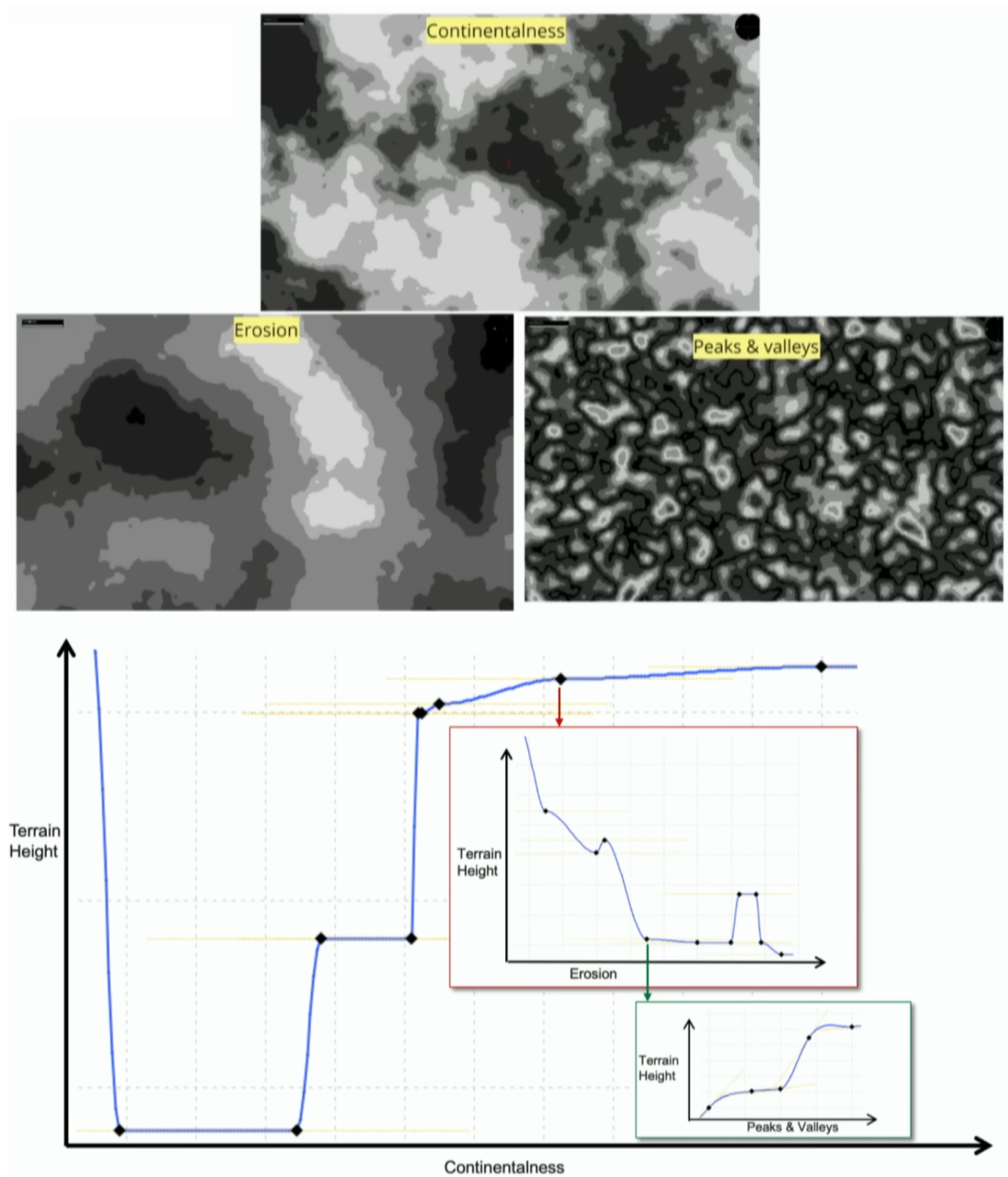


Figure 2.3: Example noise textures used for the different noise layers (above) and the splines that define how they shape the world when layered together (below) [Kniberg, 2022].

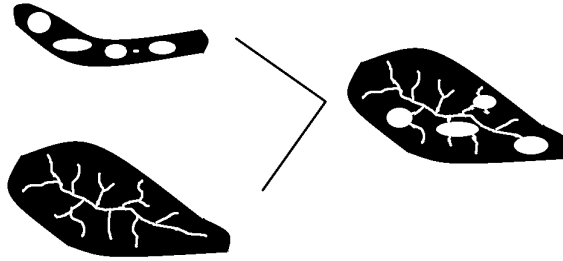


Figure 2.4: An illustration of how caves generated by different 3D noises are combined.



Figure 2.5: The above image showcases a mountain range created by a high peaks and valleys value combined with high erosion. The result is a mountain with steep slopes. Also present are overhanging features created by 3D noise. The image below shows a clear boundary between 2 biomes. In this case it was caused by the value of temperature reaching some threshold.

3. Our approach

3.1 Extracting the smooth surface

To view the volumetric data as a smooth surface we will represent it as an implicit surface. We use the same field function as proposed by Peytavie et al. [2009]. It is defined as a linear function involving a convolution function. Let M be a smoothable material, $\|q\|_\infty = \sup(|x|, |y|, |z|)$ and σ be the radius of the convolution kernel. Then

$$g(p) = \begin{cases} 1 & \text{if } p \in M \\ 0 & \text{otherwise} \end{cases} \quad h(q) = \begin{cases} 1 & \text{if } \|q\|_\infty < \sigma \\ 0 & \text{otherwise} \end{cases}$$

where g is the **skeleton** function and h is the convolution **kernel**. The convolution function i is then defined as follows

$$i(p) = (h \star g)(p) = \int_{R^3} g(q)h(p - q)dq.$$

Remark. The skeleton function g will in practice be used to match the union of all smoothable materials, instead of just a single material.

Since h is defined using the infinite distance function, the convolution kernel is a cube of radius σ . This gives us some insight into how the smoothing works. In 2D, applying a box convolution kernel to an image will produce a blurred version of the image. We are doing the same thing, just extended to 3D. By blurring the volume, we discard the high-frequency blockiness of the input data and keep only the low-frequency features which are the outlines of the terrain.

This simplifies the computation of the integral to the evaluation of the volume of material M inside the convolution kernel. Let V_M be the material volume inside the kernel and $V_\Omega = 8\sigma^3$ the volume of the kernel. Finally we get our field function

$$f(p) = 2\frac{V_M}{V_\Omega} - 1 = \frac{i(p)}{4\sigma^3} - 1.$$

This makes f essentially a material density function, transformed to the range $\langle -1, 1 \rangle$. Value 1 means the whole kernel is filled with material, and -1 means there is no material at all. We are interested in points with a density value of 0. The neighborhood of these points is half filled with material and half empty, meaning they are likely on the surface of the volume we want to render. This translates directly to the definition of the implicit surface $S = \{p \in R | f(p) = 0\}$.

Remark. The function f is a linear function of a convolution of piecewise constant functions. This makes f continuous, but only of class C^0 .

3.2 Creating the mesh

We are using the marching cubes algorithm to polygonize the implicit surface. That said, we make a few adjustments to the algorithm to better suit our needs.

When evaluating the case and corresponding triangulation for each cube separately, we end up doing more work than needed.

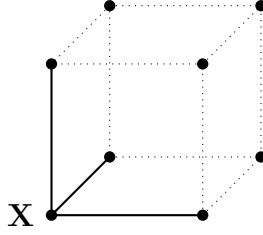


Figure 3.1: For every vertex X of the grid, only the thick edges are evaluated.

3.2.1 Sharing vertices

We need to evaluate the implicit function f for each cube vertex. Since the cubes share vertices, we evaluate the function for the same point multiple times. This is a problem because evaluating f is an expensive operation and we want to minimize the amount of evaluations. We can solve this by evaluating f for each vertex of the grid up front, and looking up the cached values for each cube vertex when needed.

3.2.2 Avoiding duplicate vertices

Iterating the grid per cube also poses another problem - duplicate vertices in the resulting mesh. This is again caused by the fact that cubes share vertices. We can solve it by having each cube evaluate only a part of its edges. We choose these edges such that in the end all of the cube edges will be evaluated, some indirectly by the neighboring cubes. Figure 3.1 shows the edges each cube evaluates. It can be easily seen that if we evaluate these edges for every grid cell, we will have evaluated all of the edges exactly once. Note that on the edges of the grid, we will be evaluating edges that are not within the grid with this schema. This is not a big problem, as we can just check that we are within the grid before evaluating the edge.

Remark. Restructuring the algorithm like this makes evaluating each cube an independent operation. This allows us to trivially parallelize the construction of the mesh, gaining a lot of performance.

3.3 The floating tree problem

Our input data contains blocks that should not be smoothed like trees and buildings. Rendering these structures as they are creates situations where blocks are floating in the air, even when they have a block under them in the discrete model. The problem is most obvious on cliff edges, where smoothing out the jagged edge moves the cliff down and away a little, leaving the tree standing on nothing.

3.3.1 Smooth minimum

We can use signed distance fields (or sdfs for short) to remedy this problem. Let f, g be sdfs. Then the union of the objects they represent can be defined as $h = \min(f, g)$. Doing this gets us the sdf of the combined objects, but the transition between them will be hard and we want them to connect smoothly.

We want a function that is equal to plain *min* when the values are far from each other but smoothly transitions between the values when they get close. This is one possible implementation of a smooth minimum function [Quilez, 2013b]:

```
float smin( float a, float b, float k )
{
    float h = max( k-abs(a-b), 0.0 )/k;
    return min( a, b ) - h*h*k*(1.0/4.0);
}
```

Right now we are drawing the implicit surface as a mesh and the rigid blocks as cubes with hardcoded geometry. We can represent the rigid blocks as sdfs easily [Quilez, 2013a].

```
float sdBox( vec3 point, vec3 boxDimensions )
{
    // Mirror the point to the positive octant
    // and move the origin to the box vertex
    vec3 q = abs(point) - boxDimensions;

    float outsideDistance = length(max(q,0.0));
    float insideDistance = min(max(q.x,max(q.y,q.z)),0.0);

    return outsideDistance + insideDistance;
}
```

Our implicit function will need some adjustments for it to work as a sdf. First, we will flip its sign so it always returns positive values for low-density points and negative values for high-density points. We now interpret the negative density as the distance to the surface. For points close to the surface this interpretation gives us meaningful values. Points that are further than the kernel radius will always result in a distance of 1, which is not ideal, but is good enough for our purposes.

Remark. Having an ill-formed sdf like this is another reason to use a mesh extraction technique for rendering. Ray marching an sdf like this would be very slow since we can never take a step bigger than 1.

Now we can render the smooth minimum of the terrain and rigid blocks, resulting in much nicer transitions between the two. A comparison of the techniques can be seen in figure 3.2.

3.4 Shading

Shading implicit surfaces poses a number of challenges. In this section, we will discuss the steps we took to get a textured, shaded version of the implicit surface.

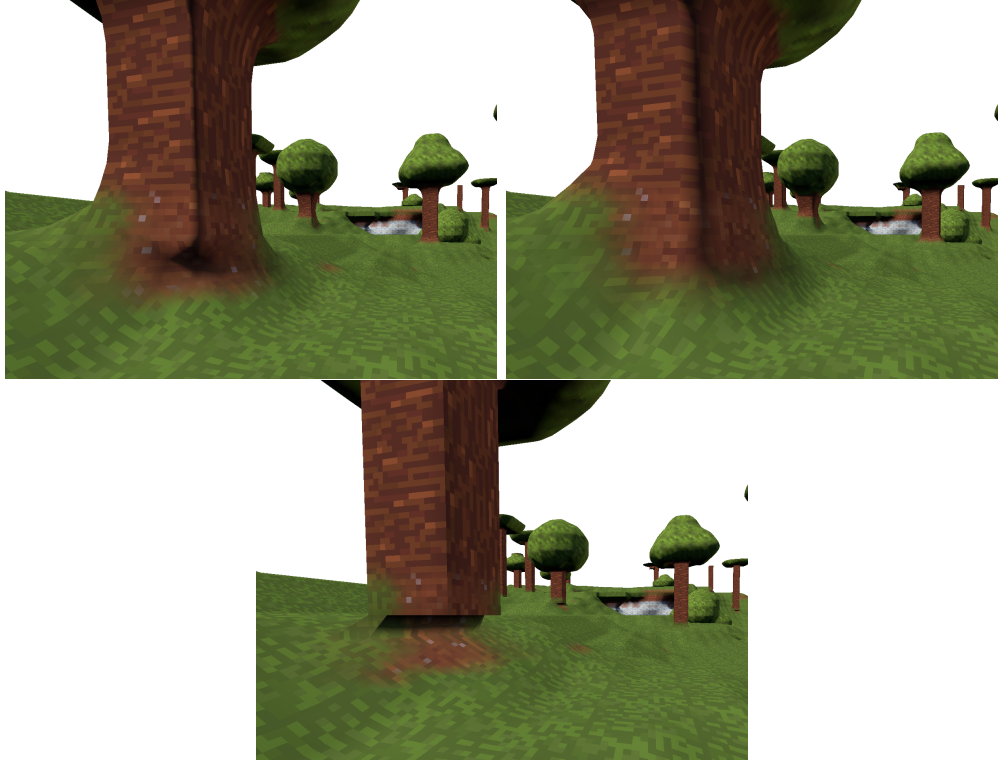


Figure 3.2: Demonstration of smooth minimum. In the bottom image, a plain min function is used to join the 2 sdfs. The top row showcases the smooth minimum function with different strenght of smoothing.

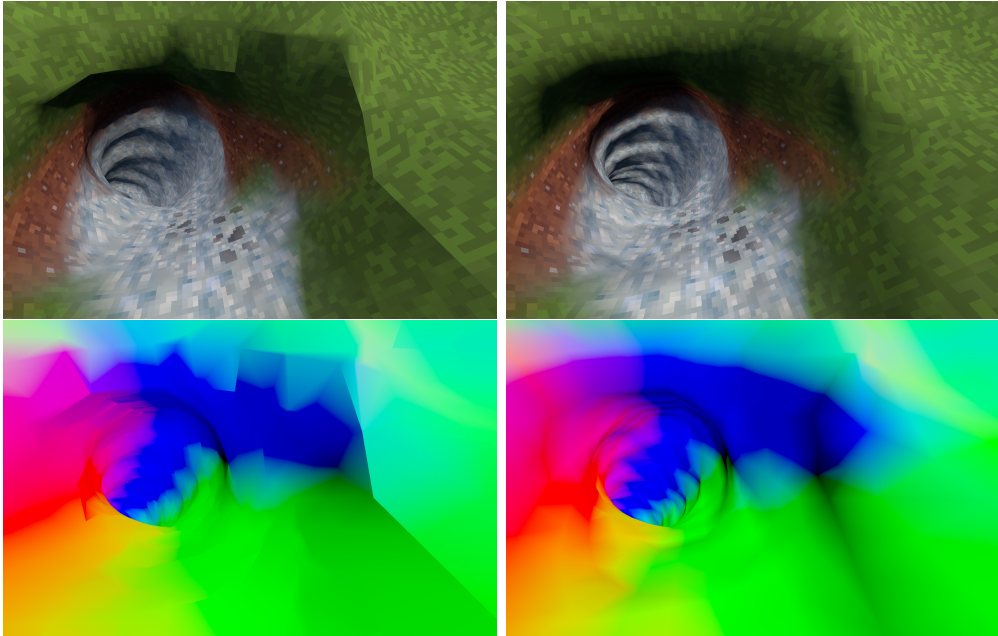


Figure 3.3: Comparison of triangle face normals (left) and gradient normals (right). The resolution of the mesh is the same in both cases. The bottom row shows a debug view with normal direction translated to a color value and better shows the discontinuities of the face normals.

3.4.1 Normals

To be able to do any shading at all, we need surface normals. In our case, we have 2 choices when it comes to obtaining them.

Since we have a mesh of the surface, we can use the normals of the individual triangles. We can get a normal for a triangle face as a cross product of 2 of the triangle's sides. Since triangles share vertices, we use the average of face normals of all triangles that share a vertex as that vertex's normal.

$$\triangle(A, B, C) \rightarrow \begin{array}{l} u = A - B \\ v = C - B \end{array} \rightarrow n = u \times v$$

Another option is to use the gradient of our implicit function. We already know our function behaves like an sdf for points close to the surface. Since we only need the normals for vertices of the mesh, which are by definition points (approximately) on the surface, this holds. Points further from the surface will have greater sdf values than points closer to it. So the gradient, or direction of greatest increase of the sdf is indeed a good normal vector for our surface.

We obtain the gradient using numerical methods. Recall that the definition of the gradient is

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^\top.$$

We approximate each of the partial derivatives using a numeric **central difference**

$$\frac{\partial f}{\partial x} = \lim_{h \rightarrow 0} \frac{f(x+h, y, z) - f(x, y, z)}{h} \approx \frac{f(x+\varepsilon, y, z) - f(x-\varepsilon, y, z)}{\varepsilon}$$

where $\varepsilon > 0$. This gives us a high quality surface normal that we can use in our renderer.

The gradient approach produces nicer results but is more costly to calculate since it requires evaluating the sdf 6 times. Figure 3.3 shows a comparison of the 2 approaches. Using a forward derivative $\left(\frac{f(x+\varepsilon) - f(x)}{\varepsilon} \right)$, we could get down to 4 evaluations, or even 3 if we assumed $f(p) = 0$ (the point p is on the surface). This version, however, proved to produce visual bugs so we stuck with the central derivative. As we discussed before, our implicit function is only C^0 , so there are discontinuities in the derivative. In practice, we didn't see any problems caused by these discontinuities when using the gradient normals.

It is worth noting that the evaluation of the gradient can fail in rare cases. Across different mesh resolutions and parametrizations of the smoothing process, a situation can happen where offsetting the evaluation point in any of the 3 major directions always yields the same value. In these cases, we are left with a 0 vector as the normal, which is not usable. It would be difficult to identify and remedy these degenerate situations at the source. Instead, we fall back to the triangle face normals in these cases. The degenerate situations are quite rare and only cover a few triangles when they happen so the use of a slightly different normal is not noticeable at all.

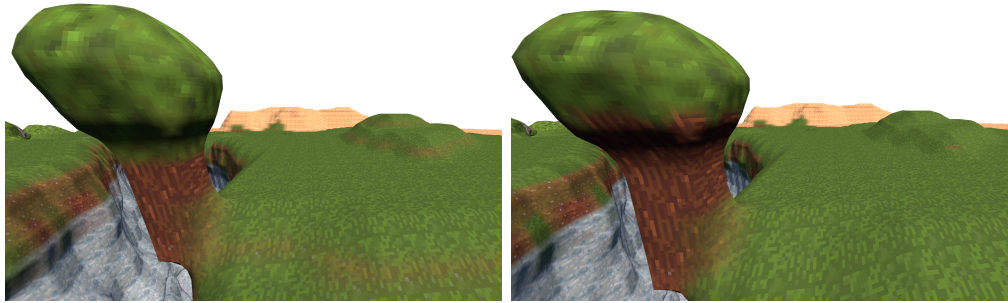


Figure 3.4: Demonstration of material blending. Left - equal blend, right - dirt and leaves suppressed.

3.4.2 Materials

Smoothing out the world removes boundaries between blocks and the different materials blend together. To get the **material blend** for a point, we have to evaluate the materials in its neighborhood in the discrete model. The blend is a set of materials with weights indicating how much of that material is present. The weights are normalized so they add up to 1, meaning we can think of the individual weights as percentages that make up a whole.

We calculate the material blend per vertex of the mesh and the weights are then interpolated to get a per-fragment blend.

Artistic control

We can also set an **activation threshold** for each material to control how they blend with other materials. This is a value between 0 and 1 indicating how much of a material must be present for it to appear in the blend. A threshold of 0 means the material is always active - if there is even a little of the material in the neighborhood, it will appear in the result. This creates smooth interpolated transitions between materials. Setting the threshold to a larger value will make the transitions more sudden, which can be appropriate for some materials.

3.4.3 Textures

Textures make it possible to create a detailed scene with relatively little geometry. In traditional rendering, texture coordinates are defined for every vertex of the mesh. They are then interpolated to get the coordinates for every fragment, which are then used to sample the texture.

Calculating texture coordinates for an arbitrary implicit surface is a challenging task. There are ways to avoid it like using procedurally generated textures or 3D textures.

We chose to go with the approach used in Peytavie et al. [2009]. It manages to map a 2D texture to the ground mesh in real-time, without computing and storing texture coordinates. Let p be a vertex of the ground mesh, n its normal, and T the 2D texture. We project the texture onto each of the world axes and blend the projections based on the normal vector.

$$\text{color}(p) = \alpha T(|x|, |y|) + \beta T(|x|, |z|) + \gamma T(|y|, |z|)$$

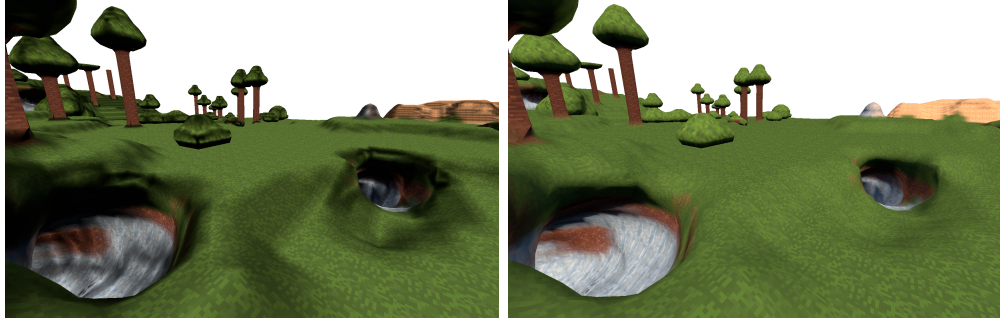


Figure 3.5: On the left we see texture mapping with variable normal length. The right image shows our version with constant normal lengths.

where the blending coefficients are computed as a function of the normal.

$$\omega = |n_x| + |n_y| + |n_z| \quad \rightarrow \quad \alpha = |n_z|/\omega \quad \beta = |n_y|/\omega \quad \gamma = |n_x|/\omega$$

The original paper uses the $\mathcal{L}^p$ norm of $\mathbf{n}$ instead of ω for a smoother blend between the textures. In practice, this only created artifacts and did not improve the blending in regions where it did work.

Normal vectors have length 1. We use the coordinates of the normal vector to drive the blending but the sum of the absolute values of its coordinates is not constant for all possible normals. This makes slopy regions appear darker than flat regions regardless of shading and is undesirable. Our version sacrifices the constant length of 1 for constant sums of absolute coordinates. Note that the vector direction is preserved. Now that the blending coefficients always add up to 1, there are no more dark spot artifacts. Figure 3.5 showcases this effect.

Remark. Using the value $p = 1$ in the original implementation is mathematically equal to our version with ω .

You may also notice that our α and γ coefficients are swapped in contrast to the original paper. α controls the XY plane, β the XZ plane and γ the YZ plane. It makes sense for each plane to be controlled by the axis perpendicular to it. Otherwise, we get banding artifacts in places where one of the projections dominates. If we used the x axis to control the XY plane for example, areas where the normal points into the x direction will look weird since the x coordinate never changes and we use it to sample the texture. This is what happens using the original implementation as can be seen in figure 3.6

The original paper also states that texture sampling is done per vertex. We find this dubious considering the quality of their renders. Nevertheless, we sample our textures per fragment with T being a blend of the individual material textures combined using the weights we calculated earlier.

3.4.4 Lighting

We used a gamma-corrected lighting setup for shading our scene. It is pretty simple and consists of 3 lights:

- A strong yellowish directional light representing **the sun**.
- A weak blueish directional light representing **the sky**.

- A weaker version of the sunlight coming from the exact opposite direction representing scattered **indirect** light.

3.5 Controlling the smoothing

Many variables influence how the final scene looks. No one configuration works best for all cases, so we allow the user to tweak them to find the best-looking result.

3.5.1 Smoothing parameters

There are 2 major variables that influence the shape of the terrain. First is the size of the convolution kernel. This controls the strength of the smoothing effect, with bigger kernels causing a stronger effect. Second, is the resolution of the surface mesh. Higher resolutions allow for more details to show, resulting in a nicer and smoother terrain. The resolution of the mesh has a large effect on the performance, however, so it is important to find a good balance between quality and speed.

We can also configure how are rigid blocks connected to the terrain. This can be done by adjusting the strength of the smooth minimum. The effects of tweaking this variable can be seen in figure 3.2.

As explained in section 3.4.2, we can control the texture mapping by adjusting the activation thresholds for each material.

3.5.2 Multiple smoothing passes

Tweaking these variables gives us a lot of freedom in configuring the terrain appearance already. In addition to these variables, we allow the smoothing to be done in multiple passes, each with a different configuration. With this we can do stuff things like making the terrain really smooth while smoothing the leaves of trees just a little so they keep their shape.

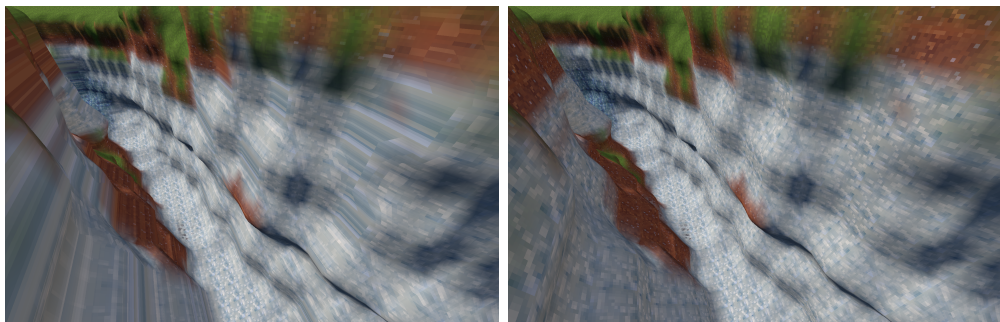


Figure 3.6: The left image uses textures mapped using the original order of coefficients and shows visible banding. The right image showcases textures mapped using orthogonal coefficients.

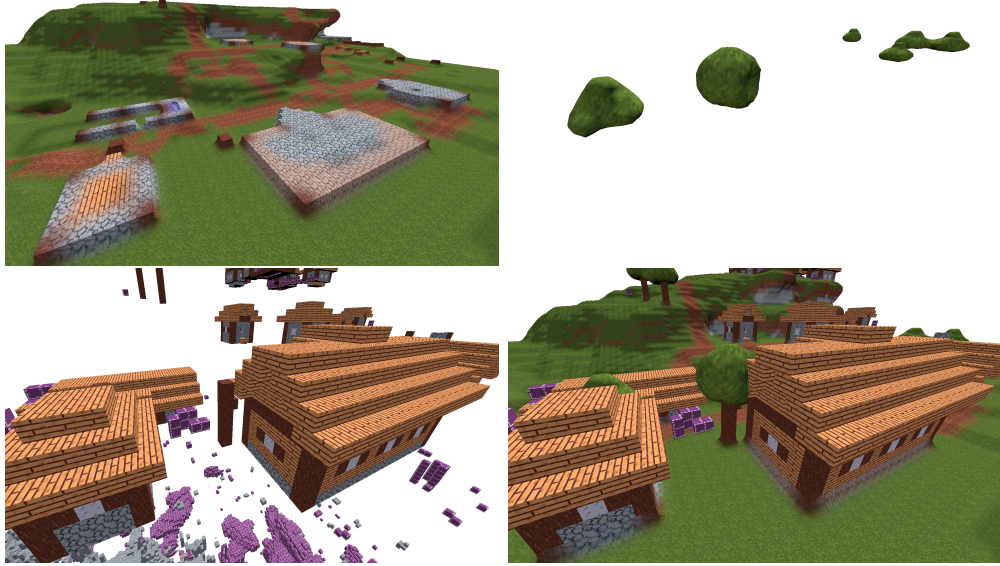


Figure 3.7: Parts of the smooth scene. The top row shows the smooth passes for the terrain and leaves. On bottom left we see the isolated noncritical rigid blocks. Finally, the bottom right image shows the final composited scene.

3.5.3 Rigid structures

Rendering rigid blocks as sdfs distorts them a little since they are pulled toward the surrounding terrain (figure 3.2, top right). This is what we want for joining the rigid structures to the terrain, but it starts to cause problems in larger rigid structures like houses. For this reason, we implemented a way to only smoothly join those rigid blocks that are incident with the terrain. The rest of the rigid blocks are then rendered as plain cubes.

In combination with multiple smoothing passes, this gives us great control over how the scene is constructed (see figure 3.7).

Remark. This works great for keeping houses intact but is distracting in natural settings. The houses are usually built from blocks that can be found in the world naturally (stone, wood), and seeing them in their rigid form in a cave doesn't look very nice. To solve this, we would need a more complex way of differentiating between blocks to see which were placed by the player/ are part of a building/ are a part of the terrain.

4. Implementation

4.1 Technologies

We implemented the renderer as a **Rust** program, using the **glium** crate for interfacing with OpenGL. This allowed us to write type-safe graphics code without having to manage the OpenGL state using magic macros. Glium provides rust bindings to the C OpenGL API, so we don't have to use unsafe rust at any point.

The application is also cross-platform, running on both Linux and Windows without any platform-specific code.

4.2 Renderer architecture

We implemented our renderer in a pretty standard way. It can handle user input and render 3D scenes with a dynamic camera angle. In this section, we will provide a high-level overview of the components of the renderer and how they interact with each other. More details can be found in the developer documentation and the source code.

4.2.1 Render loop

Creating a single frame involves the following actions:

1. Process user input
2. Update and lock in the render state for the frame
3. Update renderer components
4. Request scene geometry from the model
5. Render the scene

The render state consists of data that needs to persist across frames. We use this for keeping track of what scene we are rendering, frame timings, and more. The render state is recreated in every frame from the old state after considering user input events. This is the only point where we update the render state to avoid complexity.

The creation of the scene will be discussed in a later section 4.3.

In figure 4.1 we see a flowgraph of the render loop.

4.2.2 User input

We map the raw user input events (the W key was pressed) into our custom event type that represents some kind of action (move forward). Input events that are not mapped are ignored. This way our application only has to handle a known limited set of events and more can be added when needed.

4.2.3 Components

In stage 3 of the render loop, we pass the renderer state to all of the **components** and update their inner state. In our case, there are only 2 components: the camera and the UI, but the concept can be applied generally.

Camera

We implement a perspective 3D camera with a configurable aspect ratio and field of view. The camera responds to movement actions for updating its position and mouse move actions for updating its rotation. The transformations are implemented using homogenous matrices.

User interface

We use the **imgui** library to create a simple user interface. The UI is responsive and we use it to render useful debug info in a concise way as well as to control the behavior of the program.

4.2.4 Draw call abstraction

Assembling all the pieces required to execute a draw call is very cumbersome when working in C/C++ and OpenGL directly. Being in Rust and using **glium** for OpenGL bindings, we are operating at a higher level of abstraction. We encapsulate all the pieces into a single struct and provide an easy way of creating it utilizing the builder pattern.

```
let fragment = RenderFragmentBuilder::new()
    .set_geometry(vertex_buffer, index_buffer)
    .set_vertex_shader(VERTEX_SHADER_SOURCE)
    .set_fragment_shader(FRAGMENT_SHADER_SOURCE)
    .build(display)
    .unwrap(); // <-- build returns a Result

fragment.render(target, uniforms, draw_parameters);
```

All of the variables in the above snippet are strongly typed, so we don't have to manage the resources and can call *render()* repeatedly easily. The builder returns a **result** type that we can inspect for errors in the geometry and shader configuration.

Remark. Utilizing Rust's type system and generics, the builder state machine could be represented entirely by types. This way an incorrect configuration wouldn't even compile! We didn't implement it this way due to time constraints.

4.3 Model

Figure 4.2 shows how the model modules interact to produce geometry that is then sent to the renderer. It takes the Minecraft data along with the renderer state as input and produces either the discrete scene directly from the data or the implicit scene by transforming the data into an implicit surface.

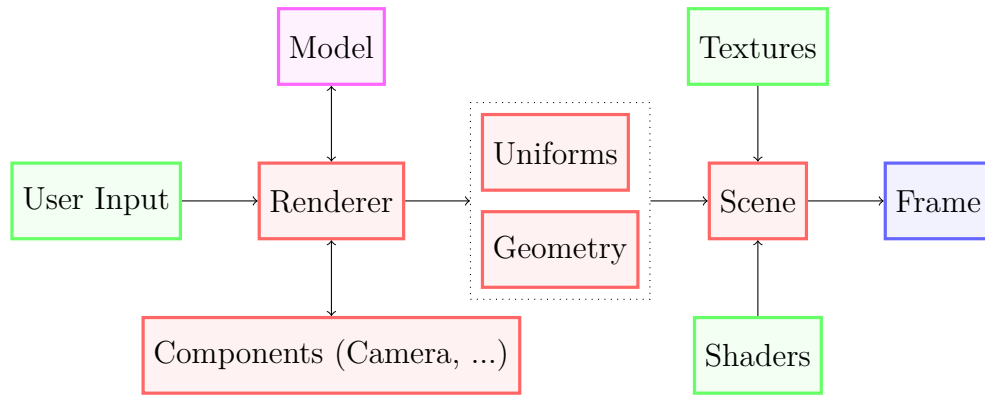


Figure 4.1: Renderer architecture

4.3.1 Smooth chunk loading

To achieve the illusion of an infinite world, we need to continually load chunks around the player as they move around. Loading a new chunk requires deserializing from the disk and building the smooth surface mesh for that chunk. We use the **fastanvil** crate for parsing chunk data from the binary anvil format that Minecraft uses. Both of these operations are quite computationally expensive and doing them for multiple chunks each update will not fit into our frame time budget.

We implement a solution that can load chunks without impairing performance. It will be discussed in detail in the performance chapter (5).

4.3.2 Discrete scene

The discrete scene involves rendering *many* cubes. We iterate over all the blocks in the scene and extract their positions and materials into a vertex buffer. This buffer is then used to spawn many **instances** of a static cube mesh in a single draw call.

A single chunk can contain up to 98304 blocks. When rendering a scene of 10x10 chunks, we are looking at about 10 million blocks. This is too much, even when using instancing. What's more, a lot of rendering time is spent on blocks that the player doesn't see and even worse, on blocks that can never be seen! A simple optimization that we can do is to only render blocks that are possible to see. We do this by checking the neighboring blocks and only rendering blocks that have at least 1 air block in their neighborhood. Blocks that have solid neighbors all around can never be seen.

Further optimizations could be done but we found excluding the inner blocks was enough to produce smooth framerates.

4.3.3 Smooth scene

We need to be able to query neighborhood data for any world position. A query like this is required to calculate the intersection with a convolution kernel which is needed both by the implicit function and the material blend. The world is partitioned into chunks, which are further partitioned into blocks so we can easily find what we need by following this schema:

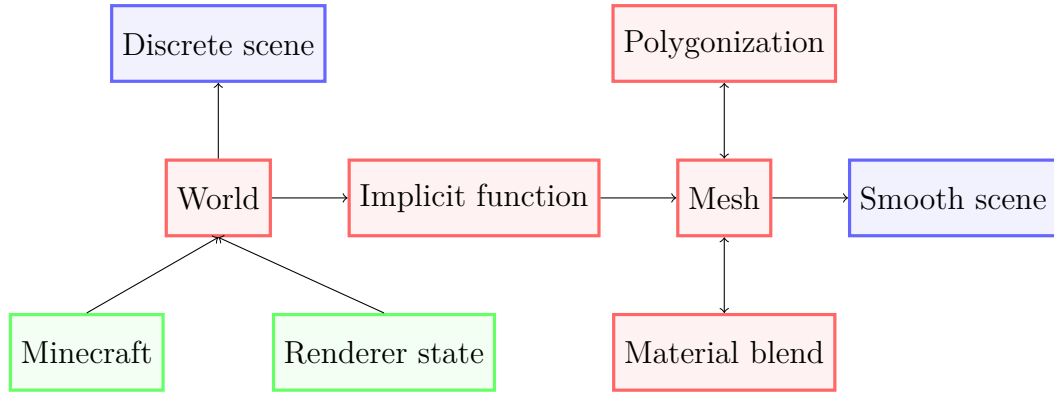


Figure 4.2: Model in-depth

1. Find all chunks that intersect the kernel in the XZ plane.
2. For every intersected chunk find all blocks that intersect the kernel in the Y direction.
3. For every block that intersects the kernel, calculate the exact volume of the intersection

We create the smooth mesh per chunk. The smaller chunk meshes are then stitched together to create one big mesh. Creating the mesh for a chunk requires that data for all neighboring chunks are loaded, otherwise, the chunks won't connect seamlessly. We solve this by ignoring the edge chunks in implicit mode. This means that when an $N \times N$ chunk world is loaded in discrete mode, implicit mode only displays $(N-1) \times (N-1)$ chunks.

Furthermore, since each chunk is polygonized separately, we can build the mesh in parallel to gain an increase in performance.

4.4 Pixel art textures

Even when rendering the smooth terrain, we use pixel art textures to preserve some of the art style. Working with such low-resolution textures presented a number of challenges. Since the texture pixels are much bigger than the sampled texels, we can get noticeable visual artifacts due to floating point errors when calculating the texture coordinates. This is not much of a problem within the textures but creates an ugly outline around the texture border. We solved this by manually offsetting the texture coordinates so they always point roughly to the center of a pixel. Another important thing was to make sure OpenGL doesn't generate any *mipmaps* for the textures as this was causing artifacts as well. Figure 4.3 shows a comparison of the textured terrain with and without these changes.

4.5 Controls

Visual aspects of the smooth mesh can be tweaked using a combination of runtime and compile time options. The compile time options are set as constant variables

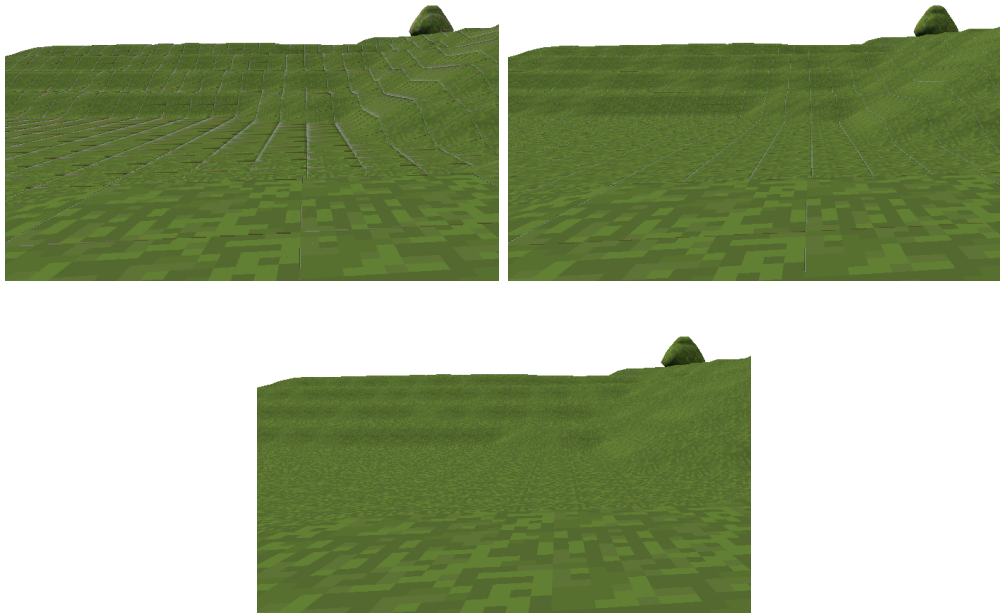


Figure 4.3: The top row shows artifacts caused by the use of pixel art textures. The effect on the left was caused by having OpenGL autogenerate mipmap levels for the textures. The image on the right shows what happens when we don't manually offset the texture coordinates. The bottom image shows the final version of the terrain, without artifacts.

in the source files. The runtime options can be set in-app in a UI window. See A.1 for a full list of options as well as a complete usage guide.

5. Performance

To provide a nice user experience, our app must run smoothly all the time. This leaves us with a very limited budget for what we can do in every frame if we want to keep a stable framerate. Some operations are too expensive, however, and don't fit into the budget. In this section, we will explore the performance bottlenecks present in our app and what we can do to maximize performance.

To evaluate the program's performance, we first need to measure it. Table 5.1 shows the average times needed to execute various actions on a single chunk. The values were measured on a desktop computer running the *AMD Ryzen 5 2600* six-core processor with *16GB* of *DDR4* RAM and a *Nvidia GTX 1070* graphics card. The smooth mesh resolution was fixed at 2 times the voxel grid resolution for the measurements. The chunks were also limited to 40 blocks in height to keep the times within an acceptable range. The times for the individual methods were measured using in-app timers and the performance of the program as a whole was measured using **valgrind**.

5.1 Caching and distribution of work

Creating the geometry for the scene is complex and expensive so it isn't something we want to do every frame. Our implementation is read-only, so it is totally valid to evaluate the geometry once and cache the result for subsequent frames. Doing it this way keeps the frame time low since everything is cached, but all the work needs to be done upfront. Still, this is the preferable approach, since most of the work can be done as part of application startup. As the player moves around parts of the cache are invalidated and work has to be done to update them, but the majority of the scene is still cached.

The updates are still pretty expensive and would cause very noticeable lag spikes if done synchronously. For this reason, we offload the work onto a separate thread, leaving the main render thread responsive. The main thread is then responsible for dispatching the work and collecting the results.

5.2 Discrete scene

The discrete scene is pretty straightforward to render. The naive approach is to just render all the cubes. Even with instancing, the FPS really starts to suffer in larger scenes. As said in section 4.3.2, we implement an optimization that cuts down on the number of cubes significantly. This improves FPS dramatically and since the blocks never change, the surface can be cached. Figure 5.1 shows how the amount of blocks affects FPS with and without our optimization.

5.3 Smooth scene

Creating the geometry for the smooth scene is significantly more complicated and computationally expensive than the discrete scene. The final mesh is pretty compact when it comes to the number of triangles, even with high mesh resolutions.

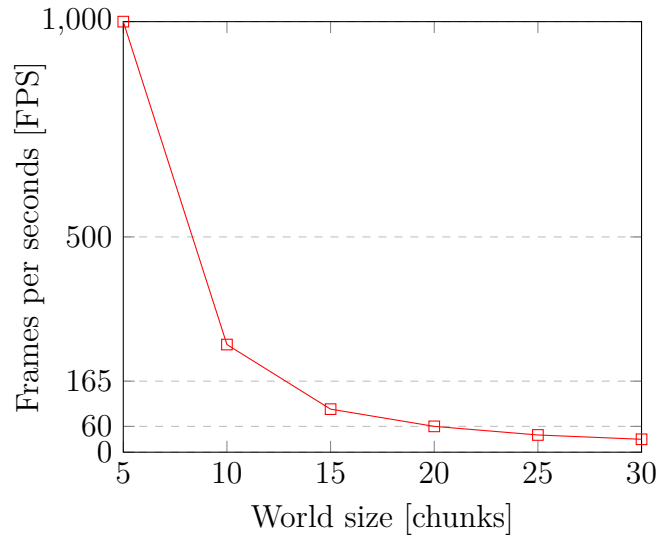


Figure 5.1: Frames per seconds measured at various world sizes

Action	1 pass	2 passes
Load chunk from disk	2.5ms	
Build chunk surface (discrete)	3.5ms	
Build chunk surface (smooth)	900ms	1600ms
Full chunk build	906ms	1606ms

Table 5.1: Average time required to execute actions on a single chunk

This means that FPS is never really a problem for this scene. The major limiting factor is the time it takes to build the smooth mesh. The built meshes are of course cached, but when a single update of the world takes 10 seconds, you are going to notice.

Table 5.1 shows how long it takes to do various operations on a single chunk. We can clearly see that building the smooth mesh takes the majority of time. This can be partially solved by building the chunk meshes in parallel. Figure 5.2 shows timings for a full update on worlds of different sizes. The times initially increase sublinearly, because we are building multiple chunks in parallel. Eventually, we fully saturate the available CPUs and things slow down. At around world size 20, the system can't keep up and you have to wait for things to load. This happens faster for higher-resolution meshes and worlds with a larger Y range.

This is not ideal, but we can do better. As we explained in 3.2.2, the mesh building algorithm can be parallelized. The performance of the polygonization algorithm is highly sensitive to tasks that have to be done for every single block. Implementing it as a compute shader and running it on the GPU in parallel per vertex could vastly improve the load times since it involves doing a little work for a very large amount blocks. This would allow us to experiment with finer meshes, bigger worlds, and more expensive visual effects. The current implementation is written in Rust using iterator abstractions and is what most of the code is dedicated to. Rewriting the whole algorithm in GLSL would take a considerable amount of time, which we didn't have.

This is also highlighted by the addition of multiple smoothing passes (see table

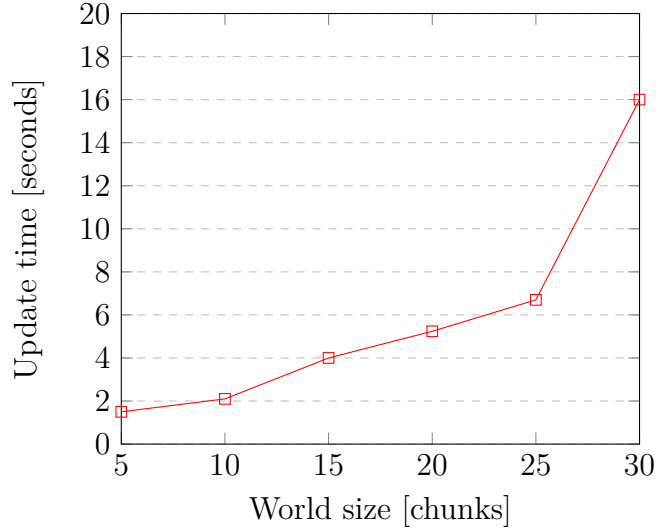


Figure 5.2: Update times for various world sizes

5.1). We didn’t find any clever way to do it, so having n passes effectively means polygonizing the whole scene n times. This lowers performance dramatically which is why we didn’t make the passes configurable and limit ourselves to 2 passes.

Another obvious improvement is to just do less work. We could for example partition the world in the Y axis also and use frustum culling to only polygonize and render parts of the world that are currently visible. This would split the work into smaller tasks and could improve throughput, especially in combination with GPU polygonization.

5.4 Scene update

As the player moves around, we need to load new chunks. This is done in stages. First, the main thread checks if we need to update and if so, dispatches the task of loading the chunk data to a worker thread. As seen in table 5.1, loading the chunks doesn’t take that long, but loading multiple chunks would still cause lags if done on the main thread. We also precalculate the chunk surfaces for the discrete scene in this step. After this step, the discrete scene is updated. Next, we dispatch another thread for building the chunk meshes. This step requires information about neighboring chunks, so we can’t build the meshes in the first step. They are built in parallel and the scene is updated once all of them are finished.

5.5 Memory

When it comes to memory, voxel representations are not very conservative. We store data about every block in memory at all times. In our case, the voxels are fairly large, though, and we didn’t run into any memory problems. A moderately big scene (world size 10) takes about 60MB of space, which is very low by modern standards.

Conclusion

We managed to create a graphical application that allows viewing Minecraft worlds as smooth surfaces. We implemented it in Rust using OpenGL for graphics. It runs in real-time and allows interactive exploration of the smoothed world as well as the original blocky world. We designed the solution by analyzing previous works related to our problem and found a method that could be applied to extract a smooth surface from the Minecraft world data. The solution involved an implicit function, which we converted into a signed distance field. Leveraging the properties of an SDF, we were able to selectively smooth out only the terrain and leave structures such as trees or houses untouched. Additionally, these blocky structures are smoothly joined to the terrain so there are no hard transitions. The sdf is rendered using a volume polygonization algorithm called marching cubes. We managed to map 2D textures onto the implicit surface mesh, blending them based on materials present in a local area. The resulting scene is fully shaded using a simple shading setup.

We managed to implement all of the initially set requirements. The shape and appearance of the smooth mesh are highly configurable. We didn't implement any terrain generation ourselves but explored how to control Minecraft's world generation instead. This makes our application a powerful tool for creating volumetric terrain with a relatively high degree of control.

Due to time constraints, we didn't manage to polish the application to the degree we wanted to. The area where this shows most is the performance of the application, but even in its current state, it is very usable.

Future work

The easiest thing to improve our work would be the inclusion of additional block types, with the eventual goal of supporting all block types. This would add to the variety of the scenes and make them more pleasing to look at. Next, exploring Minecraft world generation further could allow us to have more control over the input data. Improving the shading by adding effects like shadows and ambient occlusion would greatly add to the realism of the scene. A non-trivial feature to add would be some kind of a terrain editor, where we could fine-tune details of the implicit scene. Another nice addition would be the ability to export a part of the scene as a textured mesh that could then be used by other software. Lastly, there are many ways to improve the performance which would allow for a smoother and faster workflow.

Bibliography

- Csaba Bálint, Gábor Valasek, and Lajos Gergő. Operations on signed distance functions. *Acta Cybernetica*, 24:17–28, 03 2019. doi: 10.14232/actacyb.24.1.2019.3.
- Marcos Cirne and Helio Pedrini. Marching cubes technique for volumetric visualization accelerated with graphics processing units. *Journal of the Brazilian Computer Society*, 19, 09 2013. doi: 10.1007/s13173-012-0097-z.
- Eric Galin, Éric Guérin, Adrien Peytavie, Guillaume Cordonnier, Marie-Paule Cani, Bedrich Benes, and James Edward Gain. A review of digital terrain modeling. *Computer Graphics Forum*, 38, 2019.
- Jean-David Génevaux, Eric Galin, Éric Guérin, Adrien Peytavie, and Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32:1 – 13, 2013.
- Sarah Frisken Gibson. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. 1998.
- John Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12, 06 1995. doi: 10.1007/s003710050084.
- Houssam Hnaidi, Éric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum*, 29, 2010.
- Henrik Kniberg. Reinventing minecraft world generation. JFokus 2022, 2022. URL <https://www.jfokus.se/jfokus22/talks/913>.
- Gabor Lipton. Ray tracing implicit surfaces on the gpu. 2008.
- William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21: 163–, 08 1987. doi: 10.1145/37401.37422.
- Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Arches: a framework for modeling complex terrains. *Computer Graphics Forum*, 28, 2009.
- Inigo Quilez. 3D distance functions. <https://iquilezles.org/articles/distfunctions/>, 2013a. Accessed: 2023-7-19.
- Inigo Quilez. Smooth minimum. <https://iquilezles.org/articles/smin/>, 2013b. Accessed: 2023-7-19.
- Bruno Rodrigues de Araújo and Joaquim Armando Pires Jorge. Adaptive polygonization of implicit surfaces. *Computers & Graphics*, 29(5):686–696, 2005. ISSN 0097-8493. doi: <https://doi.org/10.1016/j.cag.2005.08.027>. URL <https://www.sciencedirect.com/science/article/pii/S0097849305001317>.

List of Figures

1	Example of terrain smoothed by our application.	4
2	Example of smooth terrain in combination with rigid blocks. . . .	5
1.1	Heightmap terrain	7
1.2	Hybrid terrain model	8
1.3	Ray marching visualization	9
1.4	Marching cubes - the 15 unique cases	10
1.5	Adaptive marching triangles	11
2.1	Minecraft world generation pipeline	13
2.2	Perlin noise	14
2.3	Multinoise	16
2.4	Cave systems in Minecraft	17
2.5	The influence of erosion on world generation	17
3.1	Modification of the marching cubes algorithm	19
3.2	Smooth minimum	21
3.3	Comparison of normals	21
3.4	Material blending	23
3.5	Texture mapping based on normals	24
3.6	Texture mapping banding artifacts	25
3.7	Parts of the smooth scene	26
4.1	Renderer architecture	29
4.2	Model in-depth	30
4.3	Troubleshooting pixel art textures	31
5.1	Frames per seconds measured at various world sizes	33
5.2	Update times for various world sizes	34
A.1	Block palette	39

A. Attachments

A.1 User guide

There is a number of variables that can be tweaked to modify how the rendered scene looks. They can be found in the **config.rs** source file (see table A.1).

Option	Type	Description
WORLD_FOLDER	string	The path to the minecraft region folder to be loaded
SPAWN_POINT	vec3	The position in the world where the camera is placed on startup
WORLD_SIZE	int	A number N. Only a NxN region of chunks is loaded at a time
MOVE_SPEED	float	How fast the camera moves
SENSITIVITY	float	How fast the camera turns
ASSETS_PATH	string	The path to the folder containing textures and other resources
DYNAMIC_WORLD	bool	If true, new chunks get loaded around the camera on demand as it moves
MULTIPASS	bool	If true, use 2 smoothing passes instead of 1
LOCK_LEAVES	bool	Use a static smoothness for leaves, independent of the terrain. Only has an effect if MULTIPASS is enabled
FILTER_RIGID	bool	Only render critical rigid blocks as sdf

Table A.1: Compile time options

A.1.1 Controls

The in-app camera is controlled using the standard **WASD** keys for moving *forward*, *left*, *back*, and *right* respectively. Moving *up* and *down* is done with the **J** and **K** keys. To rotate the camera in place, use the mouse when the cursor is captured. Use **SPACE** to toggle mouse capture. To toggle between the discrete and implicit view use **U**(discrete view) and **I**(implicit view). Pressing **B** toggles the wireframe. Closing the window or pressing **Q** will exit the application.

The in-app user interface also provides a couple of options. You can control the resolution of the smooth mesh as well as the strength of the smoothness. Note that higher mesh resolutions take a considerable amount of time to render (single digit minutes). Another option that can be tweaked in the UI for performance/visuals is the rendered Y range. Only blocks within the Y range are smoothed.

A.2 Material types

Our application supports 15 types of materials. Table A.2 shows a list of the materials along with their Minecraft counterparts. In some cases, multiple Minecraft blocks are mapped to the same material. Entries ending with "*" are wildcards and match all blocks that contain the text of the wildcard in their name. Figure A.1 then shows the textures we used in our application. When read from left to right and top to bottom, their order matches the order of materials in table A.2.

Grass	grass_block
Stone	stone*, gravel
Sand	sand*
Ore	ore*
Dirt	dirt*, granite
Wood	log*
Leaves	leaves*
Water	water*
Lava	lava*
Planks	plank*, oak_stairs
DarkStone	andesite, diorite
RedSand	red_sand
Cobblestone	cobblestone, white_terracotta
Glass	glass*
Cactus	cactus, sugar_cane
Unknown	*

Table A.2: List of supported blocks

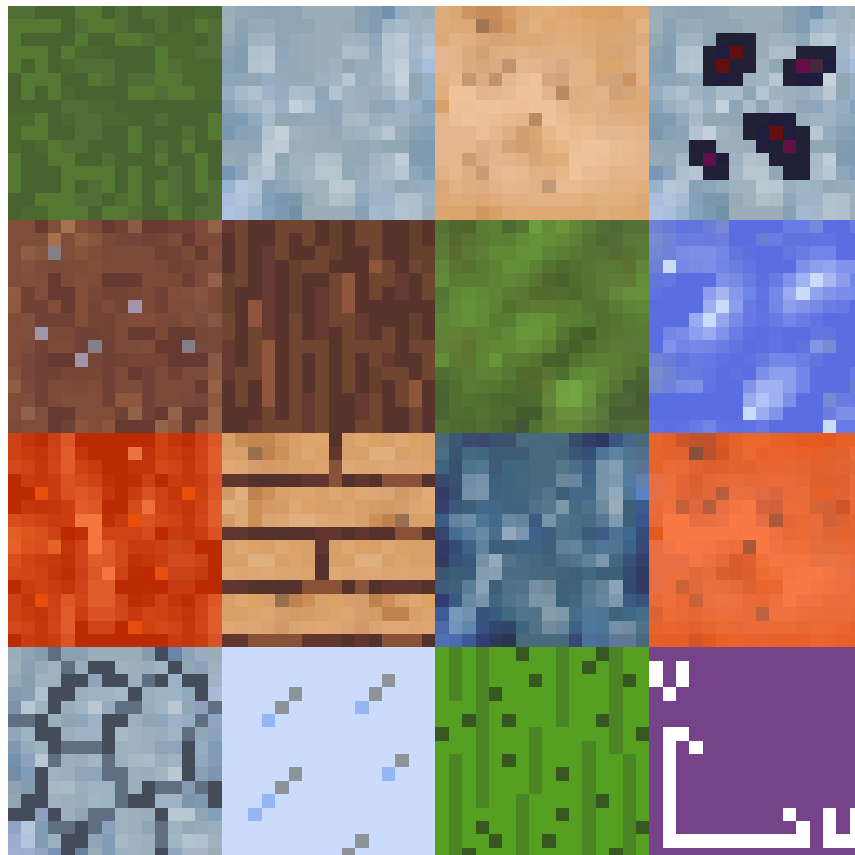


Figure A.1: Block palette