BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PROCEDURAL GENERATION AND SIMULATION OF 2D GAMING WORLD

PROCEDURÁLNÍ GENEROVÁNÍ A SIMULACE 2D HERNÍHO SVĚTA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	TOMÁŠ DUBSKÝ
SUPERVISOR VEDOUCÍ PRÁCE	Ing. TOMÁŠ CHLUBNA

BRNO 2022

Department of Computer Graphics and Multimedia (DCGM) Academic year 2021/2022

Bachelor's Thesis Specification

24365

Student:	**Dubský Tomáš**
Programme:	Information Technology
Title:	**Procedural Generation and Simulation of 2D Gaming World**
Category:	Computer Graphics

Assignment:

1. Study the basics of the rendering pipeline in OpenGL.
2. Learn about methods suitable for procedural generation and simulation of the 2D world environment.
3. Propose a non-trivial scene suitable for the demonstration.
4. Implement the proposed demo.
5. Document the achieved results.
6. Create a video presenting the results.

Recommended literature:

- Freiknecht, J.; Effelsberg, W. A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technol. Interact.* 2017, *1*, 27. https://doi.org/10.3390/mti1040027
- Ebert, David S., et al. Texturing & modeling: a procedural approach. Academic Press, 2014. ISBN 1483297020, 9781483297026

Requirements for the first semester:

- Items 1 to 3, experiments leading to item 4

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Chlubna Tomáš, Ing.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	January 19, 2022

Abstract

The goal of this thesis is to implement procedural generation and simulation of a two-dimensional gaming world. The virtual world is a seemingly-endless grid composed of small-tiled chunks which are generated and simulated when the player is nearby. The generated terrain consists of several biomes and underground caves. Liquids, gases or growth of grass are among the simulated processes of the world.

Abstrakt

Cílem práce je implementace procedurálního generování a simulace dvojdimenzionálního herního světa. Herní svět je tvořen nekonečnou mřížkou malých dlaždic. Tyto dlaždice jsou seskupeny do částí, takže svět je generován a simulován pouze pro ty části, které jsou poblíž hráče. Generovaný terén se skládá z několika biomů a podzemních jeskyní. Kapaliny, plyny nebo třeba růst trávy patří mezi procesy, které jsou simulovány.

Keywords

procedural generation, cellular automaton, 2D, tile-based game, particle system

Klíčová slova

procedurální generování, celulární automat, 2D, hra dělená na dlaždice, částicový systém

Reference

DUBSKÝ, Tomáš. *Procedural Generation and Simulation of 2D Gaming World.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Chlubna

Rozšířený abstrakt

Cílem práce je implementace procedurálního generování a simulace dvojdimenzionálního herního světa. Herní svět je tvořen nekonečnou mřížkou malých dlaždic. Každá dlaždice je tvořena dvěma vrstvami - pevným blokem, který neumožňuje pohyb hráči skrz, a pozadím, které na pohyb hráče nemá vliv. Obě vrstvy určují, z jakého materiálu jsou tvořeny (kámen, hlína, atd.), případně mohou být tvořeny vzduchem, což v případě bloku znamená, že pohyb hráče neomezuje. Tato reprezentace umožňuje hráči svět jednoduše měnit a simulovat v něm různé přírodní procesy, takže například kapaliny jsou reprezentovány jako druh bloků, který hráče pouze zpomaluje, ale nezastaví úplně.

Aby svět mohl být nekonečný, jsou dlaždice v mřížce seskupeny do pravidelných čtvercových částí, přičemž v paměti jsou udržovány pouze ty části, které jsou poblíž hráče. To znamená, že části světa se generují a ukládají podle toho, jak se hráč v něm pohybuje.

Tato reprezentace herního světa řadí výslednou aplikaci mezi hry s otevřeným světem, což znamená, že hráč se může volně pohybovat světem a není mu určen lineání postup hrou. Aplikace se dá také zařadit do kategorie her typu pískoviště, které se vyznačují tím, že hráč může svět libovolně měnit.

Generovaný terén se skládá z horizontu tvořeného několika biomy a podzemními jeskyněmi. Biomy jsou na horinzontu rozmístěny tak, aby plynule přecházely jeden do druhého a aby nevznikaly nepřirozené přechody, např. ze savany do tundry. Pod horinzontem se nachází různé jeskyně a velmi hluboko se generuje roztavené magma. Generování používá šumové funkce a celulární automaty, aby terén vypadal co nejvěrohodněji. Konkrétní podoba světa je definována magickým číslem, které určí hráč, takže je možné generování zopakovat.

Ve vygenerovaném světě jsou simulovány dva druhy procesů: dynamika tekutin a pomalé procesy jako třeba růst trávy. Simulace tekutin zahrnuje kapaliny (voda, láva) i plyny (vodní pára, kouř) a interakci mezi nimi (vypařování vody, srážení vodní páry, tuhnutí lávy). Tento systém je také použit pro simulaci ohně. Systém pro simulaci pomalých procesů je zaměřen na to, aby modifikoval co největší oblast světa. To znamená, že jeho účelem je potlačit pocit toho, že svět je mimo pohled hráče zcela statický. Kromě růstu trávy je použit pro simulaci šíření úmělých biomů do okolních, což je inspirováno podobnými hrami. Na oba druhy simulace lze pohlížet jako na celulární automaty.

Implementovaná aplikace používá knihovnu OpenGL pro hardwarovou akceleraci veškerého generování i simulování světa, což umožňuje generovat nové části světa bez viditelného zdržení a také to umožňuje simulovat velkou oblast světa i na méně výkonném počítači. Vykreslování světa je rozšířeno o jednoduchý systém osvětlení, takže například magma v podzemí osvětluje okolní jeskyně. Oproti zadání také umožňuje svět uložit a znovu načíst ze souboru. Aplikace byla testována na operačních systémech Windows a Linux.

Procedural Generation and Simulation of 2D Gaming World

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Tomáš Chlubna. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Tomáš Dubský
May 10, 2022

Acknowledgements

I would like to thank Mr. Tomáš Chlubna for helping me throughout the thesis.

Contents

Chapter 1

Introduction

The goal of this thesis is to create a two-dimensional game demo which procedurally generates a virtual world and simulates natural processes of the world.

The world is represented as an endless grid of tiles, i.e. small squares of different colors. The generated terrain consists of a landscape composed of biomes and caves underground, all composed of the tiles. The tiles are grouped into square groups called chunks. Due to memory limitations, only chunks near the player are generated and held in memory.

The generated terrain further changes by three means: modifications by the player (the player can add or remove tiles), tile transformations (e.g. growth of grass is implemented by slow transformation of nearby dirt tiles), and fluid dynamics (liquids, gases and even fire are also represented as tiles).

Both generation and simulation were designed and implemented with GPU acceleration in mind. This parallelization allows new chunks to be generated without noticeable delay. It also allows much larger area to be updated by the simulation than it would be possible if the simulation was done by CPU.

The next chapter 2 explains key technologies used for procedural generation and real-time simulation. It also overviews genres and existing games that are relevant to this project. Chapter 3 describes design of the terrain generator and design of the systems that perform the simulation. Chapter 4 explains implementation details of the design and issues that had arisen during the implementation. It also describes how the world is rendered. Chapter 5 assesses performance of the implemented application. The last chapter 6 concludes the thesis and proposes some ideas of how the project could be further extended.

Chapter 2

Technologies

The purpose of this chapter is to describe technologies which are important for the thesis. For all the mentioned technologies, only the aspects that relevant to the thesis are described. Section 2.1 overviews procedural generation, especially noises and interpolation which are used to create the gaming world. Section 2.2 explains cellular automata, a computation model used for simulation as well as generation of the world. Section 2.3 overviews OpenGL, a graphics application programming interface used for rendering and acceleration of the game. Section 2.4 explains real-time game loop, a design pattern that maintains fixed simulation speed at variously performant hardware. The last section 2.5 overviews genres and particular games that influenced this thesis.

2.1 Procedural generation

In the field of computer graphics, procedural generation refers to an algorithmic data creation process. Subsection 2.1.1 overviews noise, one of foundation stones of procedural generation. The next subsection 2.1.2 explains interpolation, an important parameter of noises. To put procedural generation into perspective.

2.1.1 Noise

Noises are commonly used to generate surfaces, shapes or other attributes of various objects - from surface of dirt or gravel, to shapes of mountains, or distribution of biomes in a terrain. It all depends on how the noise is interpreted. There are two frequently used categories of noise:

- **Value** noise uses a lattice of points which are assigned random values. The random value is usually created by hashing the coordinates of the point. The final value of the noise is interpolated based on the nearest points of the lattice.

- **Gradient** noise also uses a lattice of points which are assigned random values. The difference is that the points are interpreted as gradients of the function. The first known implementation of a gradient noise function was Perlin noise [3] by Ken Perlin. He also designed an improved noise called Simplex noise [7] which better scales to higher dimensions.

Output range of noises may differ but it is often desired to work with range [0, 1) or [-1, 1] so it is common to scale the output values to an appropriate range.

The basic noise functions do not resemble surface of any object. It is required to extend or modify them. One common modification is using fractal noise. A fractal noise is a noise created by stacking multiple octaves/layers of a noise. Each octave of the fractal has double frequency (half distance between points of the lattice) and half the amplitude (maximum value of the point). Figure 2.1 shows that the number of octaves does not need to be high to produce a fractal noise.

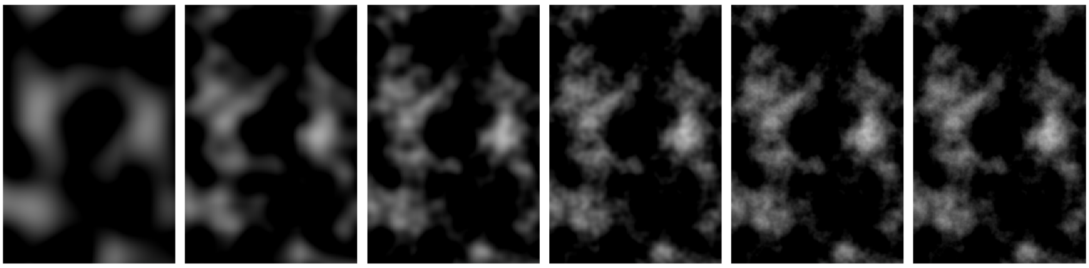


Figure 2.1: Creation of fractal noise. From left to right, each subfigure contains one more octave of Perlin noise and thus seems less blurry and more detailed. Since every next octave has less impact on the image, the difference between the last two is barely noticeable.

2.1.2 Interpolation

Interpolation is a process used by noises to determine values between the lattice points. The chosen interpolation method significantly affects appearance of the noise.

The further described method requires the lattice points to be arranged in a regular grid and uses only the nearest neighboring data points of the grid. The nearest neighbors are the vertices of the n-cube that the interpolated point lies in. This means that for n-dimensional interpolation 2^n points enter the interpolation.

To interpolate a data point, it is required to compute which points of the grid (the noise) will be used to interpolate it, i.e. the nearest neighbors. It is also required to compute its position relative to the neighbors within the n-cube that surrounds it. Formulas 2.1 and 2.2 show that respectively.

$$\boldsymbol{I} = \lfloor \boldsymbol{P} \rfloor \tag{2.1}$$

$$\boldsymbol{R} = \boldsymbol{P} - \lfloor \boldsymbol{P} \rfloor \tag{2.2}$$

Where:
$\boldsymbol{I}$ are lowest indices to the grid of the points that will be used. The other points are accessed by adding 1 to the indices, the indices can also be hashed to produced a random value;
$\boldsymbol{R}$ is the relative position of the point. Each component is in range (0, 1);
$\boldsymbol{P}$ is the position of the point that is interpolated.

The normalized position has to be converted to weight factor. Weight factor is a vector which describes relative weight of opposite sides of the n-cube. Weight 0 represents plane at lower coordinates, weight 1 presents plane at higher coordinates. There are several 1-dimensional functions that map component-wise the normalized position to the weight factor. They are crucial for the result of interpolation. A list of interpolation functions used in this thesis follows. All described functions expect already normalized position, i.e. in interval [0, 1).

- **Nearest-neighbor** interpolation 2.3, also step function or thresholding, is a simple interpolation function. Full weight is given to the nearest neighbor.

$$step(x) = \begin{cases} 0 & \text{x} \leq 0.5 \\ 1 & 0.5 < \text{x} \end{cases} \tag{2.3}$$

- **Linear** interpolation 2.4, also lerp or mix, becomes identity for normalized position.

$$mix(x) = x \tag{2.4}$$

- **Smoothstep** interpolation 2.5 uses a polynomial that has gradient equal to zero at the known points. This creates smoother transition at known points.

$$smoothstep(x) = 3x^2 - 2x^3 \tag{2.5}$$

- **Smootherstep** interpolation 2.6 also uses a polynomial, but both the first and the second order derivative equals zero at the known points [3]. This creates even smoother transition.

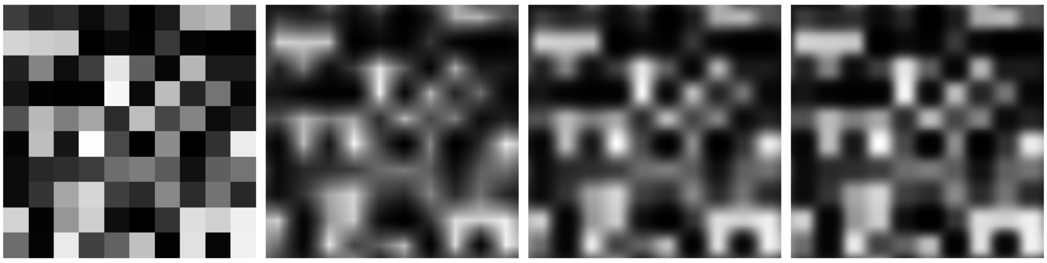
$$smootherstep(x) = 6x^5 - 15x^4 + 10x^3 \tag{2.6}$$

The weight factor $\boldsymbol{W}$ is used to compute weighted average of the neighboring points. The weighted average represents the interpolated value of the point. Equation 2.7 and 2.8 show the average is computed for 1D and 2D interpolation respectively. Higher dimensions are computed analogously. Figure 2.2 shows a value noise interpolated with different interpolation functions.

$$\boldsymbol{V} = (1 - \boldsymbol{W}_x)\boldsymbol{V_0} + \boldsymbol{W}_x\boldsymbol{V_1} \tag{2.7}$$

$$\boldsymbol{V} = (1 - \boldsymbol{W}_x)(1 - \boldsymbol{W}_y)\boldsymbol{V_{00}} + \boldsymbol{W}_x(1 - \boldsymbol{W}_y)\boldsymbol{V_{10}} + (1 - \boldsymbol{W}_x)\boldsymbol{W}_y\boldsymbol{V_{01}} + \boldsymbol{W}_x\boldsymbol{W}_y\boldsymbol{V_{11}} \tag{2.8}$$

Where:
$\boldsymbol{V}$ is the interpolated value of the unknown point;
$\boldsymbol{V_{xy}}$ are values of nearest neighbors of the interpolated point.

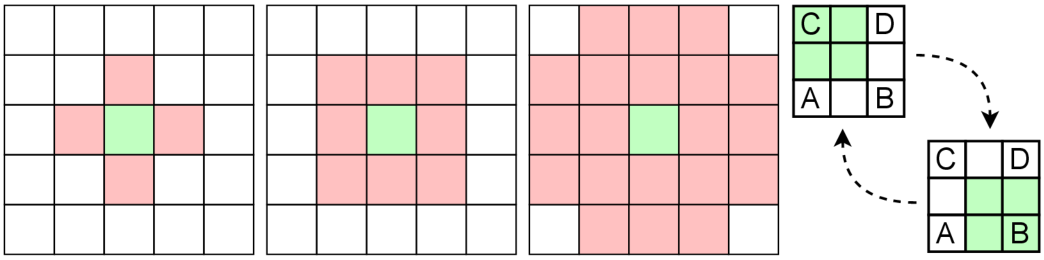


(a) Nearest-neighbor (b) Linear (c) Smoothstep (d) Smootherstep

Figure 2.2: Figures (a) - (d) show the same value noise interpolated with different interpolation functions.

2.2 Cellular automaton

Cellular automaton (CA) is a computational model which consists of an n-dimensional regular grid of cells that each have one of discrete states, and a rule which determines state transitions [8]. Typically, the model evolves in steps during which all cells of the model simultaneously switch to their next state based on the rule of the automaton. To select the next state of a cell, the rule uses the current state of the cell and cells in its neighborhood as input. Figure 2.3 shows several neighborhoods used in this thesis.



(a) von Neumann (b) Moore (c) 5×5 circle (d) Margolus (block)

Figure 2.3: Cellular automaton neighborhoods in a two-dimensional grid. The next state of the updated cells (green) depends on their state and the state of the cells in their neighborhood (red).

Even though CA are defined to work with infinite grid of cells, this usually is not possible for memory reasons. This raises a question on how to handle cells that are on the edges of the grid. There are many solutions to this. One is to use different rules specifically for these cells. Another solution is to keep their state constant (which can be viewed as a special rule too). One of the most common ways of handling this is to connect opposite edges of the grid. This means that cells on the opposite edges are considered neighbors. This solution is easily implemented and requires no special handling of the cells on edges.

Besides typical CA, several other types exist:

- **Stochastic** cellular automata differ in the fact that their rule is not deterministic. The next state is selected based on a probability distribution instead.
- **Asynchronous** cellular automata are not required to perform state transition synchronously for all cells but, instead, they may update cells individually.
- **Block** cellular automata apply the transition to a block of cells instead of single cell. The partition, which divides the cells into blocks, is usually shifted after every step.

2.3 OpenGL overview

OpenGL is a complex application programming interface (API). It is typically used to achieve GPU-accelerated rendering. The purpose of this section is not to overview the whole API. This section overviews only those parts of the API which are relevant to the thesis. The first subsection 2.3.1 outlines rendering pipeline. The second subsection 2.3.2 overviews compute shaders, a special object of the API which can be used for more general computation on GPU.

2.3.1 Rendering pipeline

Some OpenGL commands are used to specify geometry of objects that are to be rendered. Others manage various states that control how the geometry is rendered. Another group of commands invoke the actual rendering. These commands are effectively sent through a processing pipeline [5]. The pipeline starts by fetching vertices of the geometry and, after several stages, ends by writing the rendered pixels. Some of the stages are programmable, some allow selection of a fixed function. Figure 2.4 shows a diagram of stages that are relevant for the thesis.

The first stage is programmable Vertex Shader. Each invocation of the shader transforms a vertex of the geometry to an output vertex used later in the pipeline. Data types of input and output vertices may (and typically are) different but it is enforced that one input vertex is transformed to exactly one output vertex. It is usually used to control position of the rendered geometry.

The output vertices are processed by a few fixed function operations collectively called Vertex Post-Processing. Firstly, Primitive Assembly transforms a stream of vertices into a stream of base primitives. The base primitives are: points, lines and triangles. The next operation clips the primitives. Clipping removes portions of (or whole) primitives that would fall outside view. For partially clipped primitives, new vertices are generated by linearly interpolating their attributes. The clipped primitives are then rasterized. Rasterization is a process which generates fragments. Each fragment corresponds to a pixel of output framebuffer.

The next stage is also programmable and it is called Fragment Shader. Fragment Shader receives a fragment, whose attributes are linearly interpolated between the vertices of the primitive) and outputs a color.

The output color (sample) goes through another series of fixed function operations called Per-Sample Processing. Blending is the most important for 2D graphics. Blending controls how the sample is mixed with pixel of the framebuffer that it will overwrite. For example, blending allows, instead of simply replacing it, to mix them based on alpha of the sample.

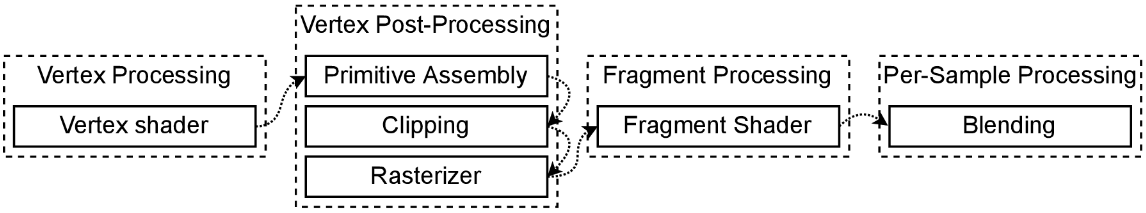


Figure 2.4: A simplified diagram of OpenGL rendering pipeline. Many stages are omitted, only stages relevant to the thesis are shown.

2.3.2 Compute shader

Compute shader is a shader stage which, unlike others, is not a part of the standard rendering pipeline. It is not primarily used for rendering but rather for arbitrary computation. Instead of using predefined inputs and outputs like other stages, it uses an abstract execution model which reads from and writes to arbitrary buffers and/or texture images [1]. The abstract execution model can be very easily used to simulate cellular automata on GPU without overhead of the standard pipeline.

Unlike other shader stages which are invoked by draw calls, compute groups are invoked by special dispatch calls. Similarly to draw calls, which specify the number of vertices that should be drawn, dispatches specify the number of work groups, which should be dispatched in an abstract three-dimensional space. Each of these dimensions can be one, so the space can be effectively reduced to two or one dimensions.

A work group does not invoke a single computational thread. Work groups also have a three-dimensional size called local size. Unlike dispatch size which may change dynamically, local size is a shader-compile-time constant specified in the code itself. If a different local size is needed, it is required to recompile the shader. Figure 2.5 visualizes the abstract space that threads execute in.

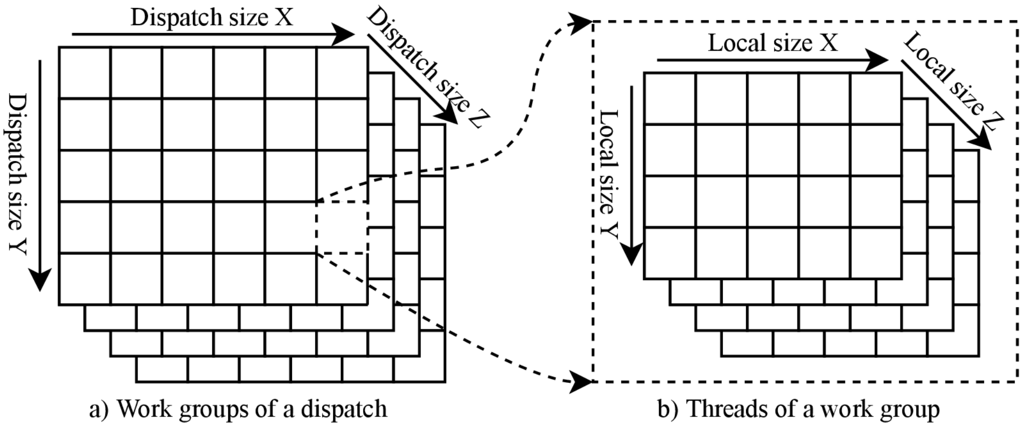


Figure 2.5: Abstract computation space of compute shaders. Each work group of a dispatch is identified by its 3-dimensional position. All work groups have the same local size - also a 3-dimensional vector which specifies the number of threads in each dimension within the group. A thread is then identified by local position within the group or global position within whole dispatch.

Given that each thread has a local position within its group $\boldsymbol{L}$, and each work group of a local size $\boldsymbol{S}$, has a position within the dispatch $\boldsymbol{W}$, global position of the thread within the dispatch $\boldsymbol{G}$, can be computed by Equation 2.9.

$$\boldsymbol{G} = \boldsymbol{W} \circ \boldsymbol{S} + \boldsymbol{L} \tag{2.9}$$

Threads within a group can communicate with each other. Among usual media such as storage buffers or texture images, compute shaders may use variables shared across the group. The communication utilizes the fact that threads within a group are executed as if in one computation unit with the same resources.

To effectively communicate, a thread has to write to a medium (shared variable, buffer, etc.), issue a memory barrier for the medium which waits on completion of the writes to the medium, and finally wait on a barrier which all threads of the group must enter until any is allowed to move past it. To ensure that threads cannot get locked on barriers indefinitely, it must be placed within uniform control flow. This ensures that either all or none of the threads reach the barrier. Another way of communication within a work groups are atomic operations. However, these are limited to operate on unsigned integer variables only.

As groups are not thought to be executed in parallel, neither communication nor synchronization between them is possible. The order of execution of groups also is not specified either so no assumptions can be done on that. Advantages of communication within groups are balanced by disadvantages that size of group must be known at compile time and also its maximum size is by orders of magnitude smaller than maximum number of groups that can be dispatched. The group size is also crucial for performance of the shader.

2.4 Real-time game loop

Real-time simulation, which real-time games are a type of, is a discrete simulation that runs at the same speed as the real world time does. This means that the next step of the discrete simulation is performed after a constant period of time from the last one. This cannot be achieved exactly on non real-time operating systems, because they cannot guarantee scheduling of threads to exact time points, but best-effort scheduling is good enough for most applications.

Another concern of real-time games is the frequency of rendering frames to screen. The simplest approach to this is to render the scene once after each step of simulation. This approach, however, shows weaknesses on both poor and powerful PCs. A poor PC may not be performant enough to even render a frame after each step, while a powerful PC could render more frames between two subsequent steps. Rendering multiple frames between two steps can be beneficial because it could render smooth transition between the two steps.

The game loop that incorporates constant rate of simulation steps and loose rate of rendering actually consists of two nested loops [6]. The outer loop renders frames and iterates until the game is closed. The inner loop checks for user input and performs one simulation step and it is iterated over as many times as it is required for the simulation catch up real time. The method is explained in detail in Algorithm 1.

Algorithm 1: Real-time game loop with periodic steps and variable rendering

```
$lag \leftarrow 0$;
$T \leftarrow \frac{1\text{s}}{\text{steps frequency}}$;
last frame stamp $\leftarrow$ current time;
while game is running do
    $now \leftarrow$ current time;
    $lag \overset{+}{\leftarrow} now -$ last frame stamp;
    last frame stamp $\leftarrow now$;
    while $lag \geq T$ do
        check for user input;
        perform one simulation step;
        $lag \overset{-}{\leftarrow} T$;
    render frame with interpolation factor: $\frac{lag}{T}$;
```

2.5 Sandbox games

Sandbox is a genre of games that have no predetermined goal and rather let the player to set their own goal. Virtual worlds of these games usually allow a lot of interaction and it is up to the player to decide what they want to do. Sandbox genre is closely related to open world games. Open world means that the player can explore the world freely, as opposed to a world with more linear gameplay. Sandbox games often use voxel-based representation of the world. Voxel-based representation uses a regular grid of values, usually called tiles in 2D, that determine what kind of material is present in the spot.

A subgenre of sandbox and voxel-based games are falling-sand games. Falling-sand games are based on a cellular automata behavior. Despite the name, the games typically do not simulate only sand but they simulate behavior of other physical phenomena, such as liquids, gases or even fire and explosions. Cellular automata cannot simulate the phenomena accurately but they produce convincing results while avoiding the complex equations that precisely describe the natural processes [2]. Performance is an important factor in these games as they need to run the simulation in real time.

Modern-day sandbox games often use procedurally generated voxel-based terrain. Terraria[1] is one of the best known and best selling[2] games of the genre. It pits the player into a world composed of many surface and underground biomes. Moreover, some of biomes are capable of spreading over the surrounding biomes so the player is challenged to keep the world in balance. Even though this project is inspired by the game, they differ particularly in the fact that Terraria's worlds are not endless and it simulates fluids by a system that is separate from tiles.

Since Terraria's release in 2011, many similar sandbox games have appeared, each having a different take on the tile-based world. Starbound[3], for example, creates more diversity by travelling between planets. Starbound uses an interesting lighting system suitable for tile-based worlds. It is undisclosed how it works, but the lighting system of this project is inspired by it.

Noita[4] is a falling-sand game that also influenced this project. This project uses tiles of the same size and also simulates fluids as tiles. The differences are that Noita generates its terrain very differently and its automata are accelerated by CPU multithreading[5], while this project uses shaders to accelerate it.

[1] https://store.steampowered.com/app/105600/Terraria/

[2] https://en.wikipedia.org/wiki/List_of_best-selling_video_games

[3] https://store.steampowered.com/app/211820/Starbound/

[4] https://store.steampowered.com/app/881100/Noita/

[5] https://www.youtube.com/watch?v=prXuyMCgbTc

Chapter 3

Design

The goal of this project is to create a 2D tile-based sandbox game with an endless world. Areas of the world, that have not been visited by the player yet, are procedurally generated. The generated terrain is composed of several surface biomes and caves beneath. The world is dynamic - the game simulates liquids, gases, fire, growth of grass etc. The player may freely modify the world and interact with the simulation. The world is saved to disk and can be loaded later.

The first section 3.1 explains the main concepts used for representation of an endless tile-based world. Section 3.2 explains in detail how the world is procedurally generated. Section 3.3 explains tile transformations, a system which simulates gradual processes within the world. Section 3.4 explains a system used for simulation of fluids.

3.1 2D tile-based sandbox

This section describes world as the central object of the project. The first subsection 3.1.1 describes tiles - the smallest unit of the world. The following subsection 3.1.2 explains why and how the tiles are grouped into sections called chunks. Subsection 3.1.3 explains how the world is stored in memory. The last subsection 3.1.4 overviews the main actions of a simulation step.

3.1.1 Tile

The in-game world is a two-dimensional grid of tiles. A tile is a small object with square shape that has two layers: a block in the front and a wall at the back. Both block and wall can be partially or completely transparent. If they are both transparent, background sky is visible. An important difference between block layer and wall layer is that blocks are understood to be solid and thus block movement of the player, while walls are considered to be behind the player and therefore do not block the player's movement. An exception to this rule are fluid blocks, these do not block the player's movement completely but rather slow it down.

As Figure 3.1 shows, each tile is defined by four attributes in total: block type, block variant, wall type and wall variant. Variant is a value that further specifies the type. The specific meaning of a variant value depends on the type. For most blocks and walls, it only distinguishes which of the similar-looking variants is drawn. For some tiles, it can store more complex information, e.g., velocity of a fluid.

The whole project is designed with GPU acceleration in mind. A tile represented by 4 attributes offers a simple mapping to typical image channels - the attributes are mapped to red, green, blue and alpha channels respectively. This implies that all attributes must be of the same datatype. This mapping is used for accessing of tiles as texels of a GPU texture as well as for saving tiles as pixels in a raster-graphics file.

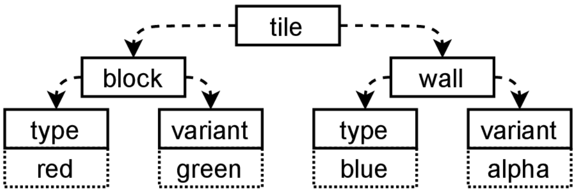


Figure 3.1: The four attributes of a tile. A tile has two layers, block and wall. Both layers have two attributes: type and variant. When tiles are stored in textures, the four attributes are mapped to red, green, blue and alpha channels respectively.

3.1.2 Chunk

For effective generation, in-memory storage and to-disk saving, the tiles have to be grouped. The approach taken was to separate the seemingly endless grid of tiles into another grid of fixed-size square sections called chunks. The world is then generated, as well as saved to disk, on per-chunk basis. Therefore, if the view approaches an inactive chunk, it is activated by one of measures, in the following order of priority:

1. The chunk is searched in the CPU memory.
2. The chunk is searched in the world save files.
3. New chunk is generated.

Activation of a chunk means that its texels are copied from one of the sources to the GPU memory for effective drawing and simulation of the chunk. Due to memory limitations, activation of a chunk may involve deactivation of another chunk. Deactivation means that it is copied back to the CPU memory. Chunks in the CPU memory are saved to disk once they have not been activated for a period of time or when the application closes.

3.1.3 Memory representation of the world

The design decision to store the world on a per-chunk basis is good for loading and saving of the world. It is also convenient for storage of inactive chunks in the CPU memory where each chunk is a fixed-size array allocated on the heap.

However, it is not ideal for storage of the chunks in terms of textures on the GPU side. For a shader that renders the tiles, it would be very inconvenient if each active chunk was a texture on its own as it would complicate fetching of tiles. Each visible chunk would require its own draw call with the corresponding texture bound, or another mechanism would be required to determine which texture to read from. Even more unpleasant issues would arise into shaders that perform the simulation of the world. Tiles near edges of a chunk would have to read from several textures and possibly write to several textures. This would be required, for example, when a liquid drop was moving near edge of a chunk.

The chosen solution was to store all active chunks in a single texture with dimensions that are multiples of chunks' dimensions. Each texels of the texture represents a tile of the world. This texture is further referred to as the world texture.

Activation of a chunk then means that its tiles/texels are copied to the corresponding location inside the world texture. Equation 3.1 explains how global position of a chunk is converted to position inside the single world texture. The chunk placement pattern that this formula produces can be seen in Figure 3.2.

$$\boldsymbol{P} = (\boldsymbol{G} \bmod \boldsymbol{W}) \circ \boldsymbol{C} \tag{3.1}$$

Where:
$\boldsymbol{P}$ is the resulting position of the chunk inside the world texture, measured in tiles/texels;
$\boldsymbol{G}$ is the global position of the chunk, measured in chunks;
$\boldsymbol{W}$ is the size of the world texture, measured in chunks;
$\boldsymbol{C}$ is the size of a chunk, measured in tiles/texels.

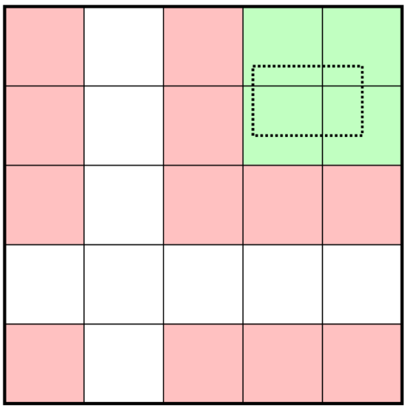


Figure 3.2: Placement of chunks inside the world texture. The black lines show how the whole world texture is divided into chunks. The dotted rectangle represents a view into the world. The chunks that are overlapped by the view rectangle (green) are the chunks that have to be active. The neighboring chunks (red) could be activated in subsequent simulation steps, depending on movement of the view.

3.1.4 Actions of a simulation step

A simulation step consists of a few important actions. Firstly, the player is moved according to user's input. Based on player's new position, position of the view is recalculated. The view smoothly follows the player. Depending on the position of the view, chunks that newly overlap the view are activated. The world is then modified by three simulation systems:

1. Modification by player is the simplest system. The player can place and remove blocks and walls by mouse cursor. It possible to modify the world with square or circular brush because manipulating each tile separately would be very tedious. Modifications by the player are instantaneous and unlimited.

2. Tile transformations are used to model slow gradual processes, such as growth of grass. To make the process gradual, tiles are not transformed, based on their surroundings,

as soon as the surroundings change but rather after some random time. For example, when a grass tile appears, it does not grow to a neighboring dirt tile until a random period of time has passed. Another important aspect of this process is that it gives the impression that the whole world is slowly transforming so that it does not seem that everything outside the view is static.

3. Fluid dynamics move tiles. In this context, moving means that the block of the fluid is swapped with the block of the target tile which generally is an air block. It is not a gradual process as it typically moves the fluids quickly until a stable position is reached and then it remains mostly static until the player intervenes the simulation. Therefore, it changes the terrain either very quickly or not at all. Unlike transformations, it only modifies area near the player because it is computationally more demanding and because it would unnecessarily update tiles that most likely would have already reached a stable position.

All three simulation systems involve some kind of stochasticity. Modification by player assigns random variants to new tiles, tile transformations update a tile after a random period, fluid dynamics updates tiles in random order. The purpose of this, among other, is to make the simulation look more natural.

After the simulation steps, the world is rendered zero or more times (depending on timing) and a new simulation step begins. Figure 3.3 shows a block diagram of the simulation.

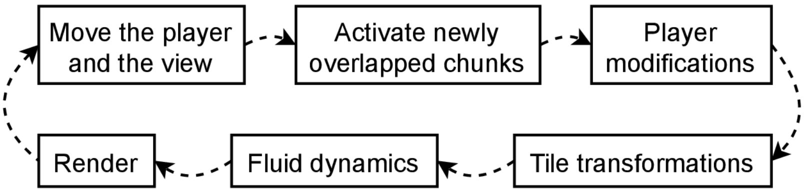

Move the player and the view
Activate newly overlapped chunks
Player modifications
Tile transformations
Fluid dynamics
Render

Figure 3.3: A simulation step is composed of several actions. At beginning of each step, the player is moved based on user's input. This also moves the view because the view follows the player. Movement of the view may trigger activation of newly overlapped chunks. Then three simulation systems modify the world texture: modifications by the player, tile transformations and fluid dynamics. The modified world is finally rendered and a new simulation step begins.

3.2 Procedural generation of the world

A new chunk is generated when the player reaches a chunk that they have not visited yet. The generation is done by shaders that manipulate a texture until it contains a new chunk. There are several consecutive stages in the generation process. After the last finishes, the new chunk is copied to its position inside the world texture.

The first subsection 3.2.1 outlines stages of the process. Subsection 3.2.2 describes generation of cave layer, subsection 3.2.3 describes generation of horizon layer, and subsection 3.2.4 explains how the layers are connected. Subsection 3.2.5 explains how bumps of cave edges are created to make them more natural. Subsection 3.2.6 explains how monotony of areas made of the same tile is avoided. The last subsection 3.2.7 covers how the generation was prototyped.

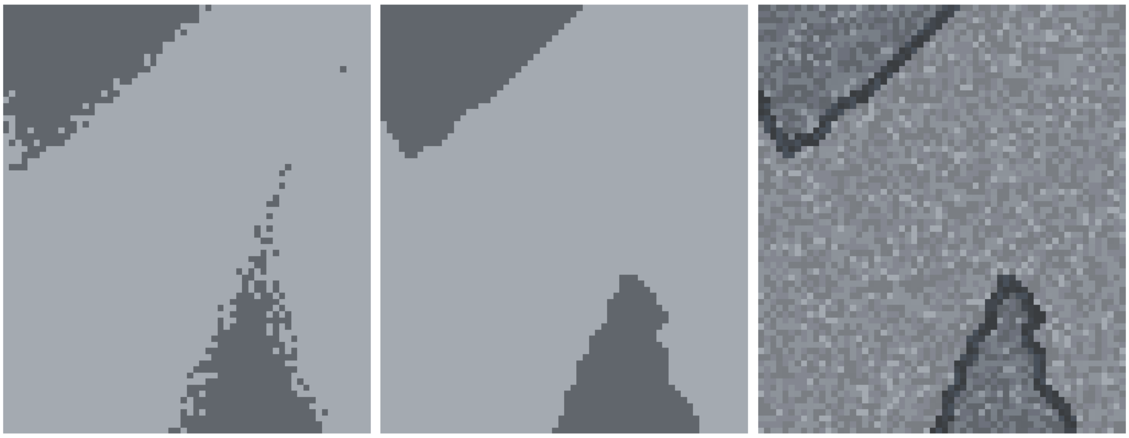
3.2.1 Stages

The generation process is divided to 3 consecutive stages. The effect that each stage has on the image can be seen in Figure 3.4. The 3 stages are as follows:

1. Fundamental structure is created. The generated terrain can be vertically divided to 3 layers: underground caves, horizon composed of various different biomes, and sky above the horizon. Edges of caves are intentionally slightly dithered.

2. Several iterations of a cellular automaton, which consolidates the edges of the caves, are performed. This creates edges that are rougher than perfectly smooth edges of the underlying noise, and thus the caves appear more natural.

3. Based on its neighboring tiles, a variant is chosen for each tile from two categories: inner and outer. This stage is introduced to highlight transitions between occupied tiles and air. Since there are several different variants in both categories, one of them is randomly chosen from the appropriate category. This breaks monotony of areas made of the same tile.



(a) Fundamental structure (b) Consolidation of edges (c) Variant selection

Figure 3.4: Effects of each generation stage on a sample cave.

3.2.2 Cave layer

Caves are created by varying solidity of underground tiles. If the solidity is higher than a threshold, the tile becomes a solid block with wall behind; if the solidity is lower than the threshold, the tile becomes background wall only. There are two distinct types of caves, both of them are based on simplex noise. The first type of caves is a fractal noise. The second type computes the minimum of the octaves. Equation 3.2 explains the fractal type of caves while Equation 3.3 explains the second type of caves in detail.

$$s_0 = \sum_{k=0}^{2} \frac{|snoise(2^k \cdot c \cdot \boldsymbol{P})|}{2^k} \tag{3.2}$$

$$s_1 = \left(\min_{k \in \{1,1.5\}} \left| \frac{snoise(k \cdot c \cdot \boldsymbol{P})}{k} \right| \right) /w \tag{3.3}$$

Where:
s_0, s_1 is the base solidity of the tile based on the two types of caves;
c is a world-to-noise position scale;
$\boldsymbol{P}$ is the position of the tile;
w is a cave width factor.

To make smooth transitions between the two types, the selection between the two types is also made using an oversaturated simplex noise. The values of this noise are then used to linearly interpolate between the two types of caves.

Caves generated by the above formula would, however, look too smooth and thus unrealistic. To create a more natural looking caves, edges of caves are randomly dithered and then, in the next stage of the generation, they are consolidated. This process results in bumps on the edges of caves and thus the caves look more natural. Equation 3.4 explains how is the base solidity dithered.

$$s = s' + t \cdot (hash(\boldsymbol{P}) - s') \tag{3.4}$$

Where:
s is the final solidity of the tile;
$hash(\boldsymbol{P})$ is a simple hash that maps the tile to interval $[0, 1)$;
s' is the base solidity of the tile;
t is the interpolation weight, Figure 3.5 shows how its value changes the resulting caves.

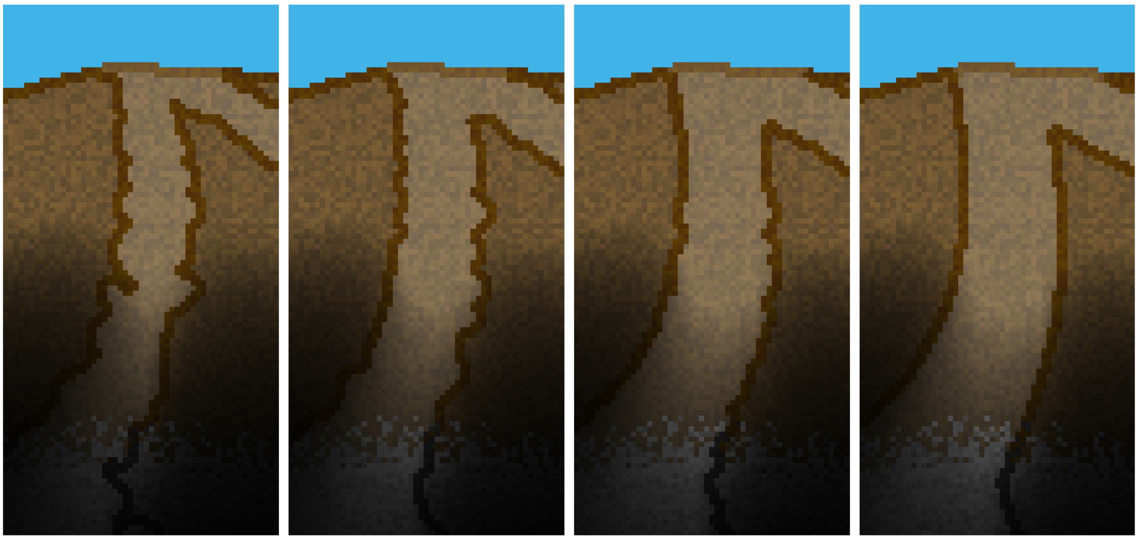
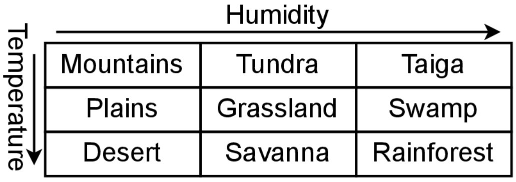

(a) 0.7 (b) 0.8 (c) 0.9 (d) 1.0

Figure 3.5: Images (a) to (d) show the same cave with increasing weight of the noise and decreasing weight of the dithering effect. Image (d) effectively shows the noise without dithering.

3.2.3 Horizon layer

The horizon is a layer between the cave layer and sky. The main borderline between the sky and terrain is based on a one-dimensional fractal value noise. More precisely, it is a sum of two such noises. One of the noises represents elevation of the surface, and the other represents roughness of the surface. The elevation generally has higher amplitude and lower frequency than the roughness. Crucial difference between elevation and roughness is that amplitude of roughness depends on gradient of elevation. The steeper the terrain is, the rougher its surface is. To further support the desired effect, elevation is interpolated using smootherstep interpolation, whereas roughness is interpolated using linear interpolation. The width of the surface layer also depends on gradient of elevation - the steeper the terrain is, the thinner the surface layer is. The purpose of this is to create fertile valleys and rocky hillsides.

To make the horizon more heterogeneous, biomes are introduced. A biome controls all properties of the horizon: the type of tiles that the surface layer is made of and its width, and also elevation and roughness of the horizon. The biomes do not correspond exactly to geographical biomes. They are rather abstract and naively imitate real biomes. In order to make the biomes at least a little realistic, unnatural transitions, such as transition from hot desert directly to snowy tundra, have to be avoided. To achieve that, biomes are created by altering climate. Climate is represented by 2 parameters: temperature and humidity. Given that each parameter is discretized to 3 values (low, medium and high), 9 different biomes are introduced. Figure 3.6 shows the matrix that the biomes form.


Humidity
Temperature

Mountains	Tundra	Taiga
Plains	Grassland	Swamp
Desert	Savanna	Rainforest

Figure 3.6: A biome controls several properties of the horizon. The biomes are laid out over the horizon by moving within the matrix and interpolating the properties. The matrix is naive and does not correspond to geographical biomes.

The biomes are then spread over the horizon by 2 noises: one represents temperature and the other humidity. The input of these noises is horizontal position of the tile. Both temperature and humidity are continuous quantities. The resulting biome is a combination of 4 nearest biomes. This is deliberate as it creates smooth transitions between otherwise discretely defined biomes. Surface layer width, elevation and roughness are linearly interpolated among the nearest biomes. The type of surface tile is the tile of the nearest biome but, for the purpose of this selection, the climate is randomly offset by a small amount. This creates a dithering effect on the transition between biomes. Figure 3.7 shows how the transition between two biomes looks like.

3.2.4 Transitions between layers

Transition from horizon layer to sky layer is as simple as comparing whether the tile is above the horizon noise function. Transition from surface to cave layer requires more complex test.

Another dithering effect is created by randomly offsetting the lower border of the surface layer for each tile. This blurs the border between surface and cave layer in a similar way as biomes do. Another issue is that the caves below are purposely much more common than in the real-world terrain. This is to make the cave layer more interesting. But, if the caves were also this common near surface, it would be so indented that the player would not be able to walk it without falling into caves constantly. To solve this issue, the solidity of the tiles is increased in close proximity to the horizon and thus the caves are thinner near the surface and it is possible to walk the horizon.

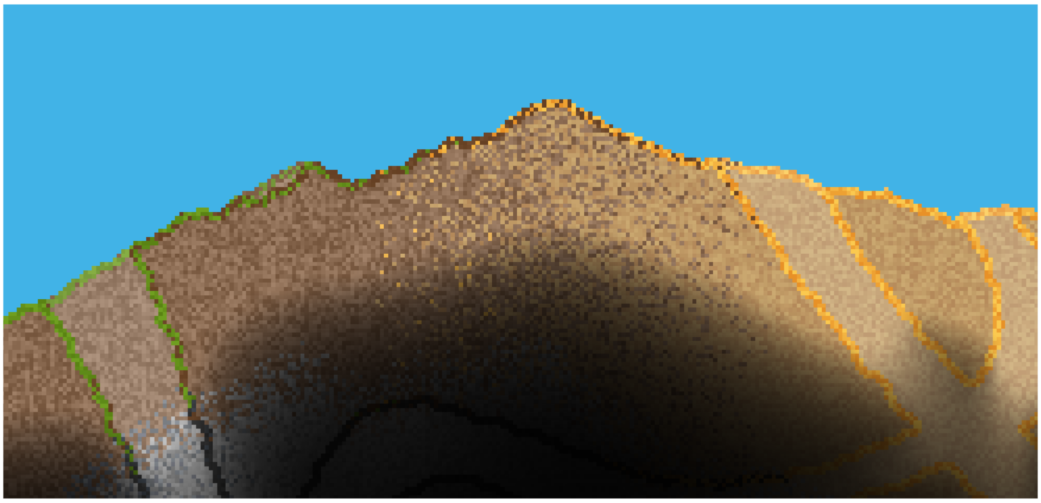


Figure 3.7: Transition between grassland and desert biome. A thin dirt biome is inserted in between to avoid unnatural transition from sand directly to grass. Also, the transition between dirt and sand is not sharp but dithered. The same applies for transition from surface layer to cave layer below.

3.2.5 Consolidation of cave edges

Once the structure is generated, several iterations of a cellular automaton are performed. The automaton turns dithered edges of caves into random bumps while other parts of the terrain remain nearly unchanged. The transition rule of the automaton follows:

1. Count the number of occupied (i.e. non-air) tiles in Moore neighborhood.
2. If the tile is occupied and the number of occupied neighbors is lower than a low-threshold, turn the tile into air.
3. If the tile is air and the number of occupied neighbors is higher than a high-threshold, turn it into the same occupied tile as its neighbors.
4. Otherwise, the state of the tile remains unchanged.
5. Apply the above to both blocks and walls separately.

The automaton intensifies extrema - areas that consist mostly of air, eliminate isolated tiles in it, and the other way around. In other words, air pushes out solids in some parts of the dithered edges, while in some parts stone pushes out air. The thresholds and the number of iterations are crucial for the result.

3.2.6 Variant selection

The last stage manipulates only the variants of blocks and walls, not their type. Variants of most blocks have no other purpose than modifying its appearance - variants are used to highlight edges and break monotony and areas made of the same tile.

This stage is technically a cellular automaton too. Unlike the previous, only one iteration is required though. The automaton's state transition rule follows:

1. Search the 5×5 neighborhood.
2. If air is found, use category of outer variants. If no air is found, use category of inner variants.
3. Within the appropriate category, select a random variant by hashing position of the tile.
4. Apply the above to both blocks and walls separately.

3.2.7 Shadertoy prototype

To ease and accelerate the development, Shadertoy[1] was used to prototype structure generation. Shadertoy is a web page that allows quick creation, compilation and visualization of fragment shaders. Shadertoy does not provide access to other shader stages and thus no input geometry can be defined. This is not a problem because the terrain generation does not use any input geometry. Also, being a web page, shaders are executed from within browser's WebGL context, which is based on OpenGL ES and not core OpenGL as the main application is. Technically, this means that the prototype is written in different shading language than the desktop application, but the languages are very similar so it takes minimal effort to port the prototype to the main application.

By using trial and error tests, the prototype was used to quickly determine various parameters of the generator. A sample of terrain generated by the prototype shader can be seen in Figure 3.8.

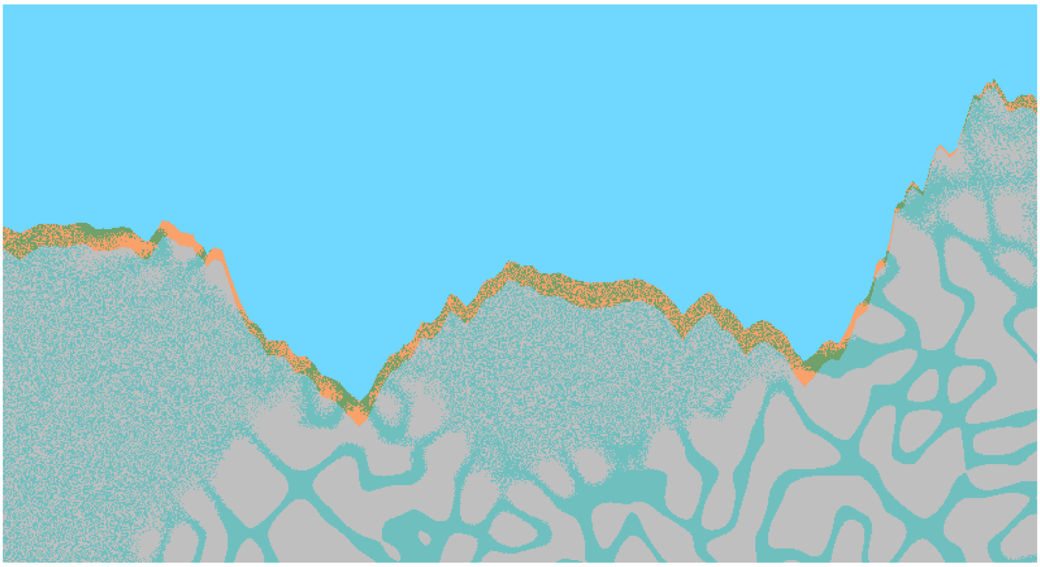


Figure 3.8: Sample image of terrain generated by the prototype shader

[1] https://www.shadertoy.com/

3.3 Tile transformations

Tile transformations is a system that modifies the world texture. It can be viewed as an asynchronous cellular automaton. Its main goal is to simulate slow processes, such as spread of biomes or growth of grass, in as big area as possible create a feeling that the whole world is simulated.

Subsection 3.3.1 explains how graduality of processes, that it is supposed to model, is achieved in the domain of discrete tiles and discrete simulation steps. Subsection 3.3.2 explains discontinuities - an undesired phenomenon that complicates selection of tiles to be transformed. Subsection 3.3.3 introduces a method which selects the tiles so that the discontinuities are avoided.

3.3.1 Gradual processes in a world of discrete tiles

Tile transformation is designed to simulate slow gradual processes, for example growth of grass. In the real world, grass grows very slowly. One way to represent this in a tile-based world is to have many variants of partially grown grass and gradually switch to a more grown variant. This was not been used because it requires all tiles to be updated every step. Instead, grass grows to neighboring dirt by instantly converting the dirt to grass but it does this only once a random period of time has passed. So the question is how to identify the moment when a grass (or other slow process) should spread.

The first way is to transform/update all tiles every nth step instead of updating them every step. This is not a good approach as it, for example, makes all grass grow synchronously which looks very odd. Another approach is to update all tiles every step but run a probability test inside the shader and update the tile only if the test succeeds. This is a good looking approach but not a performant one since the probability to transform the tiles has to be very low. This means that most invoked threads would perform no useful work.

A better approach is to update only a few (not all) randomly selected tiles every step. This is a good looking solution and a performant one at the same time as it avoids a probability test and all invoked threads perform useful work. The probability is instead introduced implicitly by updating only some tiles. The tiles are chosen randomly by hashing threads' IDs and a time-counter so that different tiles are transformed each step.

All mentioned approaches include a potential hazard however - reading of a texel that is being written to at the same time. This is a hazard because it is not well defined whether the old or the new value is read. No measures have been taken to avoid this hazard. The reason for this is that it is not critical which of the values is read because the shader is modeling a stochastic process anyway. From this perspective, the hazard can be thought of as an additional layer of randomness. Furthermore, a computationally expensive synchronization mechanism would be required to avoid a hazard that manifests rarely.

3.3.2 Discontinuities inside the world texture

It is desired to update an area that is as huge as possible. Should only area near the player be updated, it would seem that the simulation stops when the player is not nearby. While this ultimately is true, it has to be hidden from the player as much as possible.

In regard to the fact that all chunks are stored in a single texture, it would seem that any texel/tile of the texture can be selected for transformation. This is not true because of discontinuities inside the texture. Discontinuity is a phenomenon where neighboring tiles inside the texture are not actual neighbors of the world. It must be avoided to transform tiles near discontinuities because their transformation would be based on invalid tiles and therefore artifacts would appear. Since it is guaranteed that the tiles inside a chunk are continuous, the discontinuities can only appear on edges between chunks. More specifically, there can be two types of these discontinuities.

Firstly, there can be locations that contain no loaded chunks yet. This happens when a world is loaded because the world texture initially does not contain any chunks. This type of discontinuity soon disappears as the player moves and chunks get loaded into the texture.

The second type of discontinuities, which soon prevails, is the discontinuity between two chunks that are placed next to each other inside the texture but are not placed next to each other with respect to the whole world. This type is an inevitable consequence of mapping of endless world to finite-size texture. The used mapping (eq. 3.1) places chunks, whose difference in global position is a multiple of size of the world texture, to the same slot inside the texture. As a consequence, chunks that are neighbors in the texture may be a multiple of size of the texture apart.

3.3.3 Update chunks

Since all discontinuities appear on the edges between chunks, it is favorable to base the selection of tiles to transform upon chunks because if all neighbors of a chunk are valid, then all tiles of the chunk are guaranteed to have valid neighbors too.

It is required to keep track of the global position of each chunk in the texture in order to determine whether chunks are actual neighbors. So every time a chunk is activated, apart from uploading its texels to the appropriate location in the world texture, the global position of the chunk is stored in a supporting storage buffer (the buffer contains an array of the same size as the maximum number of chunks inside the texture). When a new world is loaded, the global positions in the supporting storage buffer are initialized to a special constant which denotes that no chunk has been loaded at the location yet.

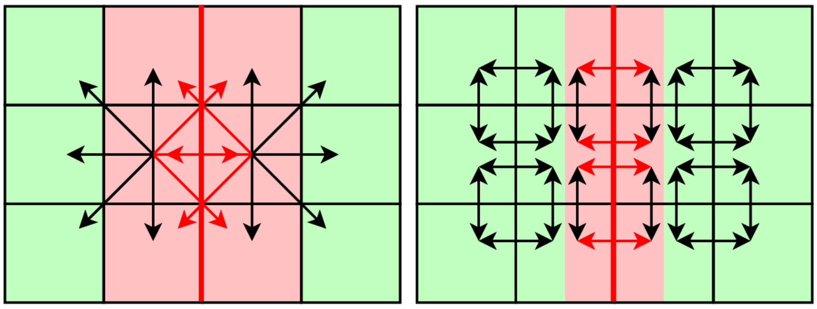
The straightforward solution would be to check each chunk whether all its Moore neighbors are valid. This is a valid solution but it has a major flaw. Every discontinuity incurs a two-chunks-wide strip which cannot be updated. This strip is unnecessarily wide. Additionally, each chunk has to observe eight chunks in its neighborhood.

A better solution is to use chunks shifted by half. Each such update chunk then overlaps four standard chunks which have to be continuous so that the update chunk can be updated. Most importantly, discontinuities incur only one-chunk-wide strips that cannot be updated when this solution is used. Secondly, memory accesses are slightly reduced as each update chunk needs to observe only four chunks. Figure 3.9 illustrates the difference between updates based on standard and update chunks.

For correct behavior, the discontinuity detection also has to respect the cyclic nature of the global-to-texture mapping. This means that update chunks that overlap an edge consider chunks on the opposite side of the texture as a their neighbors. Figure 3.10 shows an example world texture and update chunks inside it.

The final result of this procedure is a serialized list of update chunks that can be updated. This list has to be updated every time a chunk is activated but no more than that.

A fixed number of tiles is then randomly selected from each chunk in the list each step to be updated by the automaton. This means that the specific tiles, that get updated, are



(a) Continuity analysis based on standard chunks

(b) Continuity analysis based on update chunks

Figure 3.9: The advantage of continuity analysis based on shifted chunks over standard chunks. Arrows indicate neighboring chunks that need to be continuous. The thick red line represents a discontinuity. Continuity analysis based on update chunks incurs only one-chunk-wide strip that will not be updated (area tinted red) whereas standard chunks incur unnecessarily two-chunks-wide strip. This means that update chunks allow tiles that are closer to the discontinuity to be transformed without transforming tiles that are directly neighboring with the discontinuity.

different each step, but they are selected only within the safe bounds of the update chunks are guaranteed not to overlap a discontinuity.

3.4 Fluid dynamics

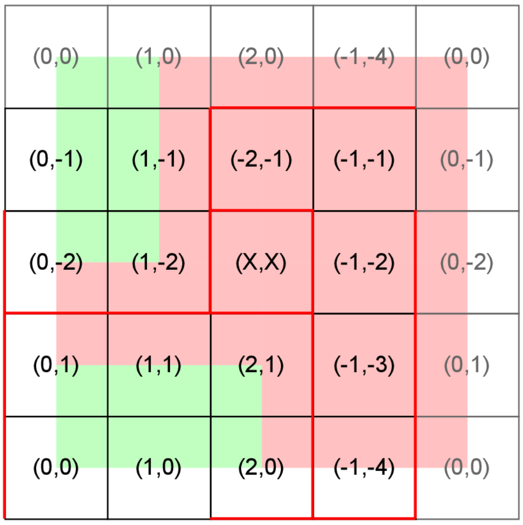
Fluid dynamics are, like tile transformations, another system that acts as a cellular automaton and modifies the world texture. Its main purpose is to move tiles (i.e. swap them with neighboring tiles). It used to simulate liquids, gases or even fire.

Subsection 3.4.1 explains, in more details, differences between fluid dynamics and tile transformations. Subsection 3.4.2 describes the basic unit used for parallel simulation of the dynamics. Subsection 3.4.3 explains how the basic units cooperate to update whole chunks.

3.4.1 Differences from tile transformations

Unlike transformations, fluid dynamics algorithm only modifies update chunks that are guaranteed to be continuous; i.e., the chunks that can been seen plus a fixed-size padding around it (further described in section 4.2.2). This means that the size of the area that it updates depends on the display resolution of the application but it is always much smaller than the area updated by tile transformations. Figure 3.11 illustrates the relation between the view and the area that is updated by fluid dynamics.

A major difference from the tile transformations is that much more care has to be taken in avoiding hazards of parallel store load/store operations. If two threads decided to move the same fluid, they would duplicate the fluid by each making their own copy in their target tile which would duplicate fluids. Or an opposite situation could happen when two threads decided to move two different fluids into the same tile - this would make fluids disappear.



(0,0)	(1,0)	(2,0)	(-1,-4)	(0,0)
(0,-1)	(1,-1)	(-2,-1)	(-1,-1)	(0,-1)
(0,-2)	(1,-2)	(X,X)	(-1,-2)	(0,-2)
(0,1)	(1,1)	(2,1)	(-1,-3)	(0,1)
(0,0)	(1,0)	(2,0)	(-1,-4)	(0,0)

Figure 3.10: Detection of discontinuities in a world texture. Each square represents a chunk with its global position written in the middle; (X, X) marks that no chunk has been activated there yet. The greyed squares are mirrored chunks from the other side. Red lines highlight the discontinuities among the chunks. Areas tinted green can be selected for transformations, areas tinted red cannot.

3.4.2 The basic unit for parallel simulation

The whole system, which simulates the fluids, builds on a premise that a fluid particle can move one tile, i.e. to its Moore neighbors, at a time. This premise creates a limit to the maximum velocity a fluid can move at. This maximum, however, is very high so it does not reduce credibility of the simulation.

Besides, the premise also avoids "tunnelling" - an undesired phenomenon of physics simulations in which moving objects teleport through an obstacle behind it. Since fluids can move only one tile at a time, each tile in its path is checked and it cannot teleport through a solid tile.

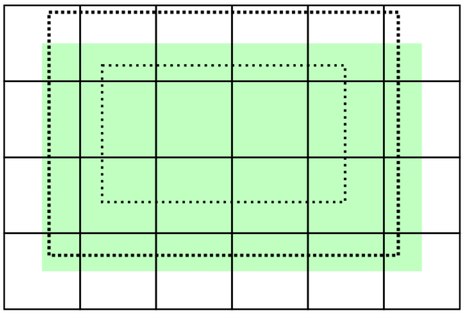
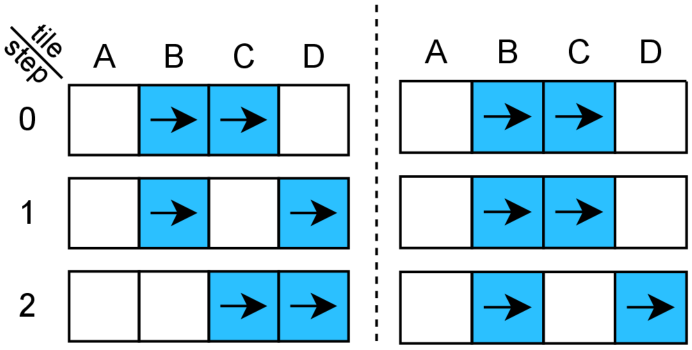


Figure 3.11: Chunks updated by fluid dynamics. The inner dotted rectangle represents the direct view into the world. The outer dotted rectangle represents the area that is required to be active and continuous (the view extended by fixed-size padding on all sides). All standard chunks that the outer rectangle overlaps must be loaded and continuous, the update chunks inside it (green) will be updated by fluid dynamics.

The premise that movement of fluids depends only on tiles in its Moore neighborhood also makes the simulation very local. It means that an independent unit can be constructed: the center tile, which is being updated, with its 8 neighbors. If the center tile (core) is a fluid, it is updated by moving it to one of the neighbors (receiver tiles). Even if one of receiver tiles contains a fluid, it is not moved. Only fluid in the core is moved. This update rule ensures that the next state of all tiles inside this 9-tiles unit depends solely on the tiles inside it. This means that such nonoverlapping 9-tile units can be updated in parallel.

These 9-tile units are not convenient though because if threads were updating such directly neighboring units, the total updated area would never have size that is a power of 2 (by definition, no power of 2 is evenly divisible by 3). Size that is a power of 2 is required to align the units over a whole chunk. Section 4.3.2 explains why size of chunks is a power of 2.

As a result, 16-tile square is used as the basic unit. It has 4 tiles in its core and 12 receiver tiles around it. For this unit, the order which the tiles in its core are updated in is crucial because simple trajectories of the fluids inside the core can cross so the first moving fluid may block others from moving. On the other hand, if the fluid is moving in the same direction as the order of updates, it can even move multiple times inside the core during a single simulation step. Figure 3.12 shows a simple situation where the trajectories cross.



(a) Tile C is updated first

(b) Tile B is updated first

Figure 3.12: The figures show two possible outcomes of a situation where two fluid tiles with the same velocity are next to each other inside a core (tiles B and C). Depending on the order, tile B either is or is not blocked by tile C.

Since there are 4 tiles in the core, there are 24 permutations of the order. An important aspect of this is each of the orders has a bias by updating tiles that move in the same direction as the order twice as often as tiles that move in the opposite direction. This bias manifests very clearly; e.g., by geometrical patterns among tiles that move closely in the same direction, or by asymmetrical spilling of fluids on flat horizontal surfaces.

The solution to avoid the bias is to randomly change the order for each unit. Biases of opposite update orders neutralize each other and the bias statistically disappears. Random changing of the order introduces stochasticity into the automaton which avoids the patterns and makes fluids flow more naturally.

3.4.3 Cooperation of basic units

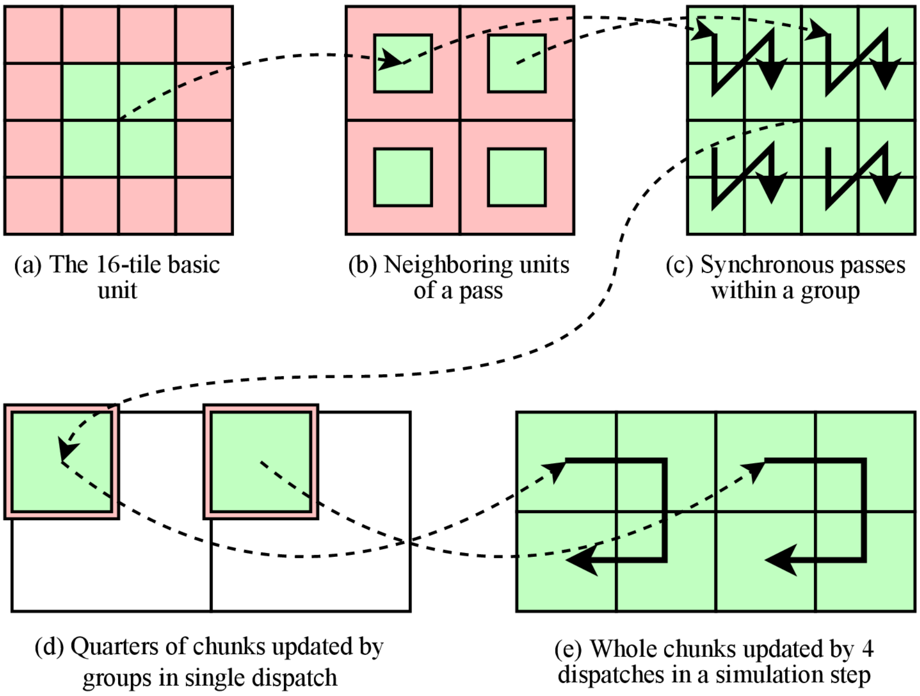
The 16-tile basic units, which update 4 tiles in their core, need to cover and update whole update chunk. Even if such units are laid out tightly over the chunk, only a quarter of the chunk is updated because of the receiver tiles around the cores. Therefore, it is required to update the remaining tiles with 3 more passes which cover the receiver tiles of previous passes. All four passes have to be performed during a single simulation step to update all tiles. Synchronization is required here because the next pass must not begin until the previous pass is finished. Otherwise, two basic units could overlap which would breach the original idea of separate units.

The four passes form a structure very similar to the tiles in the core of basic units. Again there are four elements in 2×2 grid, so again there is the question on which order they should be updated in. And the answer is also the same - any particular order would introduce bias, so the solution is to change the order after each pass. It must be ensured, however, that all threads within a group use the same order. If the threads used different orders, their basic units could overlap.

Synchronization of threads in compute shaders is limited as it blocks only threads within the same group. This is a problem because size of a single group is insufficient and its size has to be a shader-compile-time constant which does not suit the fact that the total area updated by fluid dynamics may change, so shader recompilation would be required.

To more easily utilize compute shaders, multiple groups have to be used. To evade the unavailability of synchronization across groups, synchronization between dispatch calls is utilized. A group within a dispatch call updates a square quarter of update chunk while other groups update the same quarter of other chunks. The three following dispatch calls within the simulation step update the remaining quarters of the chunks. This is, for the third time, the same idea with the same consequences on the order of updating of the quarters. Again it is important that all groups, within a dispatch call, update the same quarter of their respective chunks.

To summarize, there are 4 dispatches within one simulation step. These dispatch calls have to be separated by memory barriers. The number of groups dispatched directly corresponds to the number of update chunks that need to be updated. Each group updates one quarter of a chunk. The other quarters are updated by the other dispatches of the step. The order of quarters is global and changes after each step. A thread, within a group, updates one basic unit in each of 4 passes which need to be synchronized across the group by in-shader barriers. The order of passes needs to be the same within a group and changes after each pass. The order within cores of basic units may be completely arbitrary and it changes after each unit. Figure 3.13 visualizes the cooperation.



(a) The 16-tile basic unit

(b) Neighboring units of a pass

(c) Synchronous passes within a group

(d) Quarters of chunks updated by groups in single dispatch

(e) Whole chunks updated by 4 dispatches in a simulation step

Figure 3.13: Cooperation of basic units. Areas tinted green are being updated (they form core), areas tinted red must not be updated in parallel (they may receive tiles), respectively to each of the figures. Tiles in the core of basic unit in Figure (a) are updated in random order. Figure (b) shows area updated by 4 threads in their first pass. The receiver tiles of the first pass are updated in the following passes. Figure (c) shows that the order of passes has to be same for all of threads within the group. Figure (d) shows how a group updates a quarter of a chunk. Again, all groups need to update the same quarter of their chunks. Figure (e) shows how 4 dispatches together update whole chunks.

Chapter 4

Implementation

The goal of this chapter is to explain implementation details of the design decisions from previous chapter. This chapter also describes some issues, that had arisen during implementation, and solutions to these issues. Appendix A shows sample images of the implemented application.

The first section 4.1 describes implementation technologies used for the project. Section 4.2 explains how the world is rendered, especially calculation of shadows in a dynamic world where light is required to pass through tiles. Section 4.3 lists actual sizes of objects in the game, reasoning behind them and their consequences. Section 4.4 explains implementation details of generation on per-chunk basis. Section 4.5 explains the rules implemented within the tile transformation and fluid dynamics systems. Section 4.6 explains continuity analyzer, the system which detects discontinuities in the world texture.

4.1 Languages, libraries and tools

C++20 has been used as the implementation language of the application. C++ is a statically-typed, compiled language.

One of implementation goals has been to create an application that supports both Windows and Linux operating systems. CMake[1] is used to make the application multi-platform in terms of development. CMake is a tool that generates build system files for native build systems. This makes compilation on both platforms simpler and unified. The build was tested in conjunction with Visual Studio and GNU make.

Care also had to be taken to support compilers typical for the targeted platforms. Despite following the same language standard, different compilers have slightly different standard libraries and also issue warnings on different occasions. The goal has been to allow successful compilation without excessive number of warnings. MSVC and GCC compilers were tested.

To create an application with graphical user interface, it is required to open a graphic window and to detect user inputs (keyboard and mouse button presses, etc). This is a problem for multi-platform applications as different operating systems introduced different APIs to achieve that. The easiest and yet the least error prone solution to this problem is to use a hardware abstraction library, which deals with the various APIs. Simple DirectMedia Layer 2[2] library is used for this.

[1] https://cmake.org/

[2] https://www.libsdl.org/

OpenGL 4.6 is utilized to achieve hardware-accelerated rendering. It is used because it is supported on both operating systems. The OpenGL Extension Wrangler[3] is used to unify access to OpenGL functions across the platforms. The API was used with emphasis on Direct State Access. OpenGL Shading Language (GLSL) 4.60 is used as the shading language of the project.

OpenGL Mathematics[4], a C++ mathematics library based on GLSL specification, has also been utilized. It is used mainly to ease vector calculations. LodePNG[5] is a PNG encoder and decoder. It is used to load textures of the application and to load and save chunks of the world. Dear ImGui[6] is a graphical user interface library for C++. Main menu of the application is created with the library.

4.2 Rendering of the world

The goal of this section is to describe how the world is rendered. The first subsection 4.2.1 explains how tiles are rendered. The second subsection 4.2.2 explains how shadows are calculated and then rendered on top of tiles

4.2.1 Rendering of tiles

Rendering of tiles greatly benefits from two things: the regularity of the grid of tiles and the fact that the grid is stored in a single texture image on the GPU side. These two facts simplify and accelerate rendering of tiles significantly.

Tiles are rendered as small tightly packed squares. Thanks to the regularity of the grid, all visible tiles are drawn by a single instanced draw call. Equations 4.1 and 4.2 show how the number of instances is computed. This process effectively maps 2D coordinates to 1D instance index. Equation 4.3 shows how the instance's index is expanded back to 2D and how the tiles are aligned to the grid.

$$\boldsymbol{N} = \left\lceil \frac{\boldsymbol{V}}{\boldsymbol{T}} \right\rceil + \boldsymbol{1} \tag{4.1}$$

$$N = \boldsymbol{N}_x \times \boldsymbol{N}_y \tag{4.2}$$

$$\boldsymbol{P} = \left(i \mod \boldsymbol{N}_x, \left\lfloor \frac{i}{\boldsymbol{N}_x} \right\rfloor \right) \circ \boldsymbol{T} - (\boldsymbol{G} \mod \boldsymbol{T}) \tag{4.3}$$

Where:
$\boldsymbol{N}$ is the number of tiles rendered in x and y dimension;
$\boldsymbol{V}$ is viewport size in pixels; $\boldsymbol{T}$ is size of a tile in pixels;
N is the total instance count;
$\boldsymbol{P}$ is instances offset - the position of lower left corner of the tile; i is instance's index;
$\boldsymbol{G}$ is global position of the view in pixels.

Each tile, i.e. instance of the draw call, is rendered as 2-triangles strip that forms square with the size of tile. Figure 4.1 shows the placement pattern of instances that the above equations produce.

[3] http://glew.sourceforge.net/
[4] https://glm.g-truc.net/0.9.9/
[5] https://lodev.org/lodepng/
[6] https://github.com/ocornut/imgui

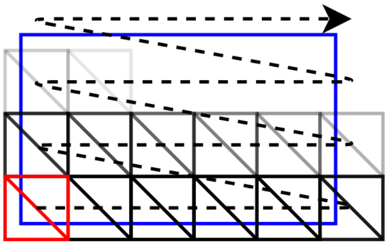


Figure 4.1: Placement of instances that draw tiles. Blue rectangle represents view into the world. Arrow shows how instance with increasing index are placed. The first instance is highlighted red. The figure also shows that the instances need to cover an area that is larger by one row and one column than the view because the tiles are not aligned with the view.

Besides computing its position, the vertex shader of the program that draws tiles has to fetch the tile that it is supposed to draw. The position of the texel that should be drawn is derived from the global position of the tile modulo size of the world texture.

With the 4 attributes fetched, UV coordinates into block-sprite atlas and wall-sprite atlas are computed. Type of the layer determines y coordinate while variant of the layer determines x coordinate. The fragment shader of the program then uses these two interpolated UV coordinates to fetch their color and immediately blends block color with wall color based on block's alpha. This becomes the final color of the tile which also may be transparent so it is blended with background color. This blending is done by rendering pipeline, not inside the shader.

One simple optimization can be used. When both block and wall layers are air, it would be unnecessary to rasterize the tile since all fragments would output full transparency, leaving the background color unchanged. To avoid these unnecessary fragments, the vertex shader forces the triangle to be clipped (by negating its homogeneous coordinate) when it detects this condition. Figure 4.2 shows a simplified flowchart of the vertex shader. This optimization is the most effective when the view is near terrain surface.

4.2.2 Rendering of shadows

Shadows are rendered as another layer on top of tiles. They are created by calculating illumination of each visible tile. The illumination is affected by tiles in the world texture and lights added by other objects of the game. The process can be described by the following steps:

1. Tiles in and around the view are converted to light sources and translucency factors. The results are stored in a secondary texture.
2. External light sources are added to the texture.
3. Illumination of each visible tile is calculated based on lights and translucency of tiles around it. The result of this step is a transparent color that directly represents the shadow over the tile.
4. The texture with shadows is rendered directly to screen as a layer above tile layer.

Firstly, each visible tile is converted to light source that it emits (RGB color and power of the light). Tiles that should not emit light are converted to a light of zero power. The

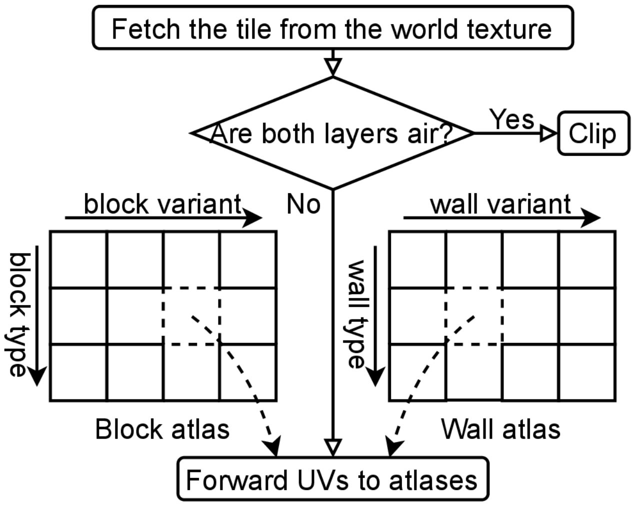


Figure 4.2: Simplified flowchart of the vertex shader that draws tiles. First it fetches the tile that it is supposed to draw. If it is an air tile, it forces the triangle to be clipped (all vertices of the triangle detect this jointly). Otherwise it computes the UV coordinates to block and wall atlases which are forwarded to fragment shader.

conversion is done through similar atlases that define color and power of light for each block and wall variant.

Each tile emits the light of its top layer - if its block is not air, it emits block's light. If the block is air, it emits light of the wall behind it. If the wall is air too, the tile emits background/sky light.

Apart from converting each tile to a light source, each tile is also converted to a translucency factor, a number that denotes how much light passes through it. Air tiles pass the most light, tiles with walls pass less light and tiles with blocks pass the least light.

External lights are added in the next step. Unlike lights produced by tiles, external lights are not centered at a tile. To produce the effect of a light shining from between tiles, the light has to be split between 4 tiles proportionally to the distance to each of the neighboring tiles. If a light is right in the middle of the four tiles, each gets one quarter of its power etc. This is an inverse process of interpolation as known point is distributed to nearest neighbors. The four distributed portions are added to the already present light sources of the neighboring tiles.

With the light-source and translucency textures prepared, illumination/shadow of each tile is calculated. The illumination is a sum of a fixed number of rotated rays that point towards the center tile. A ray starts away from the center and moves towards it while gradually adding visited lights in the way and decreasing it by the translucency factor of the visited tiles. Algorithm 2 explains the calculation in detail. It is desirable to filter both secondary textures with linear magnification filter to avoid artifacts in some directions of the rays.

The final shadow of the tile has the same color as sum of the lights that reach it. The shadow's transparency is inverse to square root of the power of the lights. The square root is used to create slower transition from light to dark. The calculated shadow is finally rendered on top tiles, again with linear magnification filter to smooth out the shadows.

The maximum range, which the rays move from, is an important parameter. Firstly, it determines the maximum distance any light can reach. Secondly, it greatly affects the

```
Algorithm 2: Computation of illumination of a tile
Function accumulate($l \rightarrow$ already accumulated light, $p \rightarrow$ tile to accumulate):
    return $(l + lightSource(p)) \cdot translucency(p)$;
Function tileShadow($p \rightarrow$ position of the center tile):
    $l \leftarrow (0, 0, 0, 0)$;
    for $d \leftarrow 0$; $d < 2\pi$; $d \overset{+}{\leftarrow} \frac{2\pi}{D}$ do              // For D directions
        $ray \leftarrow (0, 0, 0, 0)$ ;                       // Begin with light of zero power
        for $r \leftarrow r_{max}$; $r > 0$; $r \overset{-}{\leftarrow} 1$ do           // Ray moving towards the center
            $ray \leftarrow accumulate(ray, p + (\cos d, \sin d) \cdot r)$;
        $l \overset{+}{\leftarrow} ray$;                                         // Sum the rays
    $l \leftarrow accumulate(l, p)$;                                 // The center tile itself
    return $(l_r, l_g, l_b, 1 - \sqrt{l_a})$;                     // Shadow color of the tile
```

performance of the calculation. Most importantly, it determines the size of the area that has to be prepared for correct calculation of shadows because all tiles around the view, in up to the maximum distance, have to be converted too. Should they not be converted, a light right outside of the view would not be able to shine into the visible area.

The maximum range even determines the minimal required rectangle of chunks that need to be continuously loaded around the view in the world texture. This surely loaded area is also utilized by fluid dynamics as it only updates chunks inside this area. Figure 4.3 shows the difference between the view and the area that has to be converted.

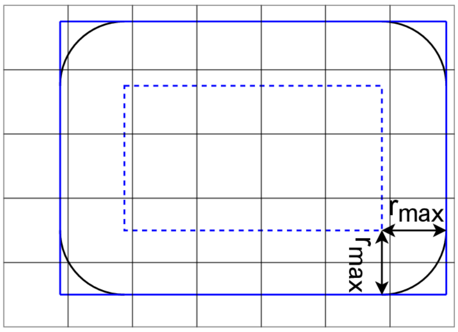


Figure 4.3: The area that is required to be loaded for correct rendering. Chunks that overlap the directly visible area (blue dashed rectangle) are not enough for correct calculation of shadows. Chunks that overlap area extended by light's maximum range in all directions (solid blue rectangle) need to be loaded too because light sources in it may shine into the visible area too.

4.3 Size of objects and its consequences

Concrete sizes of objects such as tiles and chunks have many consequences. Subsection 4.3.1 explains why the project switched from 16×16 to 4×4 pixels tiles and the consequences it had. Subsection 4.3.2 explains benefits of sizes that are powers of two.

4.3.1 Size of a tile

The project was originally designed to use tiles that are rendered as 16×16 pixel squares. Figure 4.4 shows an image of an early stage of the project that uses these tiles. When it came to liquid simulation, there was a decision to make: to simulate liquids in a system that is separate from tiles (Terraria, for example, uses this approach) or to simulate liquids as a special kind of blocks (approach used by Noita, for example). It has been quickly decided to use Noita's approach because it can easily be extended to simulate gases or even fire. It is also simpler to save such world because the whole state of the world is, instead of multiple structures, represented by a single grid of tiles.

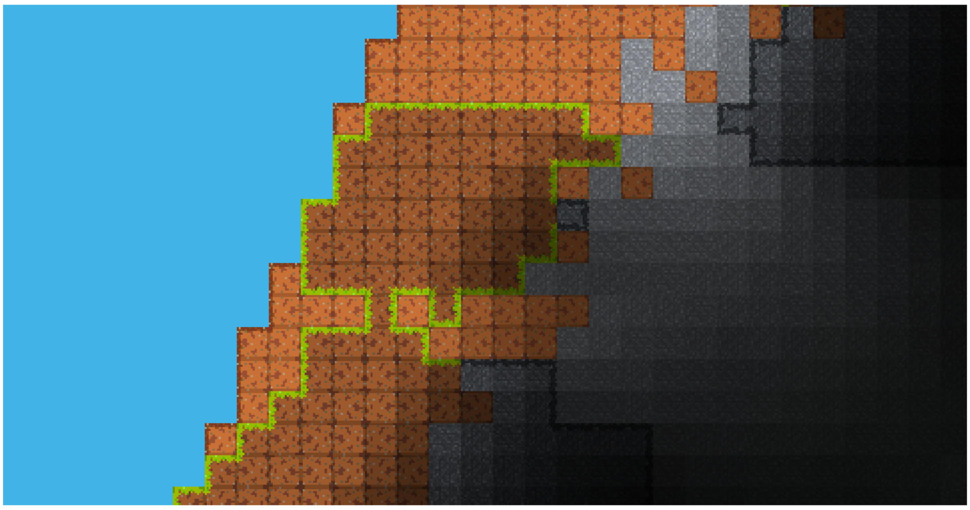


Figure 4.4: A sample image of an early stage of the project with 16×16 tiles. Tiles have later been reduced to 4×4 pixels to allow simulation of fluids as a type of tiles.

This, however, meant that tiles need to be smaller than 16×16 to make the fluids any credible. 4×4 tiles have been chosen as the best compromise between performance and credibility of the simulation. The consequences of this are greater than it seems.

Firstly, it is not really possible for each tile to have a sprite-like texture. 4×4 pixels is simply too small for any detail to fit. It turned out that it actually looks better when color of such tiles is constant. Variants are instead used to make areas made of the same tile more interesting.

Variants are also used to highlight borders between occupied tiles and air. Originally, each tile had selected its variant based on its Moore neighborhood to select the variant with correct sprite. Since borders drawn within the sprites have not been possible anymore, a different approach has to be used. This approach was already described in section 3.2.6.

Constant color of tiles has some positives too. Texture atlases for blocks and tiles can be very small since each sprite occupies a single texel. It also means that, when rendering the tiles, the vertex shader does not have forward UVs to the atlases. It can forward the color directly. This reduces $4 \times 4 = 16$ texture lookups in fragment shader to 4 lookups in vertex shader.

4×4 pixels tiles have severe impact on calculation of shadows. Since the calculation is done for each tile, its complexity increases 16 times compared to 16×16 tiles. This would halt the game completely. To avoid this, the calculation is actually done on 4×4-tiles units. The translucency and light of this unit is equal to average of the tiles it represents. This keeps the same computational complexity as the original tiles. For lonely sources of light, such as single drop of lava, it can be noticed that the light does not move as smoothly as its source.

4.3.2 More effective position conversions

A tile is 4×4 pixels. A chunk is 128×128 tiles. The world texture is 16×16 chunks. The fact that all objects have dimensions that are powers of 2 has a purpose.

Firstly, it allows effective allocation of both CPU and GPU memory. Secondly, dimensions that are powers of 2 allow more effective formulas to be used for conversions than the Formula 3.1. It allows replacing of modulo and multiplication operations by bitwise AND and bitwise shift operations respectively. Equation 4.4 shows the same formula rewritten to use bitwise operations.

$$\boldsymbol{P} = (\boldsymbol{G} \;\&\; \boldsymbol{W'}) << \boldsymbol{C'} \tag{4.4}$$

Where:
$\boldsymbol{P}$ is the resulting position of the chunk inside the world texture, measured in tiles/texels;
$\boldsymbol{G}$ is the global position of the chunk, measured in chunks;
$\boldsymbol{W'}$ is a constant bitwise mask computed as: $\boldsymbol{W'} = \boldsymbol{W} - 1$;
$\boldsymbol{C'}$ is a constant computed as $\boldsymbol{C'} = \log_2 \boldsymbol{C}$;
$\boldsymbol{W}$ is the size of the world texture, measured in chunks;
$\boldsymbol{C}$ is the size of a chunk, measured in tiles/texels.

The bitwise alternative is computed much faster than the original formula. It is because especially modulo is a slow operation. Speed is an important factor mainly in shaders where the conversion is performed very often. The alternative also avoids inconveniently defined modulo in C++ and GLSL which either leave the sign of the remainder to be undefined (GLSL [4]) or return negative remainder for negative dividends (C++11 and newer) which is the opposite than needed. Usage of bitwise operations, however, implies that this alternative formula is not usable in the floating point domain.

Similar formulas have been used throughout the project, e.g. conversion of position in pixels to position in chunks, computation of offset in tiles within a chunk from position in tiles, etc. All these conversions, where applicable, use bitwise operations.

4.4 Details of chunk generation

The whole process of chunk generation is intended to be accelerated by GPU, therefore the process is performed by 3 shader programs (one for each generation stage) that modify texture images which, in the end, contain the new chunk. Subsection 4.4.1 explains why chunks cannot be generated directly inside the world texture and how it is solved. Subsection 4.4.2 describes two implemented approaches to the types of shaders that generate the chunks.

4.4.1 Textures needed for generation

The original idea was to modify directly the world texture. That turned out to be impossible mainly because of the edge consolidation stage of the generation.

The usage of cellular automata (CA) in the generation requires valid values in the neighborhood of each tile. This is a problem for the tiles at the edges of the generated chunk. Undesired artifacts would appear when chunk was generated next to a discontinuity. Since the generated chunks are on the edge of the area explored by the player, there is always a discontinuity next to a generated chunk.

Furthermore, it is not possible to perform apply the rule of the CA synchronously without having two textures. It is required to read from one texture and write to other and then switch the roles in the next step. The world texture is too big to hold a copy only for the generation process. This issue is solved by having two separate textures used solely for generation.

The textures used for generation are actually bigger than a chunk. The intended chunk is positioned in the middle of the textures. The texels/tiles around it form a padding that absorbs the errors of the CA on the actual edges of the textures. The padding needs to be wide enough to prevent the errors from propagating to the actual chunk area. Should it not be wide enough, artifacts would be visible at the edges of generated chunks.

To safely perform a step of the CA, two such textures are used. For each step, one texture is read from while the other written to. Their roles swap after each step. This avoids the hazards of writing to and reading from a texture at the same time.

One more texture is used for the generation. It is also required for the consolidation of edges. If the CA determines that the given solid block should be turned into air, it is simple operation as air is just a constant. However if the CA determines that the given air block should be turned into a solid block, the question is which one it should be as there are many solid blocks. The third texture is used for this purpose.

The first stage writes to two texture simultaneously. Besides the main texture used by the CA, it writes to a material texture. This texture holds the specific blocks which the tiles should be turned into if the CA decides to do so. This means that the texture looks very similar to the main output texture. The only difference is that all tiles are solid instead of air inside caves. This texture is modified only by the first stage, other stages do not modify it.

The last stage, variant selection, is also a synchronous cellular automaton. Surprisingly, it does not require two textures. It is thanks to the fact that the transition rule modifies only variants, while the selection of the variant is based solely on types which are not allowed to change. As a result, it does not matter whether a neighbor has already been updated or not because the decisive type will be the same before and after the update.

The texels that represent the generated chunk are finally copied to the world texture as it is described by the chunk placement Equation 3.1. Figure 4.5 shows a diagram of all the textures used for generation and the data flow among them.

4.4.2 Shader type approach

Two distinct implementations (switchable at compile time) of the procedural generation were implemented:

- The first implementation uses **framebuffers** to control the output texture. The drawing is done by shader programs with vertex and fragment stages. The vertex stage is shared among all the shaders and it is a simple attribute-less shader that derives the output position from its vertex ID so that the resulting rectangle covers the whole output texture. The fragment stages do the main computation. The structure generation fragment shader has two output colors: one for the first generation texture

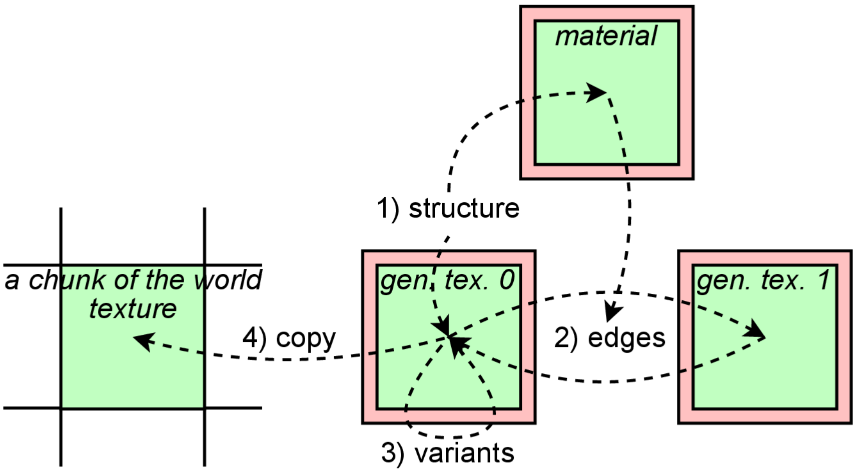


Figure 4.5: Stages of generation on the textures they work with. Fundamental structure outputs to a material texture and the first of generation textures. The cellular automaton (CA) that consolidates edges ping-pongs between two such textures, reading from one and writing to the other. Variant selection CA, thanks to its specific transition rule, does not require ping-ponging. The center of the final generation texture is copied to appropriate location inside the world texture.

and one for the material texture. Texture barriers are issued between steps of cellular automaton to ensure that updated texels are read in the next step.

- The other approach uses **compute shaders** and image load/store operations. Each thread calculates a single tile. Instead of overbinding the framebuffer a single uniform is used to choose which of the texture images is read from or written to. No in-group synchronization is needed but, similarly to previous approach, appropriate memory barriers have to be issued between dispatches to ensure that the updated texels are loaded in the next step of CA.

4.5 Rules of simulation

The goal of this section to explain the rules implemented inside the tile transformations and fluid dynamics systems. Subsection 4.5.1 explains rules of tile transformations. Subsection 4.5.2 explains rules of fluid dynamics.

4.5.1 Tile transformation rules

The tile transformations system imposes two requirements on the rules of its automaton: a thread should modify only the core tile (it should not modify neighbors of the core tile) and it should not read tiles that are farther than half of a chunk to avoid discontinuities.

A problem with definition of the transformation rules is that behavior of tiles may vary greatly. A tile could be transformed into various other tiles based on its neighbors, while another tile could not be transformed into anything. When viewed from the other side, a tile can be a result of many different transformations or none at all. Therefore, the representation of rules has to be general on one side to avoid extensive amount of unmanageable rules and specific on the other side to allow custom behavior of tiles. If the rules were not represented in a general way, the number of them would grow wildly and the system would be unmanageable. If the rules could not be specific, it would be impossible to simulate interesting transformations.

To make the rules general, each type of block and each type of wall is assigned a number of properties, represented as a bit field. A property/bit can have meaning such as “is burning” (set for lava, fire or smoke) or “is a grass” (set for all types of grass). The general properties allow, for example, grass to grow across the biomes - frozen grass from cold biome can grow to mud grass in rainforest.

Each type of block and each type of wall also have defined an ordered set of rules that can potentially be applied to them. The set is represented as a range inside global array of rules. The order of the applicable rules in the set defines their priority - the first rule whose requirements are met is applied. Empty set (a range of zero length) is assigned to types that cannot be transformed in any way. Blocks and walls each have they own separate rules. A rule is defined by four parameters:

- The properties of neighbors that are required to perform the transformation. For example, dirt requires one of its neighbors to have “is a grass” property to grow into a grass tile.
- The properties that none of the neighbors must have. Continuing the example, no neighbor must have “is burning” property for grass to grow.
- Modifiers of the rule. A modifier is also a bit field that can specify that the tile must be on an edge to transform. As an example, this would be true for transformation of dirt to grass as grass is supposed to grow on edges only. Another modifier specifies whether the properties can be mixed with properties of the other layer. If this modifier is set, properties of block and walls are mixed before comparing them to the required/forbidden properties of the rule. This controls, for example, whether grass can grow from wall to block or vice-versa. If it is set, grass can grow through the layers. If it is not set, grass on the same layer is required.
- The type of block/wall that the tile is transformed into if all the requirements are met.

The actions of the automaton can be described as:

1. Search 5×5 circle neighborhood of the core cell. Combine properties of all the neighbors in their respective layers.
2. Do the following for block layer and wall layer:
   (a) Based on the type of the core, acquire a set of applicable rules.
   (b) Iterate over the set in descending priority, transform the tile according to the first rule whose requirements are met. The type remains the same if no rule can be applied.
   (c) Select new variant for the layer according to the same rule which variant selection stage of the generation uses.
3. Store the transformed tile at the same position.

Other than growth of various types of grass, this system is also used to burn the grass when a burning tile is nearby. It is also used to simulate an ever-spreading biome that grows across all other biomes. This is inspired by Terraria’s biomes.

4.5.2 Fluid dynamics rules

The fluid dynamics system imposes more rigid requirements on rules of its automaton: a rule may only read from its Moore neighbors. On the other side, it may write to the neighbors too.

For purpose of the simulation, each type of fluid is assigned a number of parameters:

- The primary move direction - up for gases, down for liquids.
- Probability of moving in the primary direction. Gases, for example, generally have lower probability of moving up than liquids of moving down so that gases slowly ascend while liquids drop quickly.
- Fluidity, represented as a probability of moving sideways. Lava, for example, has very low fluidity so it spills slowly.
- A simple conversion rule that is used when a specific block is touched. This system is much simpler than tile transformations. Each fluid is assigned a single block that triggers its conversion when they touch. As an example, lava triggers conversion of water to steam when they touch.
- A rule for random conversion. This system is also very simple. It is used to model evaporation (water is seldom turned into steam), precipitation (steam is randomly turned back into water), conversion of fire to smoke, etc.

All fluids hold one bit of information in their variant - leaning side bit. This bit denotes whether the fluid should move left or right when the automaton tries to move it horizontally. The leaning side of a fluid changes once the fluid cannot move in the direction anymore. This can be viewed as a bounce of the fluid to the other side. Figure 4.6 visualizes the neighbors that a fluid may move to. The actions of the fluid automaton can be described as:

1. Run primary move probability test. If it fails, the fluid cannot move in the primary direction or diagonally.
2. Run fluidity test. If it fails, the fluid cannot move in the leaning direction or diagonally.
3. Try to move in one of the remaining directions, in this order of priority: primary direction, diagonal direction, leaning direction.
4. If the block which triggers conversion of the fluid is encountered, convert the fluid and stop.
5. If the fluid can move, move it and stop.
6. If could not move in any direction:
   (a) If random conversion test succeeds, convert the fluid.
   (b) If tried to move sideways, invert the leaning bit - the fluid bounces away from the obstacle.

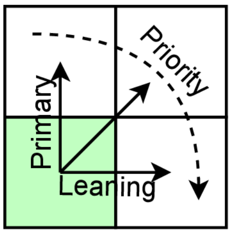


Figure 4.6: A fluid particle can move in three directions in the following order of priority: vertically (defined by the type of the fluid), diagonally and horizontally (defined by its leaning bit). A fluid does not try to move in every direction every step. Based on the type of the fluid, there are probabilities to skip some directions. As a result different fluids move differently.

By modification of the parameters, 5 fluids are implemented: water, lava, steam, fire and smoke. The conversion rules among them simulate related phenomena - water extinguishes fire and smoke, water also solidifies lava, etc.

A completely separate rule that allows a fluid to move in all directions by storing its velocity in its variant was tested. However, it proved to be unstable because 8 bits are simply too few store the velocity accurately.

4.6 Continuity analyzer

The world texture has to be analyzed to determine which of its update chunks do not overlap a discontinuity and thus can be can be selected for tile transformations. The system which analyzes continuity of the texture uses a shader storage buffer and a compute shader that manipulates it. The storage buffer contains three things:

- An array of global positions of chunks inside the world texture. Since the size of the world texture is a compile-time constant (16×16 chunks), size of the array is a compile-time constant too (256). Each position in this array corresponds to a slot inside the world texture. When a new world is loaded, the array is initialized with integer maximum value as a special constant which denotes that no chunk is loaded at the location yet.
- An array of positions of update chunks that can be updated. This array has to be large enough to hold the maximum number of update chunks that can be updated each step. Since there must be at least one discontinuity in each row and column, the size of this array ($15 \times 15 = 225$) is smaller than the number of chunks inside the world texture.
- A 3-dimensional vector used for indirect dispatch of tile transformation groups. The x component represents the number of valid positions in previous array. The y and z components are always 1.

The group size of the shader that analyzes the discontinuities matches the number of slots inside the world texture (16×16×1). There is always one group dispatched. Algorithm 3 explains what the shader does.

The view, which triggers loading of chunks, does not trigger loading of chunks if it does not move. If it does move, it triggers multiple chunks to be loaded in the same step (the

Algorithm 3: Detection of discontinuities inside the world texture

```
Function areNeighborsContinuous(p → chunk to test, o → offset to neighbor):
    $main \leftarrow globalPosition(p)$ ;        // Global position of chunk at slot p
    $neighbor \leftarrow globalPosition(p + o)$;
    if $(main + o) = neighbor$ then return true;
    else return false;
Function isUpdateChunkContinuous(p → the chunk to test continuity of):
    $left \leftarrow areNeighborsContinuous(p, (0, 1))$ ;      // Continuity of left edge
    $right \leftarrow areNeighborsContinuous(p + (1, 0), (0, 1))$;
    $bottom \leftarrow areNeighborsContinuous(p, (1, 0))$;
    if $left \wedge right \wedge bottom$ then return true; // The fourth edge is implicit
    else return false;
Function continuityAnalyzer(ID → identifier of the thread):
    if $ID = (0, 0)$ then
        $c \leftarrow 0$;
    $barrier()$;
    if isUpdateChunkContinuous(ID) then
        $i \leftarrow atomicAdd(c, 1)$ ;  // Serialize the chunks by an atomic counter
        $chunkPosition \leftarrow ID \circ CHUNK_{SIZE}$;
        $updateChunkPosition[i] \leftarrow chunkPosition + \frac{CHUNK_{SIZE}}{2}$;
    $barrier()$;
    if $ID = (0, 0)$ then
        $dispatchSize \leftarrow (c, 1, 1)$ ;  // Dispatch size for tile transformations
```

loaded chunks form a strip along a side of the view). The continuity analyzer is dispatched once after the last chunk of the batch.

Chapter 5

Performance

Performance of the application was tested on two computers. A laptop with Intel i5-8300H CPU, Nvidia GeForce GTX 1050 GPU, running Windows 10 (PC1) and older desktop computer with Intel i3-4160 CPU, NVIDIA GeForce GTX 750, running Ubuntu 20.04 (PC2). All test were done with resolution 1920×1080. Section 5.1 compares speed of two world generators. Section 5.2 analyzes the most expensive actions of the simulation.

5.1 Comparison of generation approaches

The two approaches of generation (section 4.4.2) have been compared to see which approach is faster. Since the view almost always triggers generation of multiple chunks in a batch, the approaches also have been compared on batches instead of single chunks. It is measured how long it takes to generate such batches.

The comparison is done on newly created world with a random seed. The view inside the world moves along the horizon at a constant speed instead of moving based on user's input. This periodically triggers a vertical batch of chunks to be generated. With viewport height 1080 pixels, 5 chunks are generated in a batch. The first 5 batches are not included in the measurement because they generate more chunks.

`glFinish`, which waits for all OpenGL commands to be finished, was inserted after the last chunk of a batch to ensure that the generation has truly finished and was not just queued. Maximum frame rate was limited to 300 frames per second (FPS) during the tests which is the same as for the main application. Figure 5.1 shows a histogram of 100 batches.

Surprisingly, the more performant computer (PC1) generated chunks with longer delays than the older computer (PC2). This is despite the fact that PC1 could run the tests at double frame rate (roughly 950 FPS) compared to PC2 (roughly 400 FPS) when the FPS limit was disabled. However, the limit had negligible impact on the delay of chunk generation.

Despite both approaches having a great variance in the delay, on average, generation by compute shaders proved to be faster than generation by standard pipeline, on both PCs. Compute shaders generate chunks in roughly 10 % less time than the other approach. As a result, compute shaders are used as the default generation method but the approaches still can be switched at compile time by a macro.

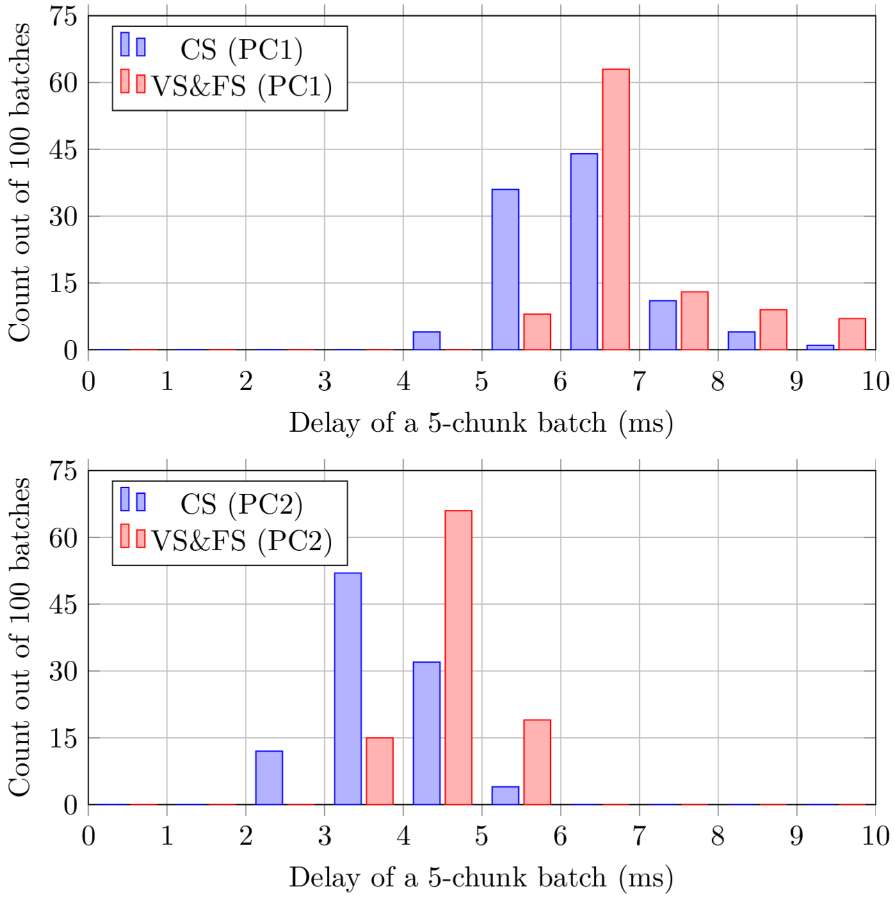


Figure 5.1: It was measured how long it takes to generate a 5-chunk batch by compute shaders with image load/store (CS) and by vertex shader, fragment shaders and framebuffers (VS&FS). The subfigures show histograms with delays of 100 such batches on both tested PCs.

5.2 Simulation step analysis

On PC1, overall performance of the application was analysed by Visual Studio's profiling tools. Firstly, it showed that the application is mostly GPU-bound (92 % utilization). Nearly 50 % of CPU-time was spent waiting for the main framebuffers to swap (swapping requires all queued rendering to be finished). GPU-boundedness is no surprise since not just rendering but all parts of the simulation are done by GPU.

It also showed that second most expensive operation (about 20 % of CPU-time) of a simulation step is download of updated position of the player to CPU memory. The new position is needed inside CPU memory to determine whether new chunks need to be activated. Activation of a chunk may require reading of the chunk from file system so this operation cannot be done by GPU. To void the expensive synchronization between GPU and CPU, pixel buffers could be utilized but it was not implemented because the game runs at 400 FPS even on the older PC so it is not needed at all.

Chapter 6

Conclusion

The goal of this thesis was to implement two-dimensional procedural generator and simulator of a gaming world. The world is represented by a seemingly-endless grid of small tiles. The illusion of endless world is created by dividing the world into chunks and simulating only those chunks that are nearby to the player.

The generator creates a world with horizon and underground caves below. The horizon is composed of nine biomes which affect all parameters of the horizon. There are two types of caves and lava pools deep underground.

The simulation of the world is done by two complementary systems. One moves tiles to simulate flow of fluids. Several different fluids such as water, lava or even fire are simulated by this system. The other system simulates slow processes such as growth of grass. The goal of this system is to update an area that is as huge as possible to suppress a feeling of the world being static right outside of the view. The simulation can also be intervened by the player. Rendering of the world includes a simple lighting system.

The implemented application is designed to run on Windows and Linux operating systems. It uses hardware acceleration for generation as well as simulation of the world. It also allows saving and loading of worlds to disk.

There are many ways the project could be expanded, especially in regards to flora. If the generator produced terrain with trees and bushes, it could also be very interesting to simulate growth or even spread of the fauna across the horizon. If weather phenomena, such as rain or wind, were simulated, they could affect the flora too. All mentioned ideas would probably need additional structures, other than the grid of tiles, to represent the world because such simulation probably would not be feasible by cellular automata. Calculation of illumination could also be improved to allow more accurate shadows to be rendered.

Bibliography

[1] Bailey, M. *OpenGL Compute Shaders*. Oregon State University, 2012. Available at: https://media.siggraph.org/education/conference/S2012_Materials/ComputeShader_1pp.pdf.

[2] Devlin, J. and Schuster, M. Probabilistic Cellular Automata for Granular Media in Video Games. *The Computer Games Journal*. june 2021, vol. 10. DOI: 10.1007/s40869-020-00122-4. Available at: https://arxiv.org/abs/2008.06341.

[3] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K. and Worley, S. *Texturing and Modeling, Third Edition: A Procedural Approach (The Morgan Kaufmann Series in Computer Graphics)*. 3rd ed. Morgan Kaufmann, 2002. ISBN 978-1558608481.

[4] Kessenich, J. *The OpenGL® Shading Language*. The Khronos Group Inc, 2019. 123 p. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[5] Mark Segal, K. A. *The OpenGL® Graphics System: A Specification*. The Khronos Group Inc, 2019. 33-35 p. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[6] Nystrom, R. *Game Programming Patterns*. 1st ed. Genever Benning, 2014. ISBN 978-0990582908. Available at: https://gameprogrammingpatterns.com/contents.html.

[7] Perlin, K. Chapter 2 Noise Hardware. In: SIGGRAPH 2002. Jul 2002. Real-Time Shading Languages. Available at: https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf.

[8] Wolfram, S. Statistical mechanics of cellular automata. *Reviews of Modern Physics*. American Physical Society. Jul 1983, vol. 55, p. 601–607.

Appendix A

Sample images of the virtual world

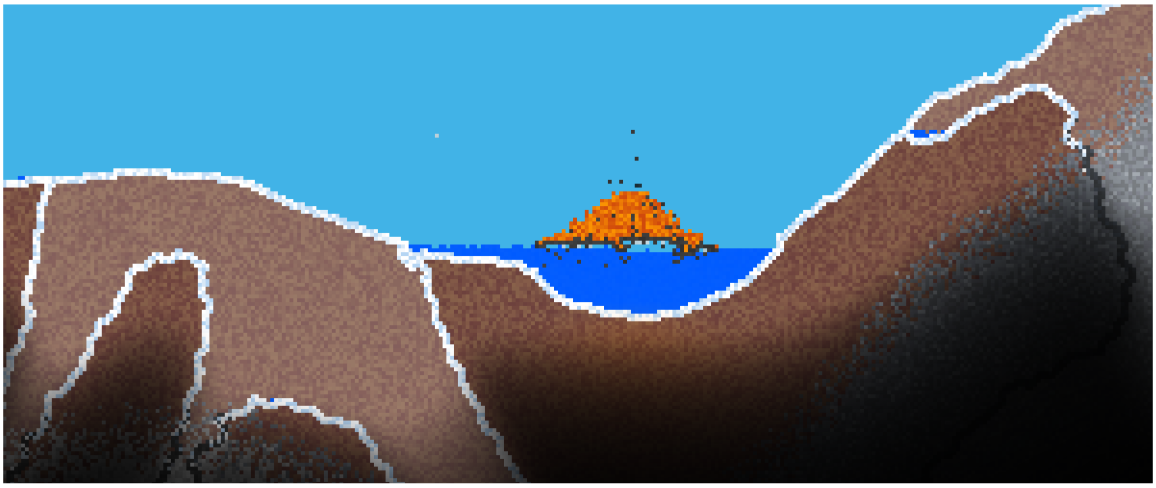


Figure A.1: Partially solidified lava dropped into a water pond inside a snowy biome.

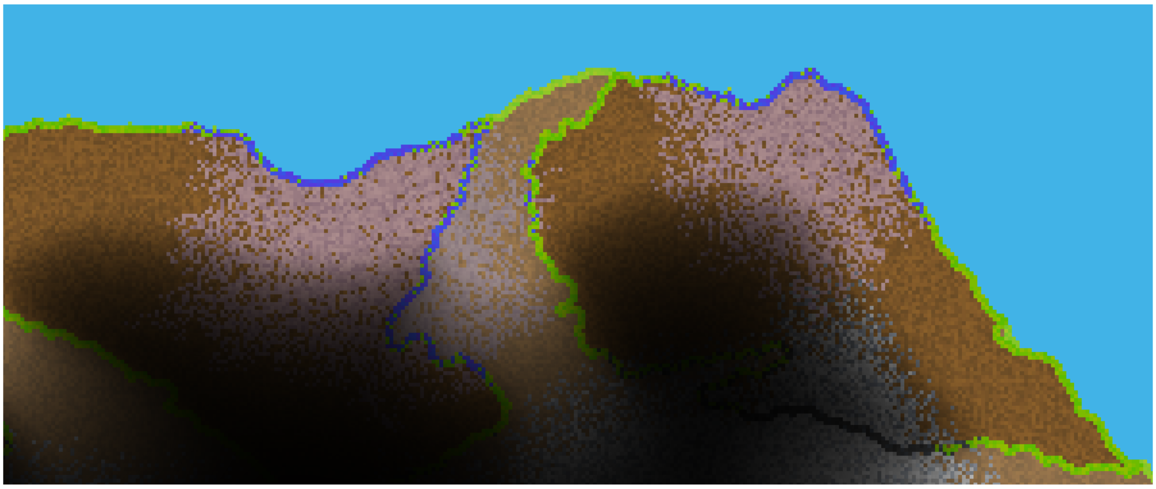


Figure A.2: An artificial biome spreading across a muddy biome.

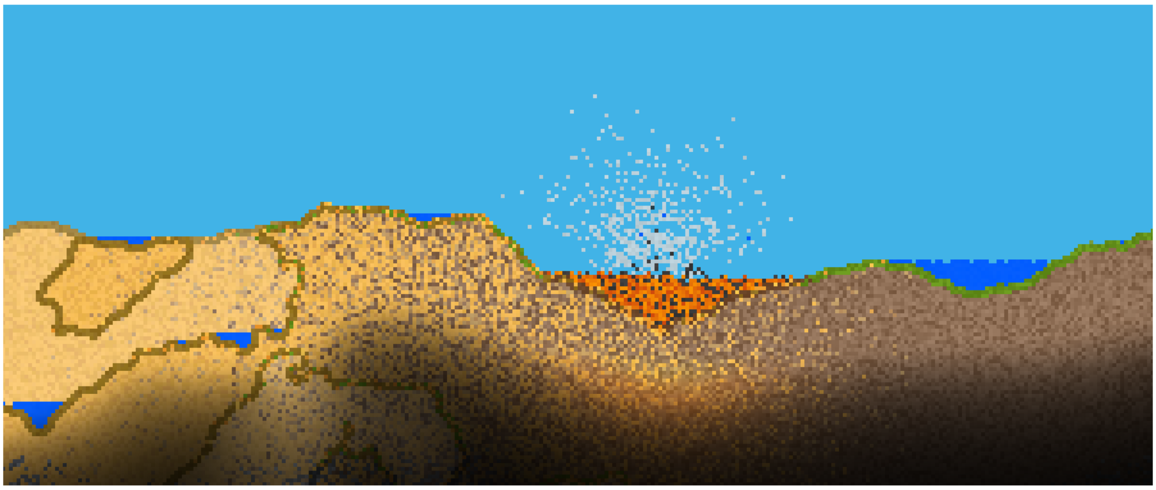


Figure A.3: Evaporation of water dropped into lava and ponds created by precipitation of the steam.

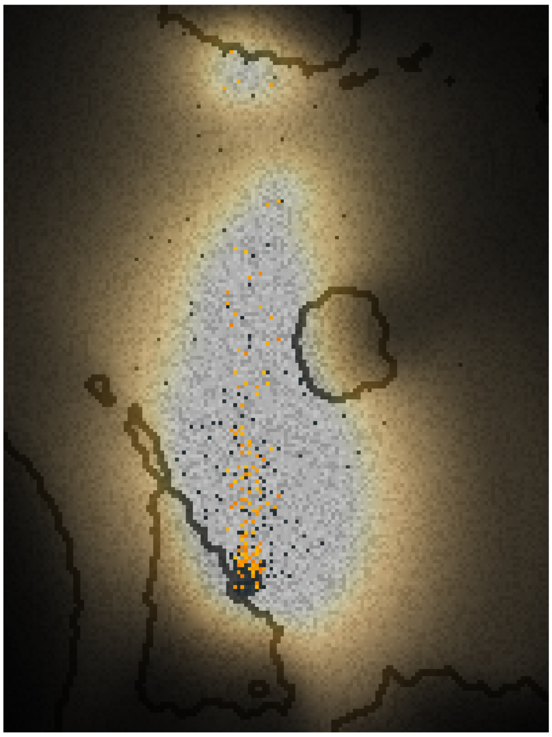


Figure A.4: Fire burning inside a cave.

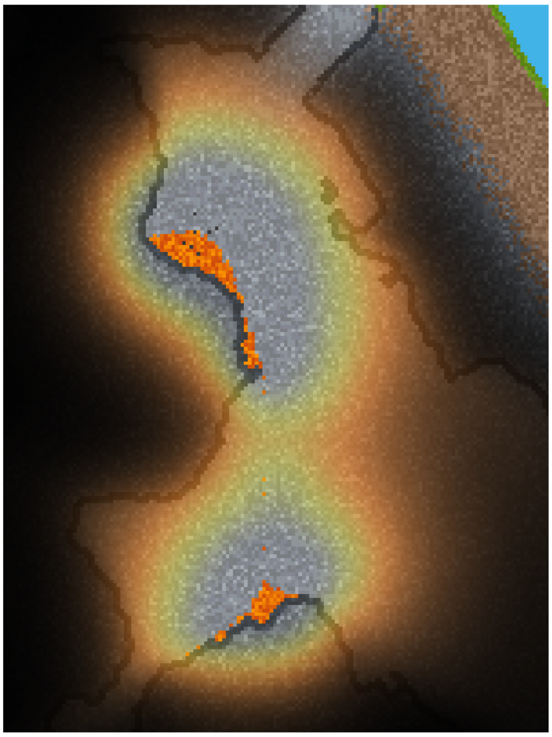


Figure A.5: Lava flowing beneath the horizon.