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

2D GAME VEGETATION GENERATION AND SIMULATION ON GPU

GENEROVÁNÍ A SIMULACE 2D HERNÍ VEGETACE NA GPU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ DUBSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ CHLUBNA**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

153549

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Dubský Tomáš, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Graphics and Interaction
Title:	**2D Game Vegetation Generation and Simulation on GPU**
Category:	Computer Graphics
Academic year:	2023/24

Assignment:

1. Study the basics of the rendering pipeline and compute shaders in Vulkan API.
2. Learn about methods suitable for procedural generation and simulation of the 2D vegetation.
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.

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 semestral defence:
Items 1-3, 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, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	9.11.2023

Abstract

The goal of this thesis is to implement hardware accelerated procedural generation and simulation of vegetation set inside a two-dimensional gaming world. By the means of L-systems, several species of vegetation, represented by interconnected flexible branch segments, are generated. The plants sway in the wind, grow leaves or may catch fire. The vegetation is visualized by square tiles to fit the graphical style of the surrounding world which is represented by a regular grid of tiles.

Abstrakt

Cílem práce je implementace hardwarově akcelerovaného procedurálního generování a simulace vegetace zasazené do dvojdimenzionálního herního světa. Pomocí L-systémů je generováno několik druhů rostlin. Vegetace je tvořena propojenými segmenty větví, které se ohýbají ve větru, rostou na nich listy a mohou hořet. Vegetace je zobrazována pomocí malých dlaždic, aby zapadla do okolního světa, který je tvořen pravidelnou mřížkou dlaždic.

Keywords

procedural vegetation, vegetation simulation, L-system, tile-based vegetation, particle system, cellular automaton, tile-based game, GPU programming

Klíčová slova

procedurální vegetace, simulace vegetace, L-systém, vegetace dělená na dlaždice, částicový systém, celulární automat, hra dělená na dlaždice, programování GPU

Reference

DUBSKÝ, Tomáš. *2D Game Vegetation Generation and Simulation on GPU*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Chlubna

Rozšířený abstrakt

Cílem této práce je implementace procedurálního generování a simulace vegetace zasazené do dvojdimenzionálního herního světa. Tato práce navazuje na bakalářskou práci, která se zabývala generováním herního světa v pohledu z boku a simulací kapalin v něm. Tento svět byl ovšem bez vegetace pustý. Cílem této práce je přidání vegetace do původního světa.

Hlavní technickou výzvou práce je fakt, že okolní svět je popsán pravidelnou mřížkou dlaždic, tedy rastrem. Vegetace se naopak skládá ze vzájemně propojených segmentů větví, což je vektorová reprezentace. Cílem práce ovšem je, aby vegetace vypadala, že je tvořena dlaždicemi jako zbytek světa.

Vegetace a svět jsou tedy při simulaci propojeny pomocí rasterizace. Vegetace je v každém kroku rasterizována do mřížky dlaždic světa. Poté proběhne simulace v mřížce světa pomocí celulárních automatů, která simuluje proudění kapalin a plynů. Nakonec je vegetace pomocí rasterizace odebrána ze světa, aby v následujícím kroku mohla být přidána na nové pozici. Při odebírání vegetace ze světa simulace ukládá, jak se změnily dlaždice rasterizované na začátku kroku. Díky tomu celulární simulace může ovlivnit vzhled vegetace.

Struktura segmentů větví je generováná pomocí L-systémů na grafické kartě. Do herního světa se generuje, v závislosti na okolním biomu, několik druhů stromů a keřů od kaktusů v poušti, přes listnaté stromy mírného pásu až po jehličnany chladného pásu.

Cílem práce také je, aby generovaný svět byl zdánlivě nekonečný. To je řešeno tak, že svět je rozdělen na čtvercové části a je generován postupně, když se hráč přiblíží k neprozkoumaným částem. Části, od kterých se vzdálí, se naopak odebírají z paměti. Všechny části vygenerovanáho světa, včetně vegetace v nich, se nakonec uloží do souboru, ze kterého lze svět obnovit.

Na struktuře větví je simulováno působení externích sil i vnitřních sil, které vegetaci udržují v přirozeném tvaru. Díky tomu je simulováno pohupování vegetace ve větru. Ačkoliv simulace pracuje se segmenty jako s pevnými tyčemi, při rasterizaci jsou využívány Bézierovy křivky, díky čemuž stačí simulovat méně segmentů při zachování přirozeně zakroucených větví.

Propojení vegetace s celulární simulací umožňuje, například, simulovat hoření vegetace. Listí kolem větví vzniká kombinací rasterizace a celulární simulace. Při rasterizaci větví je rasterizováno malé množství listí, které se rozšíří na chomáče listí díky celulární simulaci. Všechny výše zmíněné typy simulace jsou prováděny na grafické kartě.

Práce se věnuje efektivní implementaci výše zmíněného i vylepšení původního projektu, například zlepšením výpočtu dynamického osvětlení světa, a nebo přechodem na moderní nízkoúrovňové grafické rozhraní. Při implementaci byl brán ohled i na to, aby aplikace fungovala pod operačními systémy rodin Windows i Linux.

Efektivita řešení byla zhodnocena porovnáním s původní verzí projetu. Měření ukázalo, že díky optimalizacím aplikace zvládá vykreslit o více než 50 % snímků za sekundu více než její původní verze. Podrobná analýza aplikace také ukázala, které faktory limitují výkon současné verze aplikace.

2D Game Vegetation Generation and Simulation on GPU

Declaration

I hereby declare that this Master'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 15, 2024

Acknowledgements

I would like to express deep gratitude to Mr. Tomáš Chlubna for letting me continue on this project under his supervision and for providing very useful feedback throughout the thesis.

I am legally obliged to thank to my friends, Tonda Moravec and Vojta Struhár, for letting me disturb them from playing to test the project on their PCs.

Shout out to them.

Contents

Chapter 1

Introduction

The goal of this thesis is to add vegetation to a game demo. The project, prior to this thesis, generated endless two-dimensional world from side view and simulated flow of fluids in it. The generated world included several biomes spread across the horizon and caves below. The player can interact with the world, too. Chapter 2 gives a more elaborate summary of the project prior to this thesis.

Visualization of the virtual world includes dynamic lights and shadows. Chapter 3 explains an improvement to the algorithm which calculates the lighting. It also describes transition of the project from an older graphics programming interface to newer and more low-level one.

The transition to the new graphic interface is described early in the thesis because it has great impact on implementation of vegetation. Vegetation, described in Chapter 4, is the main goal of the thesis because the original world seemed barren and it was hard to differentiate between the biomes. The chapter explains how the vegetation is generated and simulated to interact with the rest of the world. There are multiple species of vegetation distributed according to the biome on horizon. The generated vegetation sways in the wind, deciduous species drop leaves, it can catch fire and burn. Both generation and simulation of vegetation is accelerated on GPU.

Chapter 5 analyzes performance of the simulation and visualization of the project, including the newly added vegetation. Chapter 6 concludes the thesis by summarizing the achieved results and proposing future improvements.

Chapter 2

Summary of the project prior to this thesis

This project started as a bachelor's thesis [3] whose goal was to create a game demo. This chapter briefly describes the state of the project before its extension presented in this thesis.

The demo features a 2D sandbox world seen from side view. The procedurally generated world features several biomes spread across horizon and caves below it. Some natural phenomena are simulated in the world — examples are liquids, gases, fire, and grass growth. The player can move within the world and modify it, i.e. affect its simulation. Figure 2.1 shows an example screenshot of the game.

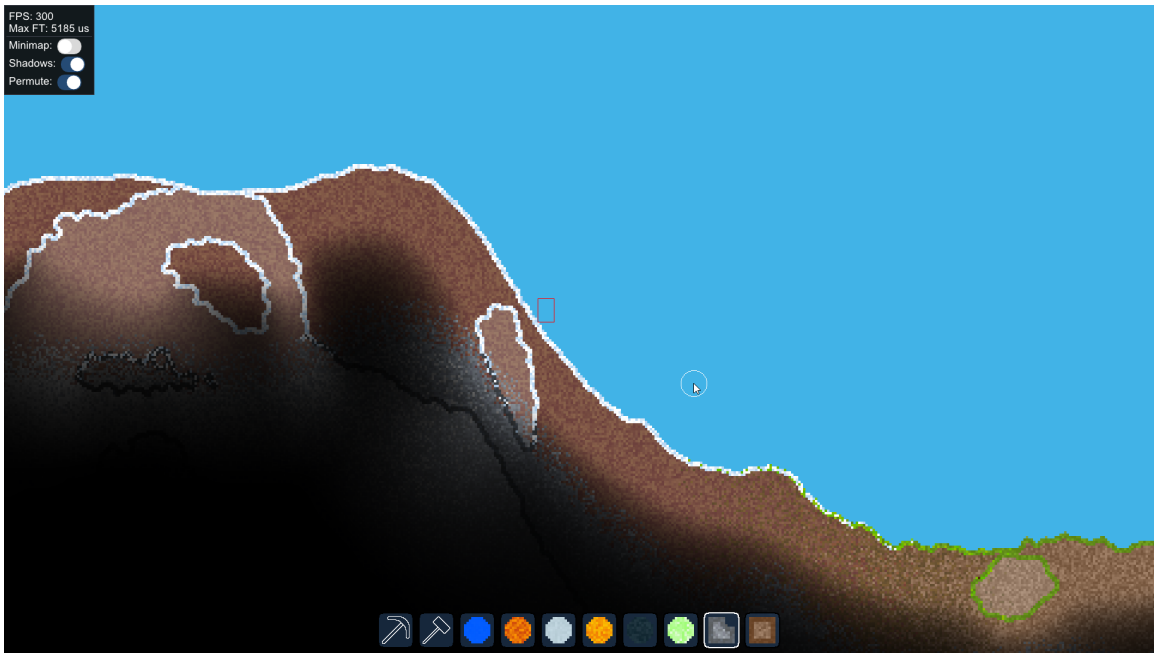


Figure 2.1: An example screenshot of the game shows the player (red rectangle) standing on a hillside. A toolbar with stone wall selected can be seen at the bottom of the screen. If the player clicked, the wall would fill the white circle around the cursor. The player can also break tiles of the world by the pickaxe and hammer tools.

Terraria[1] was the most influential inspiration of this project. Terraria is also a 2D tile-based sandbox game, albeit focused on fighting enemies which is not the case in this project. From visual perspective, the biggest difference is that Terraria uses 16×16 pixels large tiles where each tile has a small sprite texture. This project uses 4×4 pixels large tiles with solid color, this makes the project visually similar to Noita[2] which was another inspiration.

Size of tiles has consequences not only on visualization but simulation too. For example, Terraria uses a separate system to represent water in the game. Noita and this project, on the other hand, have tiles small enough that fluids are represented as a special type of tile that moves.

The first Section 2.1 explains how the game world is represented in memory, especially how it is hidden from the player that only a limited part of the seemingly endless world is held in memory. This has consequences on all other aspects of the game, including generation and simulation of the world. Section 2.2 describes the process of procedural generation used in the project. Section 2.3 describes how fluids and slow processes such as grass growth have been simulated in the generated world. The last Section 2.4 overviews languages and libraries used to implement the project.

2.1 Memory representation of a 2D tile-based endless world

Very early in the project, it was decided to represent the game world as a grid of small tiles. Each tile has two layers: a block (which limits player's movement) and a wall (which has no effect on player's movement and is drawn behind him). As Figure 2.2 shows, each layer of a tile is defined by two values - its type and 'variant' which stores additional information about the block/wall. The interpretation of the variant value depends on the type of layer, e.g. for fluids it is used to store the direction that the fluid is flowing in.

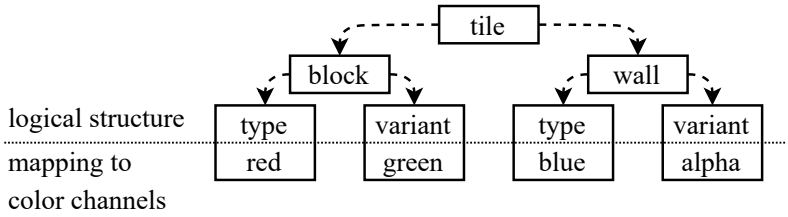


Figure 2.2: A tile has two layers, block and wall. Both layers have two attributes: a type and a variant. Interpretation of the variant value depends on the type of the layer. When tiles are stored in textures, the four attributes are mapped to red, green, blue and alpha color channels respectively. Figure is adapted from bachelor's thesis [3].

One of the defining goals of the project was to create an illusion that the world is endless. The illusion is created by dividing the grid of tiles to square slices of fixed size called chunks. Subsequently, only chunks that are near the player are generated and simulated. Very far chunks are not necessary to be available in memory. This is a common procedure used in many voxel-based games with endless worlds.

A crucial aspect of the implementation of the project is that GPU is employed to do computationally demanding tasks. CPU performs the orchestration when and where to

[1] Available at: https://store.steampowered.com/app/105600/Terraria/
[2] Available at: https://store.steampowered.com/app/881100/Noita/

generate and simulate a chunk but the actual work is done in compute shaders. Compute shaders are programs run on GPU that perform general purpose computation.

To allow the world to be manipulated efficiently by shaders, two things are required: to store the tile in video memory inside a texture on the GPU (so that it can be quickly accessed) and to store all chunks in exactly one texture as continuously as possible. If each chunk had its own texture, it would be very inconvenient to simulate anything on the boundaries between chunks and it would also complicate rendering of the tiles.

So this puts contradictory requirements on the memory representation - all tiles have to be in one big texture, called the *world texture*. However, it is also needed to dynamically load and unload chunks as the player moves in the world.

As a result, a system that orchestrates activation of nearby chunks and deactivation of far ones was developed. Activation of a chunks means that its tiles/texels are copied into the world texture. Equation 2.1 shows that the system places chunks into the texture based on their global position in the world modulo the size of the world texture. Figure 2.3 visualizes the pattern that this equation produces.

$$\boldsymbol{P} = (\boldsymbol{G} \bmod \boldsymbol{W}) \circ \boldsymbol{C} \tag{2.1}$$

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;
$\circ$ is element-wise multiplication;
mod is element wise modulo operation.

Deactivated chunks, i.e chunks that were moved out of the world texture to make room for others, are temporarily stored in RAM to allow quick reactivation when needed. All chunks are eventually saved to disk so that changes to the world are persistent.

2.2 Bringing variance to an endless world

It is a challenge to create enough variance in an endless world. 9 biomes were developed to increase diversity of the horizon. Each biome is characterized by steepness and roughness of the horizon and by types of tiles that it is composed of. Biomes are laid over the horizon by modulating temperature and humidity so unnatural transitions, such as from hot desert to cold tundra, are avoided. An example of the transition can be seen in Figure 2.4.

There are two types of caves below the horizon biomes. There are also magma pools very deep underground.

Generation of a chunk is a three-step procedure which combines several techniques - noises based on Simplex noise [9], cellular automata, hashing, and dithering. All steps of the generation procedure are implemented in compute shaders that manipulate auxiliary textures. Figure 2.5 shows how output of each step looks. At the end of the procedure, the new chunk is copied from the output auxiliary texture into the world texture (at position calculated by Equation 2.1).

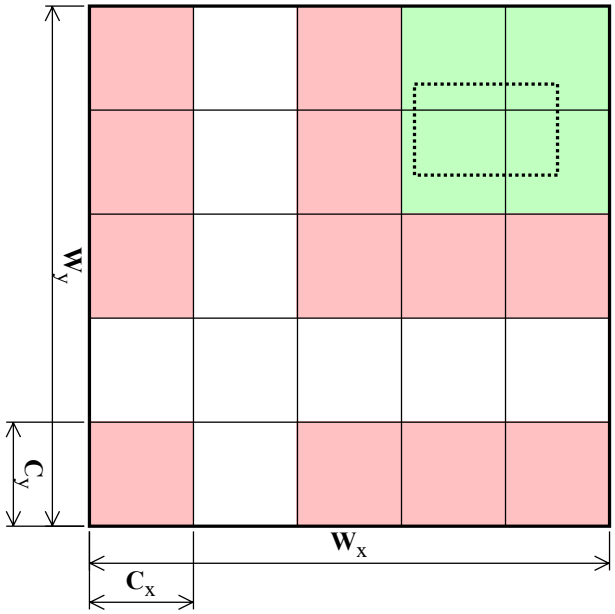


Figure 2.3: Placement of chunks inside the world texture is governed by Equation 2.1. The black lines show how the whole world texture is divided into chunks. The dotted rectangle represents a view into the world. The chunks that are overlapped by the view rectangle (green) are the chunks that have to be active. The neighboring chunks (red) could be activated in subsequent simulation steps, depending on movement of the view. Figure is adapted from bachelor's thesis [3].

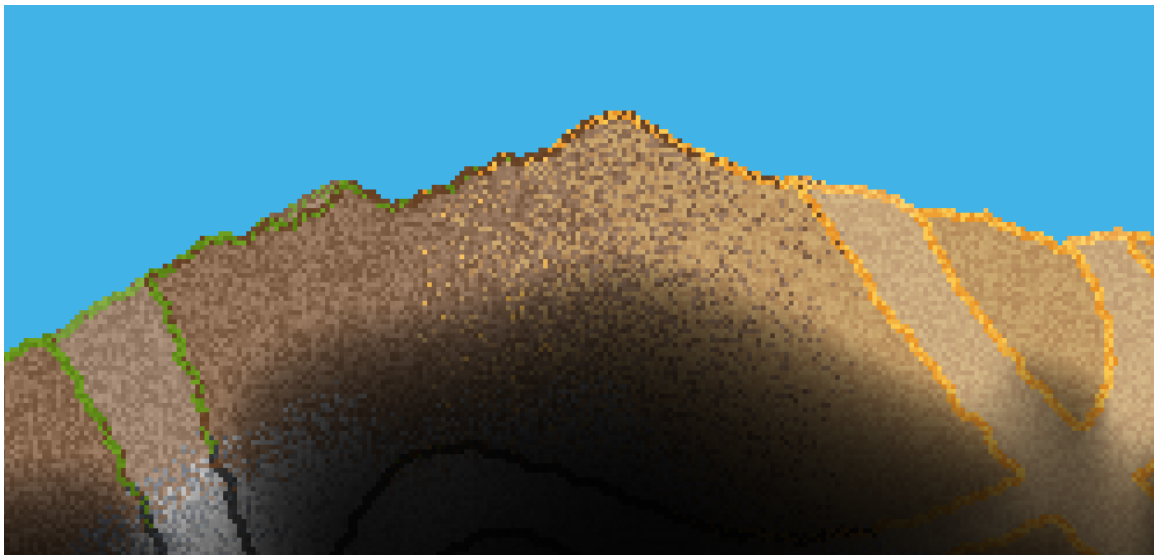
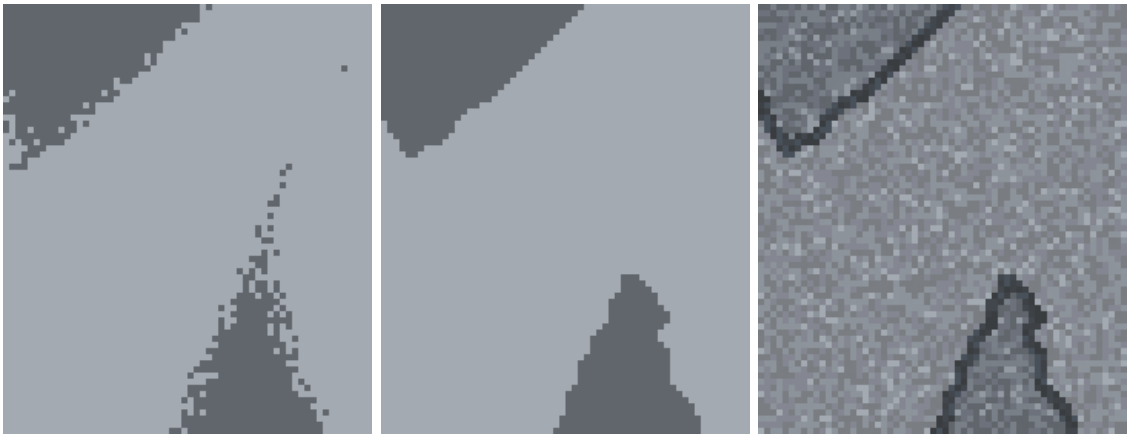


Figure 2.4: A thin dirt biome is inserted in between a grassland and a desert biome to avoid unnatural transition from sand directly to grass. Also, the transition between dirt and sand is not sharp but dithered. The same applies for transition from surface layer to cave layer below. Figure is taken from bachelor's thesis [3].



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

Figure 2.5: There are three main steps in generation of a new chunk. First, a fundamental structure is generated by combining several procedural noises. Then, the intentionally dithered edges are consolidated with a cellular automaton to create bumpy surfaces of caves. The last step selects variants of the tiles by hashing their positions and based on whether the tiles are on an edge or not. Figure is taken from bachelor's thesis [3].

2.3 Fluids and other simulation systems of the world

There are three systems that modify the world texture:

- The player can place new blocks/walls and remove existing ones. This also includes fluids as those are a special type of blocks.
- Slow transformation system manipulates as big portion of the world texture as possible to create an impression that the whole world is evolving. This system simulates slow processes such as short grass growth.
- Fluid dynamics system simulates liquids such as water or lava, gases such as steam or smoke and also fire. This system, however, only works in chunks that are close to the player.

2.3.1 Slow transformation system

The transformation system could be formally described as a stochastic cellular automaton. Apart from grass, it also simulates regrowth of 'rust' on edges of solid blocks when player places or removes some tiles so that the edges remain highlighted the same way as when the chunk was generated. The system was also used to simulate experimental mold that spreads in all directions.

All processes simulated by this system are slow. To simulate a slow process in a world of discrete tiles with discrete states, there is a small probability that the transformation (such as from dirt to grass) will happen. Nothing happens if this probability test fails.

The system is implemented in a compute shader that manipulates the world texture. To implement it efficiently, the probability test is avoided by inverting the problem. Instead of testing all applicable tiles each step, only small number of threads are launched. The

number of threads represents the portion of threads that would pass the test. Each thread then manipulates a random tile of the applicable ones without testing the probability.

Not all tiles within the world texture can be transformed. The way chunks are placed into the world texture inevitably creates discontinuities, i.e. seams where neighboring tiles are not neighbors in the whole world. Performing transformations on these tiles would produce artifacts, such as edges appearing inside solid areas.

In order to avoid these artifacts, a system which detects these discontinuities (which may only appear on borders between chunks) was employed. This system produces a list of chunks which can safely be transformed. This list is recalculated every time a chunk is activated or deactivated. Figure 2.6 shows an example world texture with some activated chunks and the discontinuities that they produce.

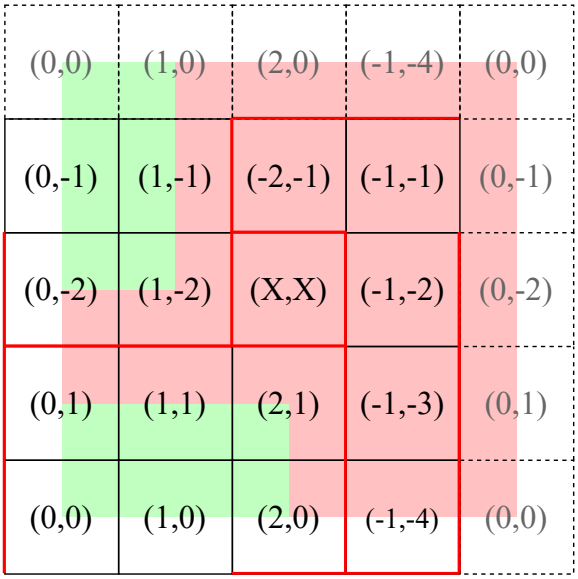


Figure 2.6: Detection of discontinuities in a world texture. Each square represents a chunk with its global position written in the middle. (X, X) marks that no chunk has been activated there yet. The dashed squares are mirrored chunks from the other side. Red lines highlight the discontinuities among the chunks. Areas tinted green can be selected for transformations, areas tinted red cannot. Figure is adapted from bachelor's thesis [3].

2.3.2 Fluid dynamics system

All fluids simulated by fluid dynamics system are a type of falling-sand simulation that is also implemented in a compute shader. Simulation of fluids based on cellular automata is not accurate and does not model pressure for example. It is, however, realistic enough for the project and it is much faster than proper fluid simulation [2]. Discontinuities do not have to be considered in this system because it only works in chunks that are so nearby to the player that it is guaranteed that they will be continuous.

Fluids are a special type of blocks - they move in the block layer but do not block the player completely, player can swim in them. Each type of fluid is defined by its vertical direction (gasses move up; liquids down) and a probability to move in this direction (generally lower for gasses to make them ascend slowly; high for dense liquids such as lava). The horizontal direction of a fluid tile changes dynamically and is defined by a direction bit

stored in variant attribute of the block. The horizontal direction inverts every time the fluid hits an obstacle. The type of the fluid also defines the probability to move in horizontal direction (e.g. lava spills horizontally slower than water).

There is also an experimental type of fluid, acid, which uses all bits of the variant attribute to store its velocity. Horizontal and vertical velocity is stored in 4 bits each.

Fluid dynamics system also simulates interactions between the fluids. Figure 2.7 shows an interaction between water and lava as an example, another example would be extinguishing of fire by water.

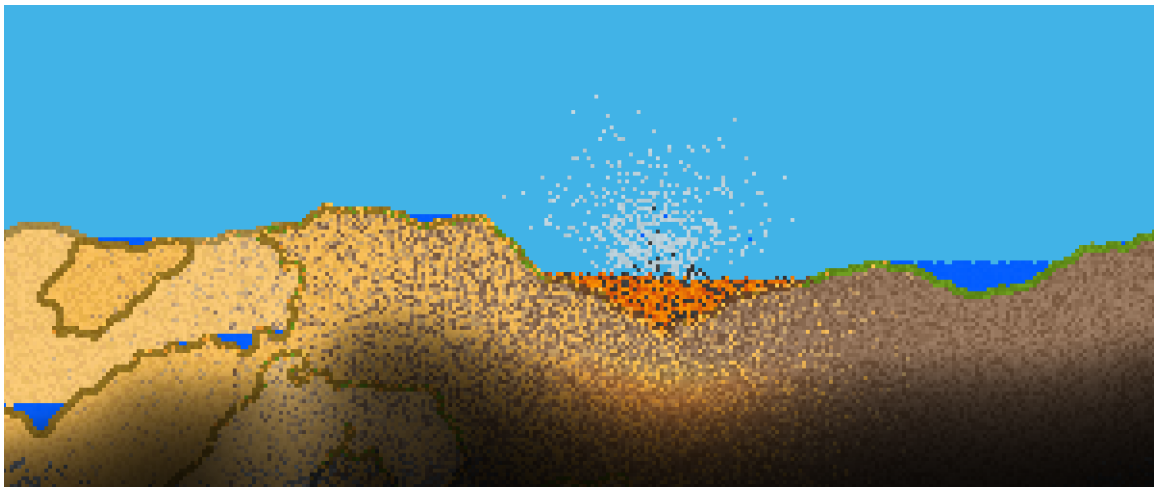


Figure 2.7: Water dropped into lava evaporates (ascending steam is visible right above the lava). Eventually, puddles are formed around by precipitation of the steam. Figure is taken from bachelor's thesis [3].

2.4 Software technology used to implement the project

The project is implemented in C++20. Care has been taken to support both Windows and Linux. Thus, CMake[1] is used as meta build system of the project and it compiles on MSVC and GCC compilers.

SDL 2[2] library is utilized to handle window creation and mouse/keyboard input. Thanks to this library, APIs for this purpose, specific for the above mentioned operating systems, are avoided.

OpenGL 4.6, accessed through GLEW[3] library, is used to accelerate rendering and parallel computations on GPU. The project makes plentiful use of compute shaders as these are used to implement generation and simulation of the world. GLSL 4.6 shader language is used to implement all shaders of the project.

Other supporting libraries used in the project include: GLM[4] (a math library imitating GLSL; used to ease vector calculations in C++) and LodePNG[5] (PNG coder and decoder; used to load textures of the project as well as save and load chunks of the world).

[1] Available at: https://cmake.org/
[2] Available at: https://www.libsdl.org/
[3] Available at: https://glew.sourceforge.net/
[4] Available at: https://github.com/g-truc/glm
[5] Available at: https://lodev.org/lodepng/

The codebase of the project is divided into two parts - a more general library, named RealEngine[1], which deals with window creation, input handling, low-level graphics etc. and an executable, named RealWorld[2], which implements the concrete simulation and rendering procedures of this project.

2.5 Limitations of the project prior to this thesis

The original thesis aimed to generate and simulate a 2D gaming world. Arguably, the biggest limitation in the fidelity of the world is that it is barren. There are multiple biomes spread across the horizon but they only differ in the type of surface tile and characteristics of the surface layer.

It is hard to create an impression of a natural biome without any vegetation. Thus, the main goal of this thesis is to add vegetation into the project.

The vegetation should keep the spirit of the tiled world. This means that it should appear to be made of tiles but, at the same time, it should be lively. It should sway in the wind, interact with the already present cellular simulation (e.g. to burn in fire) and the player should be able to interact with it too.

The visualization of the project includes dynamic shadows. The shadows improve appearance of the world immensely but the underlying algorithm is inefficient and it limits the maximum distance that light can travel. As a result, it is not possible to visualize a bright point light. The description of the original algorithm is given together with its improvement in Section 3.2 so that it is easier to understand the difference between the two algorithms.

[1] Available at: https://github.com/ZADNE/RealEngine
[2] Available at: https://github.com/ZADNE/RealWorld

Chapter 3

Performance improvements

This chapter describes performance improvements that are not directly visible from the user's perspective. The biggest such change, described in Section 3.1, is transition of the application from OpenGL graphics API to Vulkan API. The next Section 3.2 describes how dynamic light propagation and shadow casting was improved to be more efficient. The last Section 3.3 lists minor improvements added throughout the game.

This chapter is placed before Chapter 4, which concerns addition of vegetation to the project, because this was the chronological order and the transition to Vulkan API influenced implementation of the vegetation too.

3.1 Transition from OpenGL to Vulkan API

This section explains the differences between OpenGL and Vulkan APIs, it describes supporting libraries used to ease working with Vulkan. Finally, it describes the consequences of the rewrite for the code base of the project.

3.1.1 Differences between OpenGL and Vulkan APIs

OpenGL is an old graphics API. The first version was released in 1992, the latest version 4.6 was released in 2017. Even though it has evolved immensely over the years and deprecated many obsolete features, it limits parallelism and its performance is sometimes unpredictable [7].

Vulkan API is the successor API of OpenGL. The main goal of Vulkan API is to reduce driver overhead as much as possible. Driver overhead in OpenGL API comes from several sources:

- In OpenGL API, synchronization between commands is almost always implicit and thus sometimes unnecessary synchronization may happen. Vulkan provides only very few implicit synchronization guarantees [5, sec. Implicit Synchronization Guarantees] which means that the accelerator may execute most commands in parallel. If any synchronization is needed, it may be precisely expressed by one of synchronization primitives. There is a range of primitives which differ in the scope and guarantees provided by the synchronization.
- OpenGL API allows mutation of state that cannot be easily mutated. An example of this may be resizing of textures. This means that commands prior the resize in fact

refer to different image (i.e. different part of memory) than those after the resize. This also requires internal reference counting so that the old image is released when all work on it has finished. Vulkan does not allow this which forces the programmer to think more about his design.

- OpenGL API manages memory implicitly. When creating a buffer or texture, the user only provides vague hints what it will be used for. But there may be several types of hardware memory modules in CPU and GPU differing in visibility, access latency and throughput and coherence. OpenGL driver has to guess based on the hints which memory to put the object in and migrate it when the guess turns out to be wrong. In Vulkan, the user selects the memory themselves based on usage of the object but they also must adhere to limitations of the memory type, e.g. a buffer stored in GPU memory may not be mapped to CPU address space.

- OpenGL API does not support recording of commands for GPU from multiple CPU threads. Even though implementations of the API use worker threads in the background, recording of commands cannot be parallelized from the application's perspective because all calls to OpenGL are tied to a single global rendering state. Vulkan allows parallel command recording by giving each thread its own rendering state (stored in an object called command buffer).

A result of the above mentioned is that Vulkan is much lower-level and much more explicit API than OpenGL. When used well, it can be faster than OpenGL. If it is not used properly, it may perform even worse and it is much slower to develop with. Vulkan also has much brighter prospect of future support.

Another considerable difference is that Vulkan API uses SPIR-V, an intermediate representation bytecode, to define programmable shaders [5, sec. Shader Modules]. SPIR-V bytecode is typically compiled from the OpenGL shading language, GLSL, or High-Level Shader Language, HLSL[1].

Despite the old model and thanks to well optimized drivers, OpenGL still performed very well for the purposes of this project. Thus, the main reason for transition to Vulkan was personal interest in the new API rather than necessity to improve performance through this transition.

3.1.2 Supporting libraries used in the transition

The following assets from the official Vulkan SDK[2] were utilized in development:

- **Vulkan-Hpp**[3] is a header-only library which provides C++ bindings for the original C99 Vulkan API. It provides both RAII[4] and non-RAII wrappers of Vulkan objects.

- **Vulkan memory Allocator**[5] simplifies selection of the right memory heap and automatic grouping of objects into large allocations so that heap fragmentation is avoided.

[1] An overview of SPIR-V and related technology: https://www.khronos.org/spir/
[2] Whole SDK available at: https://vulkan.lunarg.com
[3] In SDK, also available at: https://github.com/KhronosGroup/Vulkan-Hpp
[4] Resource acquisition is initialization; a C++ programming idiom
[5] In SDK, also available at: https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

- **glslc compiler**[1] is utilized to compile GLSL into SPIR-V.
- **SPIRV-Cross**[2] is used to reflect on SPIR-V to ease creation of pipeline layouts. Pipeline layout objects work as an interface between the code of a concrete shader pipelines and the resources that they accesses.

3.1.3 Impact of the transition on the project

The codebase of the project is divided into two parts - a general library, which handles low level graphics, and an executable implementing the generation and simulation of the 2D world. This section describes what was done in both parts to transition to Vulkan API.

Replacement of graphics object wrappers in the general library

The RAII wrappers provided by Vulkan-Hpp were not used because they are unnecessarily generic for this project. Instead, the non-RAII wrappers were used to create custom RAII wrappers. The custom wrappers are simpler to construct and use because they use defaults appropriate for this project. For example, there is no depth output for graphics pipelines by default since the project is 2D, or the wrapper for buffer object allows easy access the CPU-mapped region (if it is mapped).

Another difference from the official RAII wrappers is that the custom ones do not delete the underlying Vulkan object right in their destructor. Instead, they enqueue the object to a global deletion queue which deletes the object at the end of the next frame. This makes the wrapper simple to use and makes the underlying Vulkan object live long enough until all commands have finished. Destroying a Vulkan object while there are commands that operate on it enqueued is undefined behavior [5, sec. Valid Usage].

Incorporation of shaders into the project

Since Vulkan does not accept GLSL directly, a number of CMake scripts were created so that shaders can be easily incorporated into the project. The goal was compile shaders into SPIR-V bytecode when building the project so that the resulting executable is bundled only with the bytecode, not with the shader sources (the OpenGL version of the executable included the shader sources in plain text).

The CMake scripts generate simple C++ wrapper code for each shader so that it can be easily accessed from C++ code. They also ensure that shaders are recompiled (by glslc compiler) whenever edited. Figure 3.1 explains the dependencies among user C++ code, user shader code and the generated files.

Impact on the executable

After having adapted the general library and shader integration, the main executable was rewritten. That required refactoring in many places. For example, a tile rendering class was coupled to a world-texture-owning class only by a global compile-time constant index of the image unit that the world texture has to be bound to. In Vulkan, this global state does not exist and the concrete image handle created in world-texture-owning class must be passed to the class that manages rendering of the world texture. In general, the necessary refactoring resulted in more tightly coupled classes.

[1] In SDK, also available at: https://github.com/google/shaderc
[2] In SDK, also available at: https://github.com/KhronosGroup/SPIRV-Cross

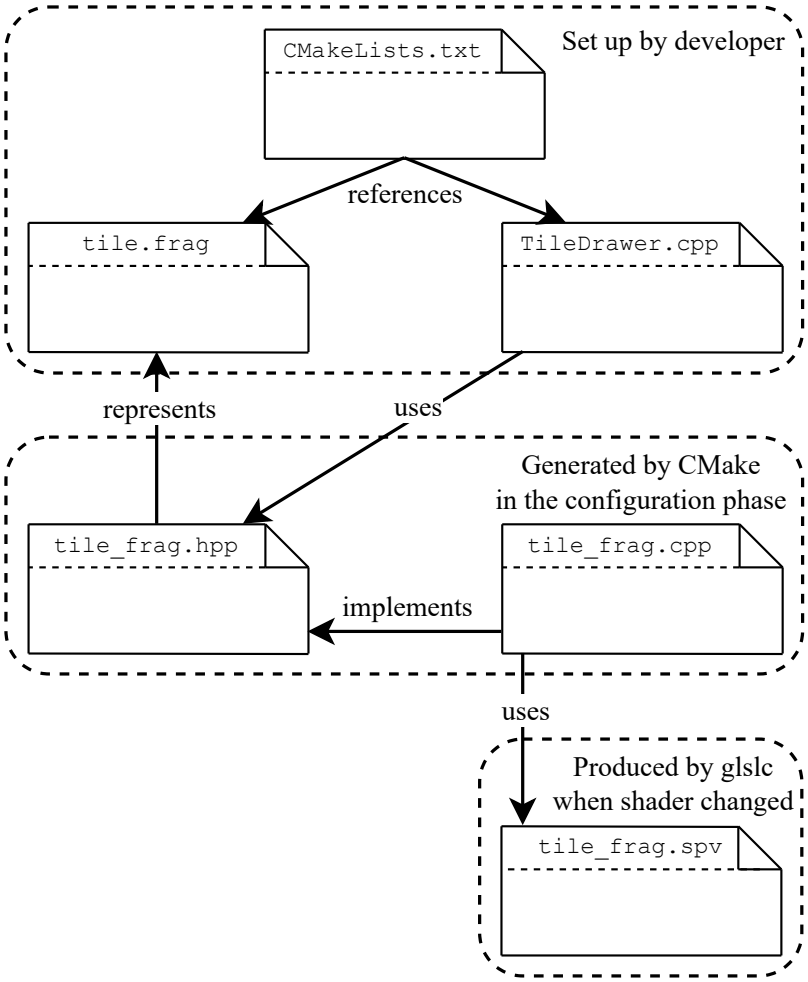


Figure 3.1: To add a shader into a RealEngine project, its GLSL source file must be specified in a CMakeLists.txt, similarly to how C++ source files are specified. CMake then generates a wrapper C++ header and source for the shader. The header declares a global constant variable representing the SPIR-V of the shader (which can be used to create a pipeline). The constant is defined in the generated source so that a change of the shader does not cause recompilation of other C++ sources that use the shader. The GLSL itself is compiled to SPIR-V, when changed, in a pre-build step.

It was also experimented to support both OpenGL and Vulkan so that the renderer could be switched after restarting the program. There were, however, many problems, e.g., an abstraction layer above the APIs with very different philosophies was needed, there are incompatibilities in GLSL dialects for each API, and there were bugs in acceptance of SPIR-V in OpenGL. Solutions to all these problems would bring a lot of complexity and little benefit so the idea was abandoned and all remnants of OpenGL implementation were in the end removed.

3.2 Improved dynamic light propagation

Dynamic light propagation and shadow casting is essential for player's immersion in this type of game. The propagation is expected to be unrealistic. Light does not reflect or refract on boundaries between air and solid tiles. It enters solid blocks and gradually attenuates inside the blocks instead. Figure 3.2 demonstrates the light propagation.

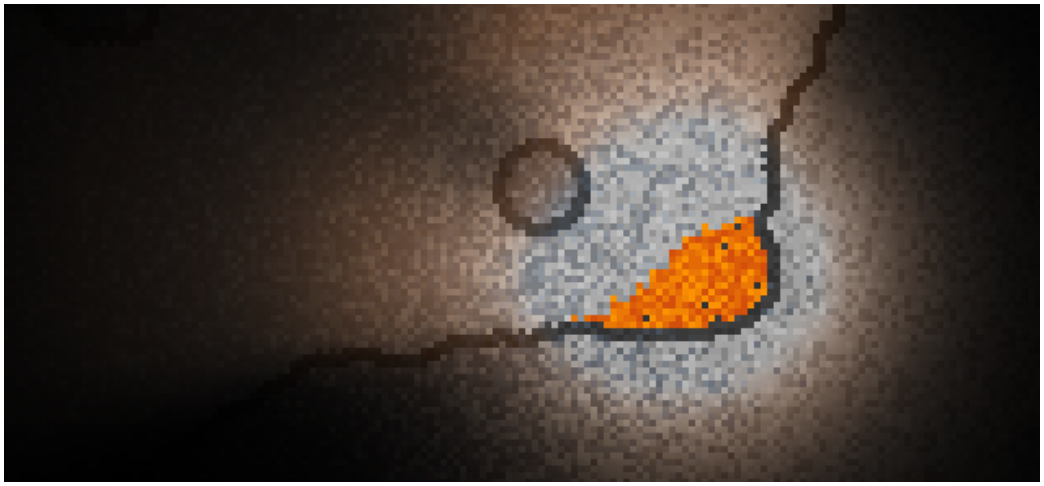


Figure 3.2: Lava spilled in a cave demonstrates how light propagates in the project. Light always travels in straight lines and it is gradually attenuated. Attenuation is slightly higher when traveling through solid tiles. This creates soft shadows and allows the player to see into the rock around them while underground.

A light propagation algorithm, which utilizes raster-based representation of the world, was developed in the original version of the project. The algorithm is, however, very texture-read demanding. This section explains the old algorithm as well as an improved version of the algorithm which reduces number of texture accesses without introducing noticeable artifacts.

3.2.1 The original light propagation algorithm

The original algorithm works in three steps. Intermediate steps store its outputs in auxiliary textures. Figure 3.3 shows an overview data flow in the algorithm. A more detailed description of the steps follows:

1. First, an analysis of tiles that are visible inside the view is performed. Outputs of the analysis are color and intensity of light emitted and a translucency factor for each 4×4 tiles unit. Tiles such as fire or lava emit red light, while most other tiles do not emit light so they are light sources with zero intensity. Translucency of a tile depends on whether it has a block or not. Solid blocks attenuate light more than air blocks. The unit represents an arithmetic mean of the emitted light and translucency of the tiles it covers. The outputs are stored in auxiliary textures (each texel of the texture represents one unit).

2. The idea of the next step is to calculate a unit's illumination by 'sweeping' light from surrounding units. More precisely, the total illumination of a unit is calculated as a sum of contributions from a fixed number of rays that travel from all directions

towards the center unit. These fixed-length rays start at a maximum distance that light can travel. As the ray travels towards the unit that is being calculated, it accumulates emitted light from each unit and attenuates the accumulated light with translucency factor of the unit. When the ray reaches the center unit, its contribution is added to the total illumination. The total illumination of the center unit is then converted to a transparent shadow color which is stored in another auxiliary texture.

3. The auxiliary shadow texture is rendered to main framebuffer on top of already rendered tiles. It is stretched with linear magnification to hide that the shadow is not calculated for each tile individually.

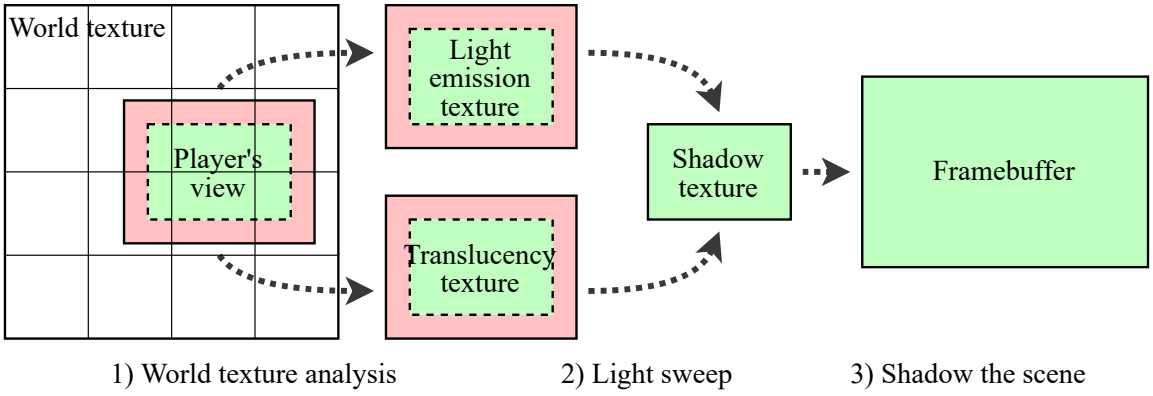


Figure 3.3: Dynamic lighting is calculated in 3 steps. First an area centered on player's view (green) is analyzed. The area is slightly bigger (red) than the size of player's view so that light that is just outside the view can still propagate inside the view. Output of the analysis are two textures representing light emission and translucency of tiles in the red rectangle. These are inputs to next stage which calculates illumination by sweeping rays. This produces a shadow texture that is rendered on top of already rendered scene in the last step. The auxiliary textures are typically much smaller compared to the world texture or the framebuffer than they are in this diagram.

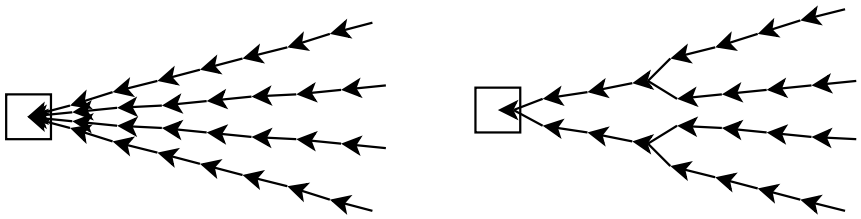
The second step is the most expensive step of the algorithm. The maximum light propagation distance is the decisive factor for performance of the algorithm. The bigger maximum distance, the larger disk to sweep, the longer rays are needed, the more rays are needed to sample the whole disk.

To give concrete numbers, to allow light to propagate 512 pixels on player's screen, each ray has to sweep 32 units (a unit is 4×4 tiles; a tile is 4×4 pixels) and about 201 rays would be required to fully cover even the farthest circle of the disk (circumference of the disk: $2\pi 32 \approx 201$). That is $32 \cdot 201 = 6432$ texture reads to sweep a single unit.

3.2.2 Improved light sweep pattern

The number of texture reads could be reduced by increasing size of units but that would have direct impact on details of the shadows - they would appear too blurry. A better way to reduce the number of reads is to exploit the fact that the closer the rays are to the center, the closer units they sweep.

So the improved version does not sweep with straight rays but a tree-like structure of rays. The farthest layer starts with same number of rays but two neighboring rays are joined and replaced by one once they get close enough. Figure 3.4 shows comparison of the original and improved pattern. Exact layers, where rays are joined, must be fine tuned to find a balance between performance and artifacts created by premature joining of rays.



a) Original straight sweep pattern b) Improved tree-like sweep pattern

Figure 3.4: Each arrow tip represents a light-propagation sampling point. The original algorithm used straight rays for light sweeping. The improved version uses a tree-like structure of short rays. Both patterns extend to all directions. The tree structure of rays reduces the number of texture accesses near the center where all rays fetch almost the same values. The simulation sweeps much larger area so the tree has more layers and the reduction is even more significant than in the diagram.

Sweeping naturally starts from the outer ends of outer rays and proceeds towards the center unit, joining the rays along the way. Joining two rays means that the swept light of the two is simply added together and the joined ray proceeds at an angle that is average of the joined rays.

Given an inner ray that represents N joined rays, light emission of the units it sweeps must be multiplied by N before adding it to the so-far accumulated light. This is to compensate for the fact that straight rays would sweep it N times too. The multiplicative translucency factor does not need any special treatment.

The improved sweeping pattern reduces texture reads by roughly 30 % - 4528 reads compared to 6432 with the old pattern on 32 units distance. But the performance gain was used to extend the maximum distance that light can travel (in other words - the maximum shine of a point light) while keeping the cost of the lighting calculation roughly the same.

3.3 Other minor improvements

A few other minor improvements throughout the project have also been done:

Firstly, size of the world texture used to be a compile-time constant. It is now possible for the user to select the size of the world texture in game settings (it always has to be a power of 2). Size of the world texture affects the amount of VRAM that the game requires and also indirectly affects simulation speed. That is because the bigger the world texture is, the more chunks will be selected for slow transformation and also more vegetation will be simulated.

Secondly, saving many chunks to disk is slow because each chunk has to be encoded to PNG format. This used to cause a short freeze when closing the game. This process was sped up by using multiple parallel CPU threads where each thread encodes a chunk independently of the other threads.

Chapter 4

Vector-based vegetation in raster-based world

The main goal of this thesis is to add vegetation to the 2D world. This chapter describes theory, design and implementation of this feature. Figure 4.1 shows example images of the vegetation added to the project. More images can be seen in Appendix A.

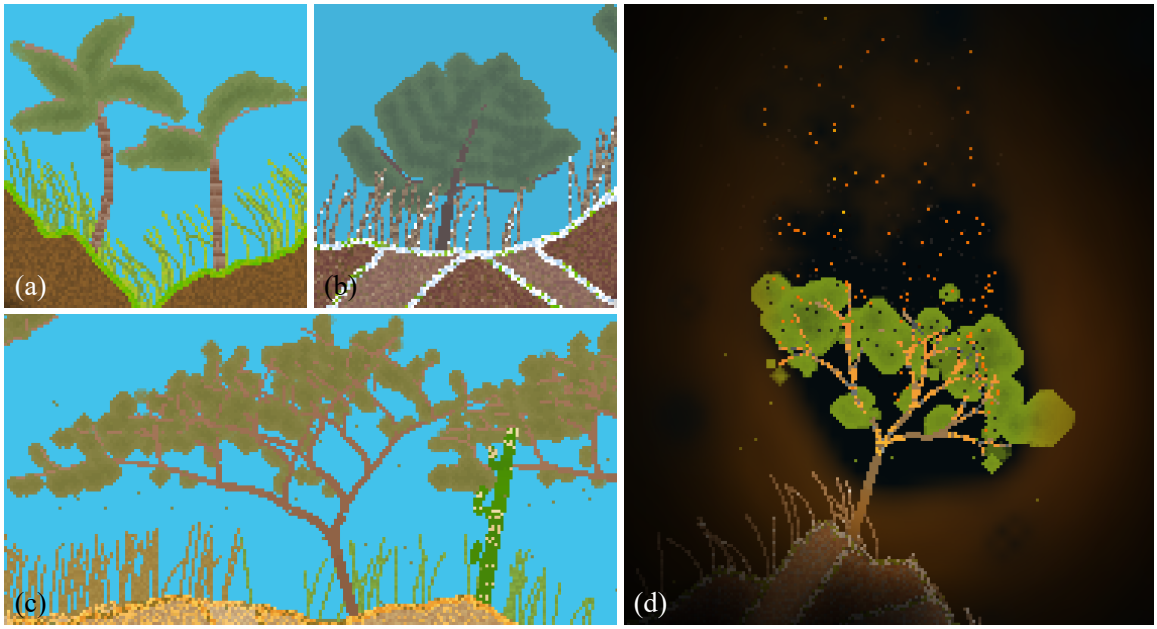


Figure 4.1: (a) Palm trees in jungle. (b) A spruce in taiga. (c) Acacias and a cactus in savanna. (d) A burning oak at night.

Since generation and simulation of vegetation differs from generation and simulation of raster world, Section 4.1 introduces existing approaches used to generate and simulate vegetation.

Generation and simulation approaches explained in the first section are vector based, i.e. vegetation is represented by a set of interconnected lines/rods/segments which form branches. But this is vastly different from the rest of the world which is raster based, i.e. it is composed of a regular grid of square tiles. Section 4.2 explains the design which ties these worlds together.

Nearly everything designed in Section 4.2 is implemented using shaders in Vulkan. The last Section 4.3 describes some interesting details of the implementation.

4.1 Theory: Approaches to vegetation generation and simulation

Vegetation cannot be generated by the same means used for the rest of the world (noises, dithering, automata). Instead, L-systems, described in Subsection 4.1.1, are used to generate vegetation in this thesis.

Cellular automata, used to simulate fluids and other phenomena in the rest of the world, cannot be used to simulate sway of vegetation in the wind. Instead, the generated vegetation is simulated to sway according to proper laws of physics that govern swaying of branches when external forces are applied to them. The laws are laid out in Subsection 4.1.2.

4.1.1 L-system: a formalism for generation of plants

This subsection describes L-systems, a formalism used to generate structure of vegetation. The simplest deterministic L-systems are formally described first. Then, selected extensions of the formalism are outlined. The selected extensions enhance realism of the generated vegetation. Finally, it is discussed what phenomena control growth of real vegetation and how they can be modeled in L-systems.

Deterministic L-systems and their geometric interpretation

An L-system is a parallel rewriting system defined by a formal grammar. The simplest, deterministic and context-free, L-systems are defined by an ordered triplet $\langle V, \omega, P \rangle$ [10, sec. 1.2], where:

- V is the symbol alphabet of the system,
- $\omega \in V$ is a non-empty string called the axiom,
- $P \subset V \times V^*$ is a set of productions (rewriting rules) where each rule replaces a symbol (predecessor) with a string (successor).

The system is deterministic if, and only if, for each symbol there is at most one rule with the same predecessor. The system starts from the axiom string and evolves by repeatedly applying a rewrite rule on each symbol of the current string to produce a new one. For symbols that have no rule assigned to it, it is assumed there is an identity rewrite rule in P so the symbol is simply copied.

Each symbol of the alphabet also has to be given a geometric interpretation so that an output string of the L-system can be converted to a plant structure. In 2D, there are four fundamental interpretations that a symbol can have [10, sec. 1.3]:

- F, move forward by step of length d while drawing a line;
- f, move forward by step of length d without drawing a line;
- $+$, turn left by angle α;
- $-$, turn right by angle α.

After assigning d and α values, the interpretation is done by so-called turtle. The turtle starts from an initial position and an initial facing direction. Then it parses the output string symbol by symbol while performing commands assigned to the symbols, gradually forming the output image. An example L-system interpreted by a turtle can be seen in Figure 4.2.

If the model of the turtle is extended to hold not just a single state but a pushdown stack of states and two more symbol interpretations are introduced, branching structures can be created [10, sec. 1.6]:

- [, push the current state of the turtle to the stack;
-], pop a sate from the stack and make it current.

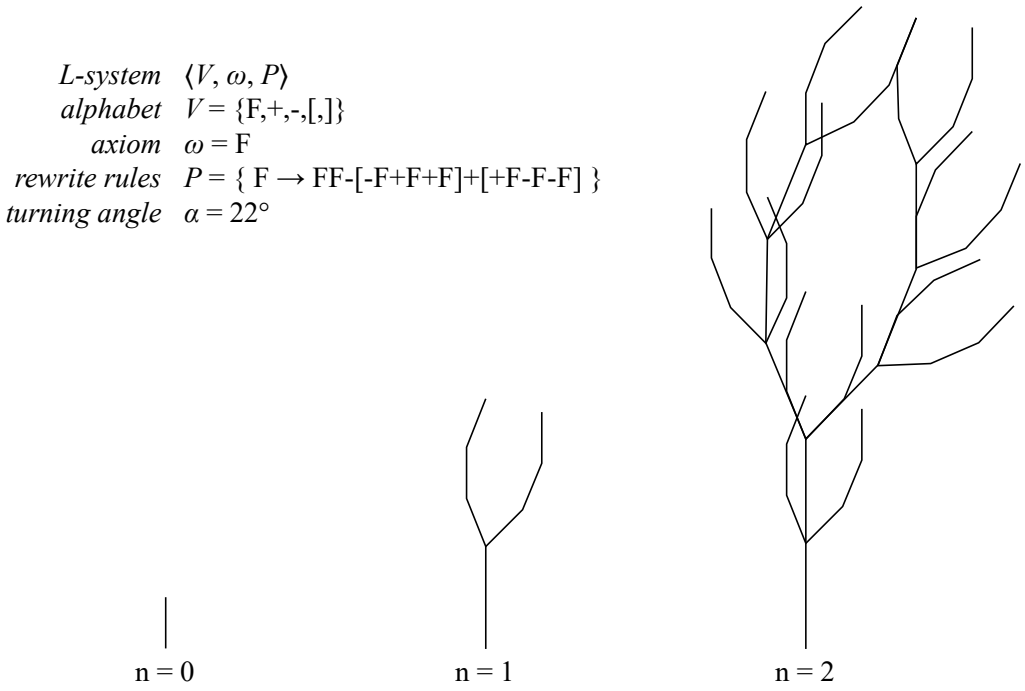


Figure 4.2: An example deterministic branching L-system with its first two iterations visualized by a turtle interpreter. It can be seen that the structure expands very quickly.

Extensions of L-systems to make the plants more realistic

While deterministic L-systems allow generation of many interesting geometric patterns, they are not very suitable for generation of realistic plants because those are never perfectly regular. Irregularities can be caused by factors such as a bird breaking a tip of a branch, or a storm causing a tree to grow crooked. These external factors have to be modeled somehow to produce realistic-looking plants.

The simplest way to suppress the regularity is **randomization of turtle interpretation** of an L-system. This means that random small offsets are added to lengths of branches and to turning angles. This improves fidelity, however, it is still noticeable that all instances of randomized interpretation share the same topology.

Stochastic grammars model random external factors by allowing multiple rules with the same predecessor. Probabilities are assigned to the rules to determine which one of the potential rules is used [10, sec. 1.10]. Rule 4.1 is an example of a stochastic rule which is interpreted that there is 60% probability that a segment only prolongs and 40% probability that the segment prolongs and branches at the end.

$$\begin{aligned} F &\xrightarrow{0.6} FF \\ F &\xrightarrow{0.4} FF[-F][+F] \end{aligned} \tag{4.1}$$

Stochastic rules make the L-system nondeterministic and similar to nature. As plants belonging to one species grow similar but not not exactly the same structure, plants generated from a single stochastic L-system also grow similar but not exactly the same structure.

Another extension of L-systems which makes them more realistic are **parametric grammars**. Parametric grammars use symbols which have real number parameters assigned to them [10, sec. 1.7]. The numeric parameters may be used to determine whether a rule is applicable or not. For example, given symbol F with parameter l, it is rewritten to the right hand side of Rule 4.2 if, and only if, parameter $l > 5$.

$$F(l)\colon l > 5 \rightarrow F(l \times 3)F(l - 1) \tag{4.2}$$

The numeric parameters are also used when interpreting the output string. Example interpretations of parametric symbols are: branch segment with length as parameter, turning symbol with the turning angle as parameter, etc. This avoids the underlying limitation of non-parametric L-systems in which lengths of all branches must be integer multiples of the base length (the same limitation applies to turning angles). Since there can be any mathematical expression involving the parameter on the right hand side, branch segments of parametric L-systems may have arbitrary lengths.

Stochastic and parametric rules complement each other well. Figure 4.3 shows example trees generated by stochastic parametric L-system.

Tropism of plants

When designing L-systems to generate realistic plants, it is important to look at principles that govern growth of real plants. The most important phenomenon governing plant growth is light [4]. Branches of most plants grow towards light, i.e. they exhibit positive phototropism, while trunk of trees grows strictly upwards, i.e. negative geotropism.

Tropism of plants is modelled in L-systems in interpretation phase by turning the turtle towards tropism vector [10, ch. 2]. Given tropism vector $\boldsymbol{T}$, current orientation of turtle interpret $\boldsymbol{H}$, angular difference θ between them, and coefficient e representing the segment's sensitiveness to tropism, the turtle is turned towards tropism vector by the angle α according to Equation 4.3. Figure 4.4 visualizes the tropism adjustment.

$$\alpha = e \sin \theta \tag{4.3}$$

Parallelization of L-systems

There might be need to speed up generation by L-systems. One of the viable approaches is to parallelize them. This can be done on three levels [6]:

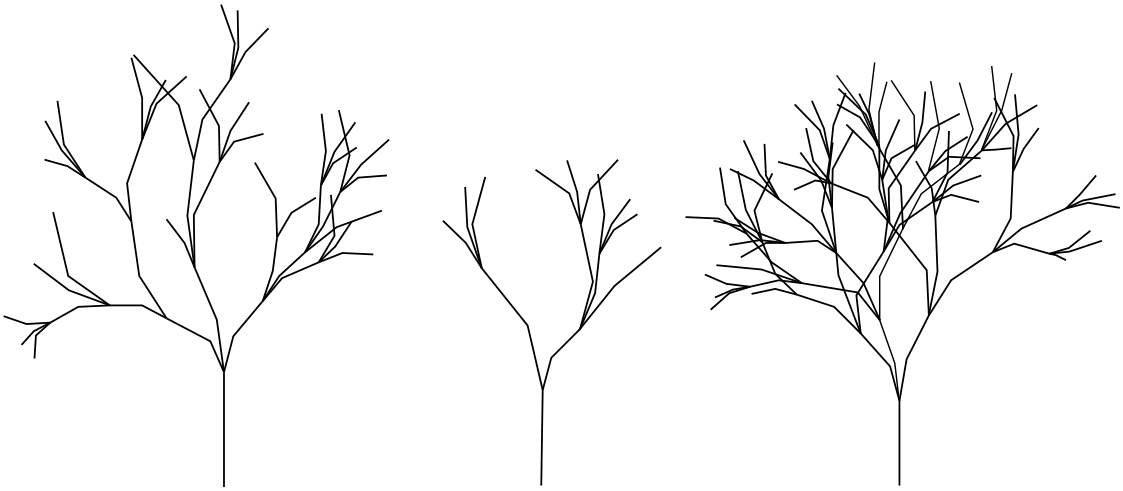


Figure 4.3: Sample trees generated by the same stochastic parametric L-system of six rules and in the same number of iterations. Botanic interpretation is that these are specimen of the same species and of the same age. The only aspect which causes the trees to differ is the initial seed of randomness.

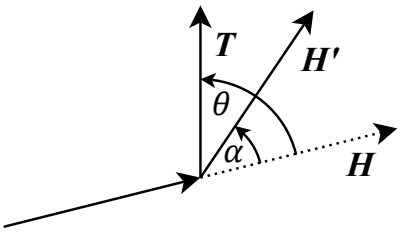


Figure 4.4: When interpreting output string of an L-system, tropism causes the next segment to rotate slightly in the direction of tropism vector $\boldsymbol{T}$, i.e. it rotates from orientation $\boldsymbol{H}$ to $\boldsymbol{H'}$. Equation 4.3 explains how the adjustment is calculated. Tropism may be used to, for example, make the plant lean towards light. Figure adapted from book The Algorithmic Beauty of Plants [10, fig. 2.9].

- Derivation of L-systems is parallel by definition - all applicable rules are applied simultaneously in an iteration. This is trivially parallelizable, except for the fact that the parallel threads need to concatenate the output strings. The concatenation requires coordination by a prefix scan to calculate offsets where to put the derived symbols.
- Interpretation of L-systems is serial. The turtle interpret processes each symbol one by one from the beginning. However, parallelism can be found in branching L-systems. Each opening bracket creates two independent work units - interpretation of symbols within the brackets and interpretation of symbols after the closing bracket.
- Multiple L-systems can be independently derived and interpreted in parallel. This corresponds to generating multiple trees in a forest in parallel.

4.1.2 Mechanics of swaying branches

Swaying of the generated vegetation can be simulated by a periodical leaning to a side with the offset relative to the distance to the base of the tree [13]. This simulation is computationally cheap but it does not model varying stiffness of branches properly.

So, instead, the following branch mechanics model, greatly inspired by paper Real-Time Procedural Animation of Trees [1], is used because it simulates swaying more accurately.

For the purposes of physics simulation of swaying, branch segments, generated by L-systems, are interpreted as cylindrical rods of a fixed length. A branch segment is always attached to the end of a parent segment. A segment has exactly one parent but a segment may have zero or more children segments attached to it. The root of a tree is handled by a degenerate segment of zero length that is parent of itself. The actual trunk segment of a tree is a child of this.

A segment cannot detach from its parent in simulation but, when an external force is applied to it, it may rotate relatively to its parent. The initial angle between a segment and its parent (as it was generated by the L-system) is remembered throughout the simulation and segments should bend back to this rest angle when no other force is applied.

The goal of the simulation is to create a swaying animation of the branches by rotating the individual segments. Angular acceleration of each rod needs to be calculated each step of the simulation to do so. Equation 4.4 shows how angular acceleration of a rod α relates to its torque τ and moment of inertia I.

$$\boldsymbol{\alpha} = \frac{\tau}{I} \tag{4.4}$$

Equation 4.5 shows how to calculate the moment of inertia I of a thin rod of mass M and length L rotated about an end which is the case for branch segments.

$$I = \frac{1}{3}ML^2 \tag{4.5}$$

In 2D space, the torque τ can be calculated by Equation 4.6 where L is the length of the rod, F is the net force applied to the end of it and θ is the angle between the rod and the force F.

$$\tau = LF\sin\theta \tag{4.6}$$

There are three forces that contribute to the net force F:

- an external horizontal wind force F_{wind},
- a spring-like force F_{spring} which pulls it back to rest angle β_{rest} (Magnitude of this force is proportional to stiffness, an internal parameter of each segment),
- and a downward-facing gravitation force F_G.

A situation involving two segments with forces and angles among them is illustrated in Figure 4.5.

4.2 Design: Connecting vegetation with the rest of the world

The goal of this section is to explain design of the systems which generate and simulate the vegetation of this project. It should be first stated what vegetation this project aims at. The requirements for vegetation are:

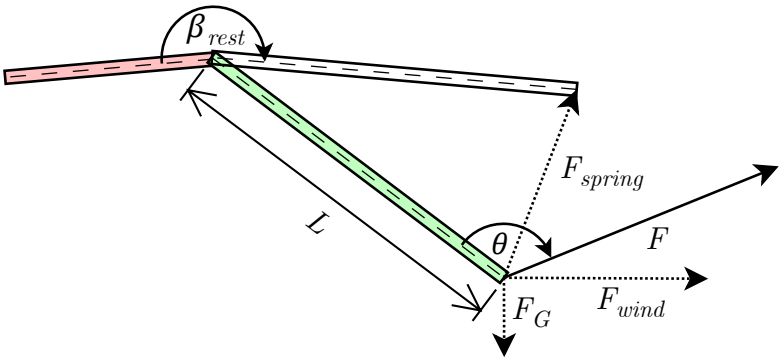


Figure 4.5: Given a parent branch segment (red) and its child segment (green), there are three forces applied to the child segment: downward facing gravitation force F_G, horizontally facing wind force F_{wind} and a spring like force F_{spring} which faces the rest orientation of the child segment. The resultant force F creates torque (see Equation 4.6) which in turn causes the segment to rotate (see Equation 4.4) around the pivot point where it is attached to its parent. Collisions between the segments are not considered.

Requirement 1: No matter the underlying model, the vegetation must be visualized as if it was composed of the same tiles as everything else in the world.

The intent is not to break the graphical style of the project.

Requirement 2: The vegetation must be generated on a per-chunk basis as the rest of the world does. It also must be able to be activated and deactivated on a per-chunk basis.

The intent is to maintain the illusion that the world is endless.

Requirement 3: The vegetation must interact with the already present simulation systems based on cellular automata. Player must be able to modify it, too.

The intent is to keep the world lively and interactive.

The main technical challenge of the project comes from the difference in representation of vegetation and the rest of the world. Branch segments are simulated as vector-based rods during the simulation. However, Requirement 1 says that it should seem that the vegetation is composed of tiles aligned to a grid, i.e. a raster.

The first Subsection 4.2.1 overviews the main game loop and shows how rasterization is used to connect vector and raster representations together. Subsection 4.2.2 describes how space for branch segments is allocated so that vegetation can be dynamically added and removed from the world.

Subsection 4.2.3 overviews the procedure of vegetation generation as more than expansion of L-systems is involved. Subsection 4.2.4 describes how simulation of swaying is parallelized so that it can be implemented efficiently in GPU.

Subsection 4.2.5 explains more details about rasterization of branches. It also describes how leaves are formed because leaves are not rasterized like branches.

4.2.1 Game loop overview

Rasterization is used to breach the fundamental difference between vegetation's vector representation and the surrounding world's raster representation. The goal is to meet Requirement 1 (i.e. vegetation should seem to be made of tiles) and Requirement 3 (i.e. vegetation should interact with the rest of the world).

Three approaches, how this could be approached from the perspective of what is the target texture of the rasterization, were considered:

1. Vegetation is rasterized to the main framebuffer

   Despite being the most straightforward option, it has only downsides. It struggles with meeting Requirement 1 because vegetation must be composed of 4×4 pixels large tiles but hardware does not do this. For example, if a branch segment not aligned to an axis was rasterized, it would rasterize it smoothly on pixel level but jagged 4×4 pixel edges are actually needed.

   It struggles in meeting Requirement 3 too because there is no connection between the vegetation and the raster world.

2. Vegetation is rasterized to a helper texture of the same resolution as the world texture

   The helper texture could be added as another input to the shaders which render world texture to the main framebuffer so vegetation would essentially become another layer, next to block and wall layer. Before each frame, the helper texture would be cleared similarly to how framebuffer is cleared before every frame.

   This option meets Requirement 1. However, Requirement 3 is troublesome as there is no feedback loop between vegetation's branch swaying simulation and cellular simulation. For example, if wood is rasterized next to fire tile and the cellular simulation turns it into a burning wood, the next step it is cleared and normal wood is rasterized again.

3. Vegetation is rasterized into the world texture and *unrasterized* back

   Since the vegetation is rasterized directly to the world texture, it cannot be just cleared. To remove the vegetation from the world texture before next step, it is rasterized again with a different shader which replaces all its wood tiles with air tiles.

   The second rasterization is essential for the procedure. It provides an opportunity to see how the wood tiles rasterized previously have changed by cellular simulation. So the second rasterization, called *unrasterization*, moves the changed tiles from the world texture to an auxiliary raster. The auxiliary raster is used in the next step to move the tiles of branches back to the world texture. This completes the needed feedback loop between the simulation systems.

   Despite being the most complex solution, it meets all requirements. Requirement 1 is met trivially, no modification to the world rendering shaders is needed. Requirement 3 for interactivity is met thanks to the unrasterization phase.

As the last option is the only option which meets all requirements, it will be the only one considered further. Figure 4.6 shows the three steps of vegetation simulation and their effect on the world texture.

Figure 4.7 puts vegetation rasterization process into context of the whole game loop of the project. The game loop consists of a simulation loop which runs at constant rate

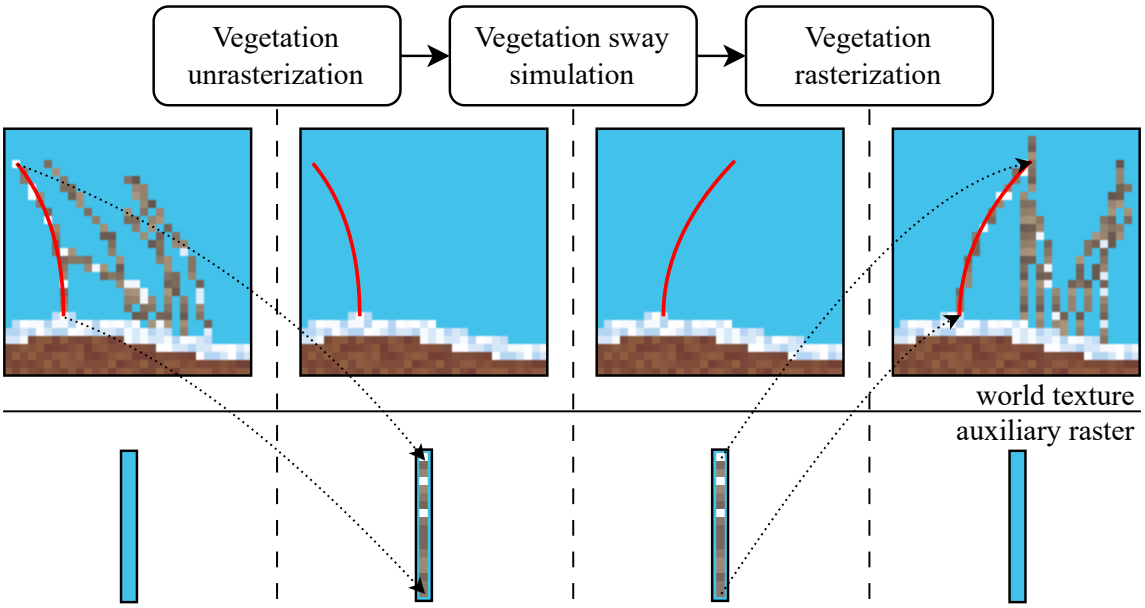


Figure 4.6: Simulation of vegetation consists of three main steps. First, unrasterization moves branches from the world texture to an auxiliary raster. Vector representation (red curve) of branches is then updated according to physical forces applied to them. Finally, branches are moved by rasterization from the auxiliary raster back into the world texture at their new position.

per second and a nested loop which renders the current state of the simulation to the main framebuffer of the application. Iterations of the rendering loop fill the time between simulation steps so its rate is variable based on speed of the machine running the application. A game loop with rendering rate not bound to simulation step rate is more adaptable to variously performant hardware than a game loop with rendering and simulation bound together [8].

4.2.2 Memory representation of vegetation

Previous sections made it clear that vector representation of branch segments is maintained throughout the simulation. But this only concerns one segment - this subsection explains how vegetation as a whole is represented in the world.

Two things are crucial for the chosen representation:

- The vegetation must be held in GPU memory so that shaders can effectively simulate its sway and rasterize it. This means that the branch segments should ideally be stored in a continuous range.
- The vegetation must be added and removed from the world on a per-chunk basis (see Requirement 2). The problem is that while every chunk has the same count of tiles, the count of branch segments per chunk varies.

This leads to general memory allocation problem - given a preallocated buffer of branch segments, subranges of different sizes have to allocated and freed while trying to avoid unnecessary fragmentation of the whole range.

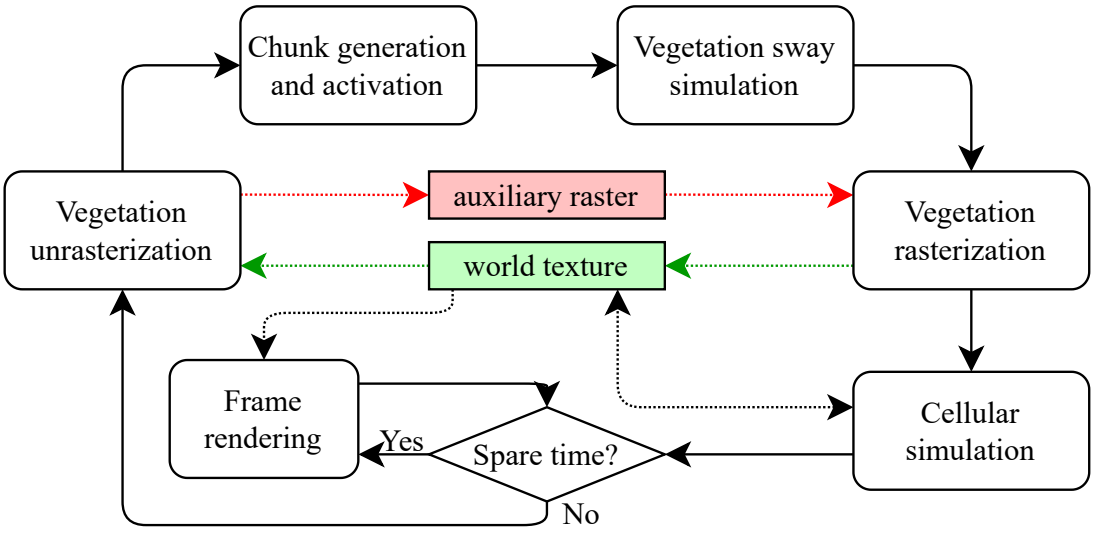


Figure 4.7: The game loop consists of six main blocks. First, vegetation from previous step is unrasterized, i.e. wood tiles are moved from the world texture to an auxiliary raster. This is followed by optional generation and activation of chunks, based on player's movement. Then new position of branch segments are calculated and they are rasterized in the new position, i.e. wood tiles read during unrasterization are moved back to the world texture. Finally, cellular simulation is performed, this may alter the wood tiles again, and zero or more frames are rendered, depending on time. The dotted lines represent important data transfers.

The designed data structure has three parts: the main array of branch segments $\boldsymbol{S}$, a small array of allocation ranges $\boldsymbol{R}$ and a small 2D array $\boldsymbol{I}$ working as an index to speed up chunk deallocation.

The gist of the data structure is that i-th allocation $\boldsymbol{R}_i$ in $\boldsymbol{R}$ owns a number of consecutive segments in $\boldsymbol{S}$ in between the range owned by $\boldsymbol{R}_{i-1}$ and $\boldsymbol{R}_{i+1}$. Allocation is done by searching for a big enough range in $\boldsymbol{R}$. Deallocation is done by merging subsequent ranges. The gist of the data structure can also be seen in Figure 4.8.

The number of segments owned by a range is called *capacity*. An allocation range also has *size*. Size denotes how many segments, out of capacity segments, from the beginning of the range contain valid data, and thus may be simulated and rasterized. Table 4.1 shows all valid combinations of capacity and size values of a range and their respective interpretations.

The data structure is initialized as follows: all indices in $\boldsymbol{I}$ are invalid, allocation range $\boldsymbol{R}_0$ is available with capacity equal to N, all other ranges in $\boldsymbol{R}$ are shrunken, branch segments in S may be left uninitialized.

Two operations on the above data structure are needed: allocation of given number of branches for a given chunk that does not have any branches allocated yet, and deallocation of all branches of a given chunk.

To allocate, array $\boldsymbol{R}$ is searched for the first available allocation $\boldsymbol{R}_i$ that is big enough and its leftover unused capacity is donated its right neighbor $\boldsymbol{R}_{i+1}$. The donation is a crucial part of the procedure because it moves the boundaries between the ranges and

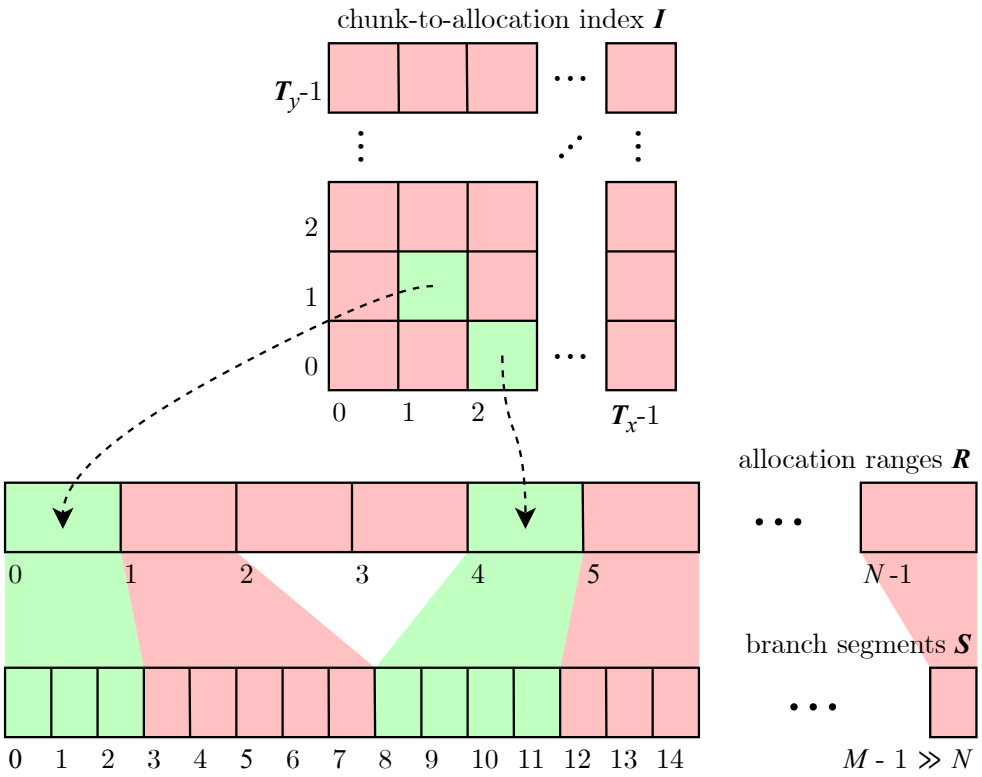


Figure 4.8: Dynamic allocation of branch segments from array $\boldsymbol{S}$ is managed by the following data structure. Every branch segment in $\boldsymbol{S}$ is owned by some allocation range in array $\boldsymbol{R}$. The occupied ranges are green, available ranges that may be occupied are red. Some ranges ($\boldsymbol{R}_2$, $\boldsymbol{R}_3$) do not own any segments. Every chunk of the world texture that has at least one branch segment in it has all branch segments in one range which can be found in the index $\boldsymbol{I}$.

capacity	size	interpretation
$= 0$	$= 0$	shrunken range
$\neq 0$	$= 0$	available range
$\neq 0$	$< capacity$	partially occupied range
$\neq 0$	$= capacity$	fully occupied range

Table 4.1: There can be four interpretations of i-th range $\boldsymbol{R}_i$ in an array of ranges $\boldsymbol{R}$. Apart from fully occupied and available ranges, there can be shrunken and partially occupied range. Shrunken range is created when consecutive ranges are deallocated - the leftmost range in $\boldsymbol{R}$ gets capacity of the right ones which become shrunken. Partially occupied range is created when a range is occupied but its right neighbor $\boldsymbol{R}_{i+1}$ is already occupied too so $\boldsymbol{R}_i$ cannot donate its leftover capacity to any other range.

gradually subdivides the initial available range. Algorithm 4.1 describes this procedure more precisely.

The allocation algorithm is stripped off a check whether there has not been a range allocated for the chunk already. This is for clarity and also because the existing system

which manages activation of chunks already performs this bookkeeping so it never asks for two ranges on a chunk.

Deallocation frees all segments of a given chunk. It uses search index $\boldsymbol{I}$ to find previously allocated range of the chunk (a chunk may have at most one allocation). As allocation shifts boundaries between the ranges by donating free space, deallocation must do the opposite. That is, deallocation merges consecutive available range into the leftmost one and leaves other shrunken. Algorithm 4.2 describes the procedure precisely.

Algorithm 4.1: Allocation of branch segments of a chunk

Data : Array $\boldsymbol{R}$ of N allocation ranges, 2D array $\boldsymbol{I}$ of $\boldsymbol{T}_x \times \boldsymbol{T}_y$ indices
Inputs : Count of segments C_{req} of chunk at position $\boldsymbol{P}$ within the world texture
Output: Index of the first segment if space available

for $i \leftarrow 0$ **to** $N-1$ **do** // Search every range
  if $\texttt{size}(\boldsymbol{R}_i) = 0 \wedge \texttt{capacity}(\boldsymbol{R}_i) \geq C_{req}$ **then** // If range is suitable
    if $\texttt{capacity}(\boldsymbol{R}_i) > C_{req}$ **then** // If range is bigger than necessary
      if $i \neq N-1 \wedge \texttt{size}(\boldsymbol{R}_{i+1}) = 0$ **then** // If next range is available
        $\texttt{capacity}(\boldsymbol{R}_i) \leftarrow C_{req}$; // Donate unused space
        $\texttt{begin}(\boldsymbol{R}_{i+1}) \leftarrow \texttt{begin}(\boldsymbol{R}_i) + C_{req}$;
        $\texttt{capacity}(\boldsymbol{R}_{i+1}) \overset{+}{\leftarrow} \texttt{capacity}(\boldsymbol{R}_i) - C_{req}$;
    $\texttt{size}(\boldsymbol{R}_i) \leftarrow C_{req}$; // Occupy the range
    $\boldsymbol{I}_{\boldsymbol{P}} \leftarrow i$;
    return $\texttt{begin}(\boldsymbol{R}_i)$; // Allocation success
return *invalid index*; // Failure: no available range is big enough

Algorithm 4.2: Deallocation of branch segments of a chunk

Data : Array $\boldsymbol{R}$ of N allocation ranges, 2D array $\boldsymbol{I}$ of $\boldsymbol{T}_x \times \boldsymbol{T}_y$ indices
Inputs : Position $\boldsymbol{P}$ of a chunk within the world texture whose branches to deallocate

$i \leftarrow \boldsymbol{I}_{\boldsymbol{P}}$;
if $i =$ *invalid index* **then return**;
$\boldsymbol{I}_{\boldsymbol{P}} \leftarrow$ *invalid index*;
$R \leftarrow \boldsymbol{R}_i$;
$\texttt{size}(\boldsymbol{R}_i) \leftarrow 0$; $\texttt{capacity}(\boldsymbol{R}_i) \leftarrow 0$; // Shrink the original range
$\texttt{size}(R) \leftarrow 0$;
if $i \neq N-1 \wedge \texttt{size}(\boldsymbol{R}_{i+1}) = 0$ **then** // If right neighbor available
  $\texttt{capacity}(R) \overset{+}{\leftarrow} \texttt{capacity}(\boldsymbol{R}_{i+1})$; // Merge with it
  $\texttt{capacity}(\boldsymbol{R}_{i+1}) \leftarrow 0$;
for $i \leftarrow i-1$ **to** 0 **do** // Search neighbors to the left
  if $\texttt{size}(\boldsymbol{R}_i) = 0$ **then** // If neighbor available
    $\texttt{capacity}(R) \overset{+}{\leftarrow} \texttt{capacity}(\boldsymbol{R}_i)$; // Merge with it
    $\texttt{begin}(R) \overset{-}{\leftarrow} \texttt{capacity}(\boldsymbol{R}_i)$;
    $\texttt{capacity}(\boldsymbol{R}_i) \leftarrow 0$;
  else break; // Found an occupied neighbor to the left
$\boldsymbol{R}_{i+1} \leftarrow R$; // Store the merged range

4.2.3 Generation of vegetation for the endless world

Given the data structure for dynamic allocation of branch segments and given L-systems, formalism for generation of vegetation, the goal is to generate trees, grasses and bushes for chunks on the horizon of the 2D world.

Generation takes place between vegetation unrasterization and sway simulation when the player gets close to a chunk that they have never visited. Since vegetation is stored in GPU memory for simulation and since calculations of terrain horizon and biome distribution, which are inputs for vegetation generation, are expressed by compute shaders, it makes sense to generate vegetation in compute shaders too. So each of the four steps of vegetation generation in Figure 4.9 represents a compute shader.

Invocations of compute shaders are dispatched in work groups. Invocations within a group can communicate efficiently but the size of the group must be known beforehand. The amount of work in each step varies so the count of work groups of each step varies too:

Step 1: Based on the biome in the chunk, it is decided how many specimens of which species there will be. There is one work group per chunk. The output of the compute shader is an array of vegetation instances. Vegetation instance can be thought of as a plant seed. It identifies its species (the L-system that will be used to generate it). It stores its position in the world and some characteristics of the surrounding world which further affect the L-system expansion, e.g. 'richness' of the soil, gradient of the terrain.

Step 2: Each vegetation instance then has its L-system expanded and interpreted into a series of branch segments. There is one work group per vegetation instance of this shader. The interpreted segments are not stored into main buffer because there must be exactly one allocation of branches per chunk. So the branches are instead stored into an auxiliary buffer and a sum of branches per chunk is calculated.

Step 3: There is always exactly one work group of the third step. It takes the count of branches per chunk and its sole purpose is to allocate space for the branches in the main buffer using Algorithm 4.1. It outputs offsets into the main buffer.

Step 4: The final step copies the branches from auxiliary storage into the main buffer. There is one work group per branch because it also generates raster representation of the branches.

4.2.4 Parallel sway simulation

Section 4.1.2 describes that there are three forces applied to each branch segment to make it sway: an external wind force, a spring-like force which pulls it back to rest angle and a gravitation force.

A problem is that the wind force and the gravitation force should be composed of not only the contribution of the single segment, it should include contributions of all its children segments. Especially the wind force is problematic because it changes every step so it cannot be precomputed like gravitation force.

The sum of forces can be calculated by sweeping from leaf segments towards the root segment [11]. This sweep is, however, not trivially parallelizable because the threads running in parallel need to cooperate on which segment to sweep next.

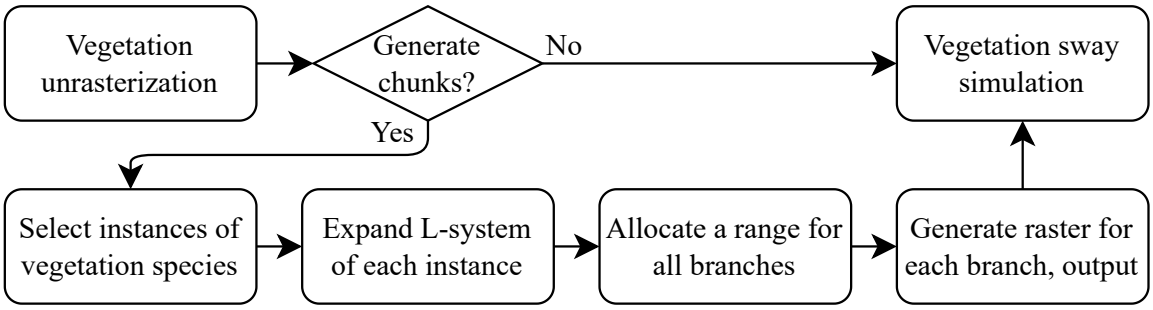


Figure 4.9: The generation of a chunk consists of tile generation (not shown in the figure) and vegetation generation. The procedure of generation of vegetation for a chunk has four steps, all of which are done on GPU by compute shaders.

This project aims to perform the simulation in a shader on GPU so, ideally, the simulation of each segment should be completely independent on other segments. To achieve this, this project's sway simulation does not consider contributions of children segments. This simplification makes the simulation less physically accurate but the difference between the correct simulation and the simplification is not big enough to be noticeable in the project.

In fact, yet another inaccuracy is introduced to make the simulation trivially parallelizable. A segment needs its parent's position and orientation to calculate their relative angle and the resulting spring force. But the segment and its parent are updated in parallel without any synchronization so it is unknown whether the parent's orientation is already updated or not.

The race condition is avoided by double buffering the array of branch segments (array $\boldsymbol{S}$ in Subsection 4.2.2) and performing the simulation from one buffer to another. This is a common solution but it means that a segment moves according to the parent's state in the previous step. Figure 4.10 shows that a result of this is that the tip of a parent branch and the base of its child segment do not meet precisely.

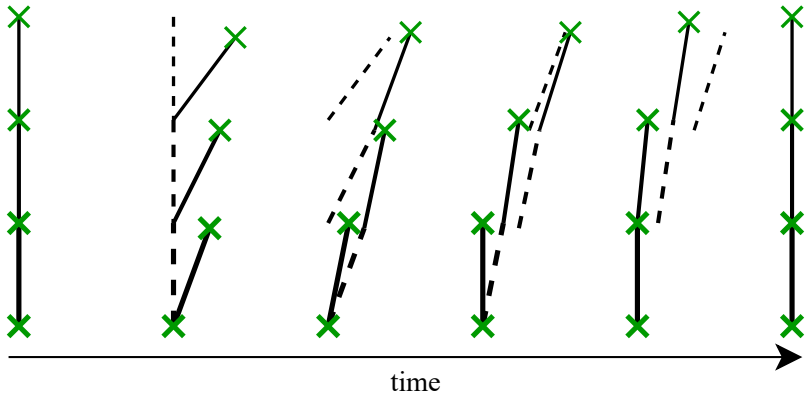


Figure 4.10: A branch segment (solid line) moves according to the position and orientation of its parent in the previous step (dashed line). This makes the simulation trivially parallelizable but related branches do not meet precisely when moving. The figure shows the effect on reaction of three branches to single blow of wind to the right.

On the other hand, the artifacts shown in Figure 4.10 are very exaggerated. They are not visible in the actual project for two reasons. Firstly, the simulation runs at 100 steps per second so the differences between the current and previous step's positions are much smaller. Secondly, the branches are small in the resolution of the world texture where they are rasterized to. So the large tiles hide the inaccuracies, too.

4.2.5 Interaction between vegetation and the tiled world

As Subsection 4.2.1 describes, the vegetation is simulated by removing the vegetation from the world texture (storing its tiles in a auxiliary raster), updating its vector representation (to simulate sway), and adding it back to the world texture (reading its tiles from the auxiliary raster).

The repeated movement of the branches' tiles between the world texture and the side raster is the key principle which connects the sway simulation with the cellular simulation.

Curved branch segments

Firstly, the branches are not rasterized as simple straight line segments but thick Bézier curves. The thickness is for visualization of wide tree trunks while the Bézier curves improve fidelity of thin long branch segments. Figure 4.11 compares rasterization with straight line segments to Bézier curves.

(a) Straight line rasterization (b) Bézier curve rasterization

Figure 4.11: The difference between visualization of branches as straight lines and Bézier curves is the most noticeable in long thin segments, such as tall grass or willow withies.

3-control-point Bézier curves were chosen over more advanced parametric curves because they are very cheap to compute. It takes only 9 additions and multiplications to evaluate the curve [12] which is a important since every branch segment is converted to the curve twice every simulation step (during rasterization and unrasterization). The same effect of curved branches could be achieved by generating more shorter segments. From this point of view, the usage of Bézier curves can be viewed as an optimization to reduce the total number of branch segments that need to be simulated while achieving similar looking results.

Figure 4.12 shows how the curves are constructed. A consequence of using only 3-control-point curves is that a parent's and its child's curves do not meet tangentially. Even tough this could be improved by using four control points, it turned out to be unnoticeable in the resolution of the world texture. Another consequence of using curves is that the

length of a branch changes slightly based on the relative angle to its parent. This is, however, also unnoticeable so no measures were taken to compensate for this.

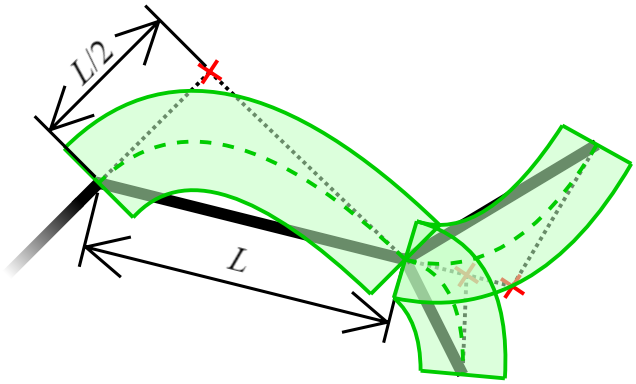


Figure 4.12: Even though the simulation system considers each branch segment as straight rod, they are visualized as thick 3-control-point Bézier curves. The endpoints are the same as the endpoints of the rods. The middle control point (red cross) is constructed by extending the parent's rod by half of the length of the current branch. These three control points create the axis of the visualized branch (dashed green curve) which is widened perpendicularly to the axis in both directions (solid green) to give them thickness.

The auxiliary raster for tiles of branches

Having the exact shape of branches defined, a very closely related problem is how to store the tiles of a branch in the auxiliary raster. Even when considered that the axis curve is straight so that the shape of the branch becomes a rectangle, each branch segment has different dimensions.

Allocation of such variously sized rectangles from a large raster and defragmentation, which would reuse space of removed branches, would be very hard.

So instead a much simpler solution has been chosen. Every branch segment has a rectangular area of the same dimensions. This makes allocation and deallocation simple because the small constant-size per branch raster can be stored next to vector representation of each branch and it can be managed by the same data structure as described in Subsection 4.2.2.

The constant raster space per branch means that a short or thin branch underutilizes it, while a long or thick branch cannot store all of its in-world-texture tiles uniquely in its auxiliary raster. In other words, multiple tiles of large branch rasterized to the world texture map to a single tile in auxiliary raster.

But, as Figure 4.13 visualizes, these conflicts are inherent to the rasterization-based transformation process between the world texture and the auxiliary raster. Sole rotation of a branch with matching representation in the auxiliary raster is enough to create the conflicts in the world texture. Stretching of branches along the Bézier curves makes such conflicts even more common.

On the other hand, the conflicts are not a major problem. The rules of cellular simulation change tiles of branches to simulate fire which behaves somewhat unpredictably. Thus, this level of unpredictability (whether the normal wood tile or the burning wood tiles is stored back) is not an issue.

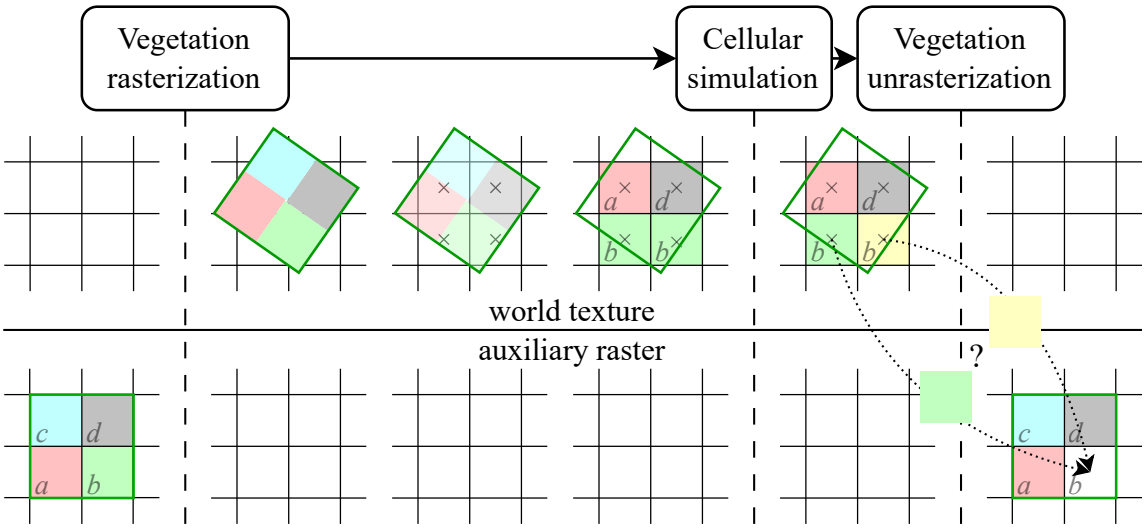


Figure 4.13: The rasterization of branches from auxiliary raster to the world texture is not bijective. The figure shows an example where tile b was rasterized into two tiles in the world texture. If cellular simulation, which works with the tiles in the world texture, modifies one of them, a conflict appears. It is unclear which of the values should be put back into tile b in the auxiliary raster.

Leaves

So far only the structure of branches was considered to be visualized in the world texture. But all trees and bushes would look dead. So this subsection explains how leaves are simulated and visualized to make the vegetation appear alive.

Two approaches to simulation and visualization of leaves were considered:

- Leaves appear around tiles of branches as a result of a cellular simulation.
  This approach turned out to be unfeasible because to create a natural crown of a tree, there must be a lot of leaves around thin branches near top of the tree and no leaves growing directly from the trunk. But individual wood tiles cannot store width of the branch precisely.

  Another problem of this approach was that it produced uniformly wide strips of leaves around the branches. This looked unnatural as leaves tend to grow in clumps in nature.

- Leaves are rasterized similarly to branches.
  The problematic parts of this approach is how to rasterize the leaves so that clumps are created and how to connect a clump to the movements of the branch that it is attached to. The best shape of a clump would be an oval or an ellipse but rasterization of such shapes requires a lot of vertices even in the low resolution of the world texture.

As neither of the above approaches is flawless, the designed solution is a fusion of both rasterization and cellular simulation.

The origin of leaves is in raster representation of branches. There is a special type of wood tile, called *bud*, that spawns a *seed leaf* tile every step during its rasterization. The position of the seed leaf is relative to the position of the bud and orientation of the branch which it comes from.

A clump of leaves will grow from each seed leaf. Similarly to real buds, a bud has several levels of development. The development level is also stored in its seed leaf tile which in result determines how big its clump will be.

Leaves are not unrasterized like branches, they are managed by rules of cellular simulation. The rules for a leaf tile can be described as:

1. Reduce the leaf's power (level of development).
2. If zero power is reached, turn it into air.
3. Otherwise, put leaves to all air tiles in 4- or 8-neighborhood with same power as the center leaf.

By carefully adjusting the initial power of seed leaves and the reduction coefficient, a balance can be reached that produces big enough round clumps that also disappear quickly at its edge when the center seed leaf moves. This means that, since seed leaves are put relatively to the current orientation of its bud tile, the clumps also tightly follow the sway of branches. Figure 4.14 visualizes the process of leaf clump growth.

Fire, dropping of leaves and other cellular simulations

Having set up the rasterization procedures, phenomena affecting the tiles of the vegetation are simulated simply by adding new rules to the already existing cellular simulation systems. For example, burning of vegetation is simulated by a few rules such as:

- If this tile is wood and there is a burning tile nearby, turn it into burning wood.
- If this tile is burning wood, turn it into burnt wood with a small probability.
- If this tile is a leaf and there is burning nearby, turn it into a fire tile.

Similar rules are developed to make vegetation drop leaves or to make the branches decay when blighted by a mold.

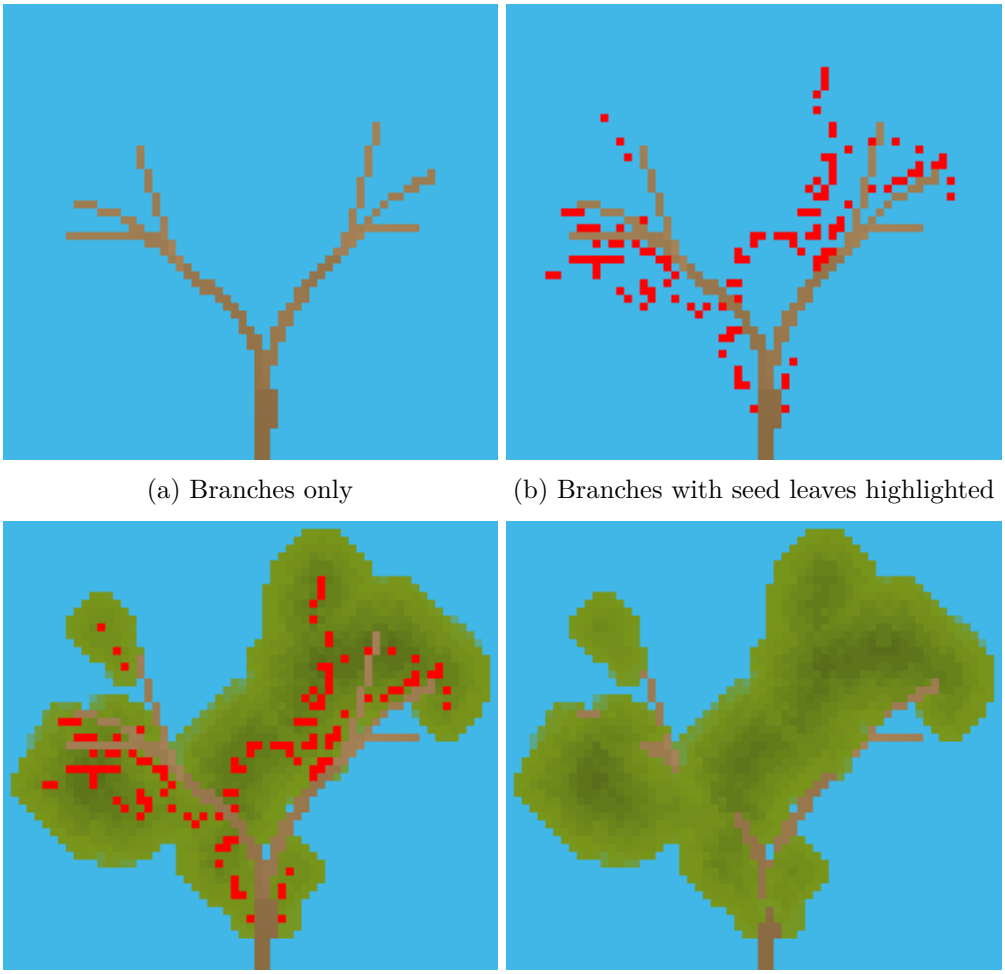
4.3 Implementation: Efficient generation and simulation of the vegetation

Nearly everything designed in the previous section is implemented using Vulkan graphics and compute shaders. This section highlights some interesting details of the implementation.

Subsection 4.3.1 covers problems with programming of L-systems in GLSL which does not support strings natively. Subsection 4.3.2 describes allocation of branches in shaders and a procedure that is needed to allocate branches from CPU. Subsection 4.3.3 reveals that branch sway simulation and rasterization is merged into a single step to reduce memory accesses.

4.3.1 Expanding L-systems in shaders

All steps of generation of vegetation, described in Subsection 4.2.3, are implemented in compute shaders. This also includes expansion and interpretation of L-systems. This subsection describes some interesting details of the implementation.



(a) Branches only (b) Branches with seed leaves highlighted

(c) A crown grown from seed leaves (d) Final crown partially covering branches

Figure 4.14: A tree's crown is formed by the following procedure: Each bud tile produces a single seed leaf tile. A cellular simulation then spreads leaves in proximity of the seed leaves so clumps of leaves are created. Likewise, non-seed leaves that have no seeds nearby are almost immediately replaced by air. This ensures that leaves follow swaying branches tightly and do not appear to float nearby regardless of the wind.

Strings in GLSL

Working with characters and strings in GLSL, the shader language used to implement the shaders, is cumbersome for a few reasons. Firstly, there is no character type in the language at all. 8-bit integers[1] are the closest type to a character but it does not allow conversion from a character literal.

Working with strings is even more inelegant than characters alone because there is no pointer type so it is not possible to represent strings by a pointer to the first character. There is an extension[2] which provides types similar to pointers of the C language but it is intended for a different purpose and is not usable in this context because the L-systems are expanded in shared memory. Shared memory is a special type of memory that compute

[1] Bit-sized integers are provided by extension `GL_EXT_shader_explicit_arithmetic_types`.

[2] Limited pointer types are provided by extension `GLSL_EXT_buffer_reference`.

shaders use to communicate within a work group. It is not possible to take address of an object in the shared memory so it is not possible to create a pointer to it either.

Thus, the only way to represent a string is a simple integer denoting an offset from the beginning of a larger array of 8-bit integers. This, unfortunately, becomes confusing when there need to be multiple of such arrays — in the shared memory and an uniform buffer, for example.

Serialization of L-systems for use as input in shaders

Another challenge of L-systems in shaders is the form of providing of their definition. A stochastic parametric L-system, as described in Subsection 4.1.1, is defined by the axiom string and a set of rewrite rules. A string is a sequence of characters (symbols) and a different-sized sequence of numeric parameters of the symbols. A rewrite rule is represented by the head character (predecessor), a condition (limited to a single comparison of a parameter with a constant) and two sets of rule bodies (successors) with a probability assigned to each body. One set of bodies is used when the condition is true, the other one is used when it is false.

This is a lot of variable-sized parameters to define an L-system and there are multiple species of vegetation in the project so there are multiple L-systems. All this information must be somehow serialized into a uniform buffer to be used as input in the shader.

Developing L-systems by altering this data structure manually would be extremely tedious because, for example, changing the length of an axiom string shifts all other strings in the array of symbols behind it. So the offsets denoting the other strings should be adjusted.

To avoid this, the data structure is created by serializing other smaller structures that are simple to define by a programmer. Capabilities of modern C++ are utilized to perform the serialization at compilation time[1]. It can also validate that, for example, the number of symbols in a string corresponds to the number of parameters.

4.3.2 Allocation of branches in shaders

As Subsection 4.2.2 explains, the data structure used for allocation of branches has three parts: the main (double-buffered) array of branch segments $\boldsymbol{S}$, an array of allocation ranges $\boldsymbol{R}$ and a small index $\boldsymbol{I}$ which speeds up chunk deallocation.

It is necessary to store the main array $\boldsymbol{S}$ in GPU memory because it is accessed by shaders every step to simulate the vegetation. On the other hand, allocation ranges $\boldsymbol{R}$ and the index $\boldsymbol{I}$ are modified only when a chunk is activated or deactivated.

Chunk activation or deactivation are procedures initiated by CPU so it would a viable option to store the structures in CPU memory and perform the allocation (Algorithm 4.1) and deallocation (Algorithm 4.2) by CPU. However, it is also needed to allocate branches when a new chunk is generated and that is done by shaders. Because of this, the whole data structure is in GPU memory.

It also allows to dispatch rasterization (or unrasterization) of all branches with a single indirect draw call. An indirect draw call takes some its arguments from GPU memory. The indirect draw call used here, `vkCmdDrawIndirect`, indirectly takes an array of structures, `VkDrawIndirectCommand`, which denote the index of the first vertex and the count of the vertices.

[1] All functions involved in the serialization are `constexpr`-qualified.

This maps elegantly to the array of allocation ranges $\boldsymbol{R}$ because each allocation range in $\boldsymbol{R}$ has memory layout compatible with structure expected by the indirect draw command. This results in one invocation of vertex shader per branch of each occupied range (as it is hinted in Figure 4.8).

A downside of allocation of branches in shaders is that when CPU needs to allocate or deallocate, it has to schedule a shader that will do it. Figure 4.15 shows a timeline of four steps that happen when activating a chunk with branches. The procedure could be simplified to two simulation steps by usage of an indirect copy command but that is provided by an extension[1] implemented only on NVIDIA GPUs so it was not used.

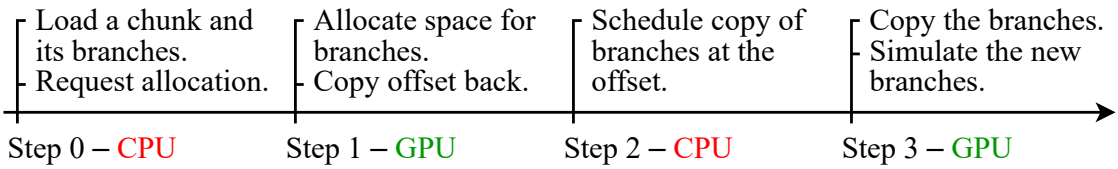


Figure 4.15: The procedure of activation of a chunk with branches spans four simulation steps. It is divided into steps to avoid synchronization between CPU and GPU within a simulation step which could cause hitching. The system handles multiple of such procedures for different chunks to be run in parallel.

Mutual exclusion must be ensured when allocating and deallocating branches. This is done by a mutex implemented with atomic instructions and appropriate pipeline barriers between allocation for newly generated chunks and allocation for chunks activated from CPU.

4.3.3 Combined simulation and rasterization pipeline

The simulation of vegetation was described to have three main steps: unrasterization of branches from the world texture, updating vector representation to simulate sway and rasterization of the branches at the new positions.

The implementation optimizes this and performs both sway simulation and rasterization in a single graphics pipeline. The pipeline utilizes all shader stages. Figure 4.16 visualizes the geometrical operations of the vector stages of the combined pipeline. Description of each shader stage follows:

1. **Vertex shader**
   There is one vertex shader invocation per branch segment. The shader loads the branch and its parent from the first array of segments (see Subsection 4.2.4), calculates new position and rotation, stores the new values into the other array of segments and also forwards the new values to next stage.

2. **Tessellation shaders**
   Tessellation control shader constructs the three control points of the Bézier curve (see Figure 4.12). It uses quad topology that is tessellated into a triangle strip. The number of triangles in the strip is proportional to the curvature and length of the branch.

[1] Indirect copy commands are provideded by extension `VK_NV_copy_memory_indirect`.

Tessellation evaluation shader calculates its position along the curve and offsets half the branch's width perpendicularly to the axis.

3. **Geometry shader**
   Geometry shader is used to suppress an artifact that would appear when branches cross an edge of the world texture. Branches are placed into the world texture according to the same equation as chunks (see Equation 2.1). This means that a triangle crossing an edge should be partly on one side of the world texture and partly on the opposite side. But the rasterizer simply clips it within the bounds.

   Thus, the geometry shader takes a triangle and, if it crosses an edge, duplicates it on the other edge by shifting by the size of the world texture. In fact, the worst case scenario is that a triangle is duplicated four times when it is in a corner of the world texture. On the other hand, no duplication is needed for overwhelming majority of triangles that do not cross an edge so the shader forwards these unchanged.

   It was also experimented with an alternative solution without a geometry buffer. In the alternative solution, the logical framebuffer was bigger than the world texture so the triangles were not clipped. The fragment shader then adjusted its position (using Equation 2.1) to be inside the world texture and used image load and store to access the world texture instead of render output. This solution is not used because it exhibits strong artifacts in areas where multiple branches overlap as there is no ordering guarantee by the render output.

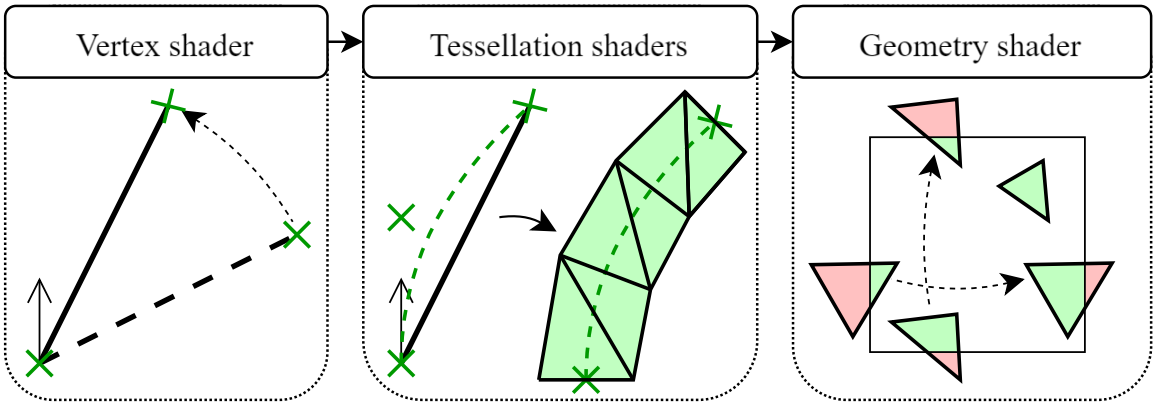


Figure 4.16: Vector shaders of combined simulation and rasterization pipeline do the following. The vertex shader loads the branch and its parent (green crosses), simulates sway of the branch, stores the updated values and also forwards them to the next stage. Tessellation shaders construct a Bézier curve out of the control points and tessellate it using a triangle trip. Geometry shader duplicates triangles that cross an edge on the other side. This ensures that seams on edges of the world texture are not visible.

4. **Fragment shader**
   Fragment shader uses subpass input. Subpass input is a resource available in Vulkan fragment shaders. It is similar to texture read but the texture is always read at viewport coordinates of the fragment. If the subpass input texture is the same as the render output texture, it guarantees that the input is coherent with render output of other primitives within the draw call.

The fragment shader uses subpass input to load the previous tile on its position in the world texture. If it is a solid tile, such as stone or dirt, it outputs the tile using render output so that it stays unchanged.

If the previous tile is not solid, it loads its corresponding tile from the branch's auxiliary raster and outputs it using render output. Also, if the wood tile is a bud, it puts a seed tile into the world texture using image store.

Using render output and image store on a texture simultaneously is unsafe in Vulkan [5, sec. Render Pass Store Operations] so this deserves a better solution. However, no artifacts were spotted because of this on any of tested hardware and a cleaner solution would, arguably, be slower.

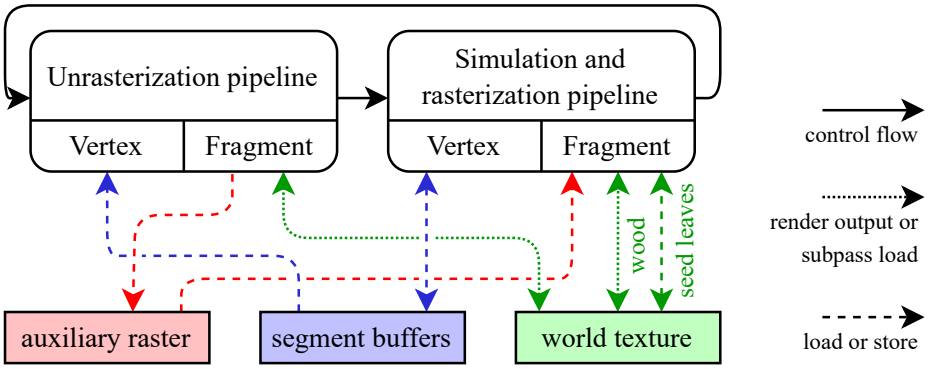


Figure 4.17: The graphics pipelines that simulate and rasterize branches have multiple side inputs and side outputs so they go beyond a classical pattern where the vertex shader reads some buffers and the fragment shader writes using render output. Legend on the right explains what each arrow means. Shader stages that do not access global memory are omitted.

Figure 4.17 summarizes accesses to global memory (buffers and textures on GPU) of the combined and unrasterization pipelines. The benefit of combining the sway simulation and rasterization over a solution where the simulation would be a separate compute shader is that the vector representation of branches is loaded only once from global memory, whereas the alternative would load it in the compute shader and, second time, in the graphics pipeline. It also naturally parallelizes the two steps, whereas parallelization of separate pipelines would be more complicated.

Chapter 5

Performance assessment

Performance of the project is analyzed from two perspectives: Section 5.1 compares overall performance of the project compared to the version of the project prior to this thesis. Section 5.2 analyzes a GPU trace to understand the factors limiting the performance.

5.1 Performance comparison with version prior to this thesis

The goal of this section is to compare performance of the current version of the project and the version prior to this thesis. To asses performance of the project in a reproducible way, a slightly modified version of the project is used. The modified version does not expect any user input. Instead, it creates a world with a preset seed and starts moving the view inside the world at a constant speed. The test stops after 30 seconds and calculates the average number of frames per second rendered (FPS) over the time.

By moving the view of the world, the test assesses all aspects of the project. It causes new chunks to be generated, including vegetation, so simulation of vegetation is assessed, too. Because the test runs long enough to move through the whole world texture, it also causes deactivation of chunks to happen.

The same modification is also done to the OpenGL version of the project so the view also moves at the same speed along the horizon, forcing chunks to be generated etc.

Table 5.1 shows specifications of five machines where the performance tests were run. All tests were measured at full screen 1920×1080 resolution.

CPU	GPU	GPU memory
Intel Core i3-4160	NVIDIA GeForce GTX 750	1 GiB GDDR5
Intel Core i5-8300H	NVIDIA GeForce GTX 1050	4 GiB GDDR5
Intel Core i3-10100	NVIDIA GeForce GTX 1660 Super	6 GiB GDDR6
Intel Core i7-12700H	Intel Iris Xe Graphics	32 GiB DDR4
Intel Core i7-12700H	NVIDIA GeForce RTX 3060	6 GiB GDDR6

Table 5.1: Performance of the project is analyzed on five different configurations. Intel Iris Xe is the only CPU-integrated graphics chip, the other are dedicated GPUs. The release date of the graphics chips ranges from 2014 to 2022 so their computational power varies greatly, too. Abbreviations of GPUs are used as identifiers of the machines.

The performance test was run five times on every of the machines, Figure 5.1 shows the results. It shows that, even tough the current version simulates vegetation which the

original version did not do, it still outperforms the original version by more than 50 % on all configurations that were tested.

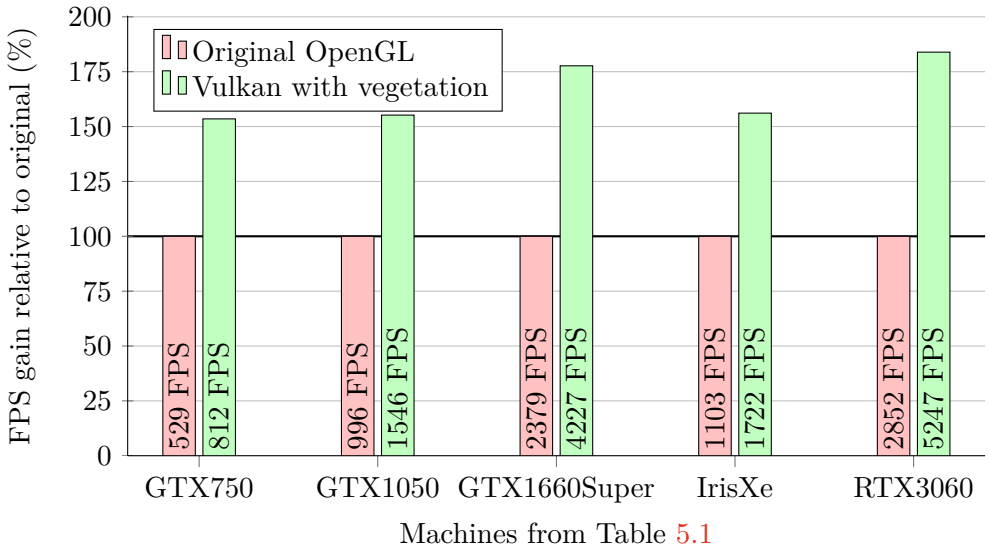


Figure 5.1: The current version of the project generates and simulates vegetation which the original did not do. Despite this unfair comparison and thanks to optimizations implemented in this thesis, it performs better by more than 50 % on all configurations that were tested.

5.2 GPU trace analysis

Since all computationally demanding aspects of the project are offloaded to GPU and CPU only performs the orchestration, it is expected that the performance is bound by the GPU. NVIDIA Nsight Graphics[1], a profiling tool for NVIDIA GPUs, was utilized to prove this and analyze the limiting factors inside the GPU. The analysis was done on GTX1660Super machine from Table 5.1.

The simulation of the project always runs at 100 steps per second, the remaining time is used for rendering of frames. This is important for understanding of the GPU trace shown in Figure 5.2. It shows that the rendering takes much shorter time than a simulation step. Thus, after performing a more demanding simulations step, the GPU is capable of rendering many frames very quickly until the next simulation step should be carried out. This explains why FPS may be so high in the performance tests.

It can also be seen that calculation of dynamic shadows takes the majority of the simulation step. It is questionable whether this calculation should be part of simulation or rendering. The reason why it is in simulation is that every tested configuration was capable of rendering at much more than 100 FPS. This means that the calculated shadows are rendered multiple times. If the calculation was part of rendering, it would be recalculated multiple times with the same input world texture.

It was analyzed further why the shadow calculation takes so long compared to the rest of the simulation step. Figure 5.3 shows a zoom in of the trace at the simulation step with

[1] Available at: https://developer.nvidia.com/nsight-graphics

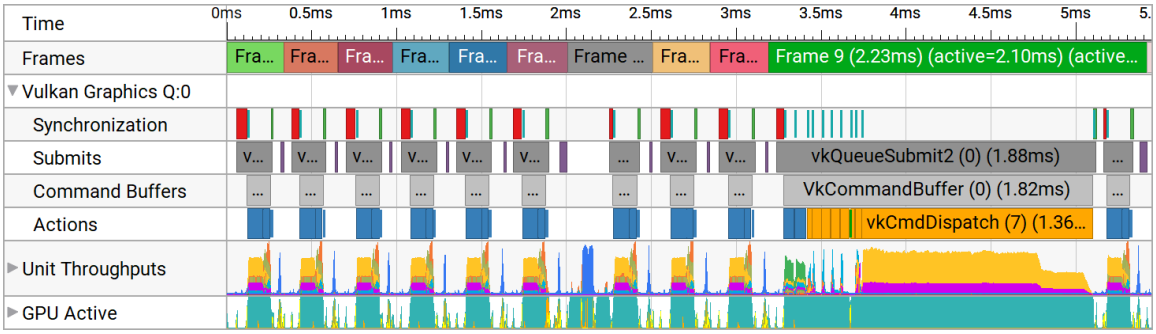


Figure 5.2: The picture shows a GPU trace of 10 frames of the project. The trace shows that a simulation step, done by `vkQueueSubmit2(0)` in Frame 9, takes significantly longer than rendering of frames. Metric `GPU Active` shows that the GPU remains busy, except for short intervals ($\approx$ 0.1 ms) of internal overhead when a rendered frame is being made available for presenting.

memory access metrics. The metrics show that, while the calculation is limited by reading of the input textures of the algorithm (see Figure 3.3), the textures are small enough to fit into caches so the penalty of the reading is not as high.

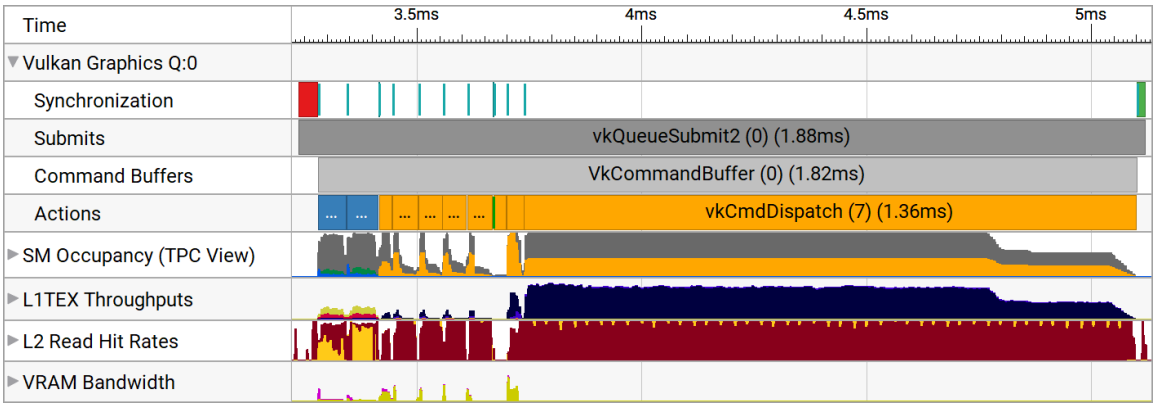


Figure 5.3: The GPU trace in the picture shows that majority of a simulation step, `vkQueueSubmit2(0)`, is spent in calculation of shadows, `vkCmdDispatch(7)`. It can be seen that it reaches about 70 % of L1 texture cache utilization and almost 100 % L2 cache hit rate. The yellow portion of `SM Occupancy` metric shows active warps in multiprocessors (cumulative), the gray portion represents state where all warps of a multiprocessor were waiting for the texture reads.

The traces show that the optimization presented in this thesis, aimed to reduce reads from the input textures, is a step in good direction. The texture reads in shadow calculation remain the bottleneck so it is still the main procedure to optimize in the project. But, from the absolute time of a simulation step (less than 2 ms), it can be concluded that there is plenty of room to implement new features into the project.

Chapter 6

Conclusion

The goal of this thesis was to add vegetation to a game demo. The game demo prior to this thesis generated two-dimensional world from side view with biomes spread across the horizon.

This thesis added trees, bushes and tall grass into the world. The generation of the vegetation respects climate of the biomes so the range of species spans from cactuses in desert to oaks in temperate forests, spruces in cold taigas and more. The generated vegetation is not static. It is simulated to sway in the wind, deciduous species drop leaves, it can be set on fire and burn.

The main technical challenge of the thesis was the fact that the vegetation is represented by vector-based branch segments but the rest of the world is raster-based grid of tiles. Rasterization into the grid of tiles is used to connect the two representations. It also makes vegetation appear to be made of tiles so it fits into the graphical style of the project.

The thesis focused on efficient implementation so both generation and simulation of the vegetation is parallelized on GPU. Calculation of dynamic shadows is also improved and the whole application was transitioned to a modern graphics programming interface. The performance was tested on a number of machines and it improved by more than 50 % on every of the machines.

The project could be improved in many ways. It would be very interesting if branches, cut off by the player or broken off by strong wind, fell to the ground naturally. However, the simulation of collisions between the branches and the raster world would be challenging. It would also be interesting if the vegetation grew gradually, regrew the removed branches or even spread with seeds. With a simple weather simulation, the growth could be controlled by supply of rainwater.

Bibliography

[1] Barron, J. T.; Sorge, B. P. and Davis, T. A. Real-Time Procedural Animation of Trees. In: Roberts, J. C., ed. *22nd Annual Conference of the European Association for Computer Graphics, Eurographics 2001 - Short Presentations, Manchester, UK, September 3-7, 2001*. Eurographics Association, 2001. Available at: https://doi.org/10.2312/egs.20011034.

[2] Devlin, J. and Schuster, M. D. Probabilistic Cellular Automata for Granular Media in Video Games. *The Computer Games Journal*, Jun 2021, vol. 10, no. 1, p. 111–120. ISSN 2052-773X. Available at: https://doi.org/10.1007/s40869-020-00122-4.

[3] Dubský, T. *Procedural Generation and Simulation of 2D Gaming World*. 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Chlubna, T. Available at: https://www.vut.cz/en/students/final-thesis/detail/145096.

[4] Holdrege, C. The forming tree. *In Context*. The Nature Institute, Fall 2005, no. 14, p. 18–22. Available at: https://www.natureinstitute.org/article/craig-holdrege/the-forming-tree.

[5] Khronos. *Vulkan® 1.3 - A Specification*. The Khronos® Vulkan Working Group, 2023. Available at: https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html.

[6] Lipp, M.; Wonka, P. and Wimmer, M. Parallel generation of multiple L-systems. *Computers & Graphics*, 2010, vol. 34, no. 5, p. 585–593. ISSN 0097-8493. Available at: https://doi.org/10.1016/j.cag.2010.05.014. CAD/GRAPHICS 2009 Extended papers from the 2009 Sketch-Based Interfaces and Modeling Conference Vision, Modeling & Visualization.

[7] Lujan, M.; Baum, M.; Chen, D. and Zong, Z. Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server. In: *2019 International Conference on Computing, Networking and Communications (ICNC)*. 2019, p. 777–781. ISBN 978-1-5386-9223-3. Available at: https://ieeexplore.ieee.org/document/8685588.

[8] Nystrom, R. Game Loop. In: *Game Programming Patterns* online. 1st ed. Genever Benning, 2014, chap. 3. ISBN 978-0990582908. Available at: https://gameprogrammingpatterns.com/game-loop.html.

[9] 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.

[10] Prusinkiewicz, P. and Lindenmayer, A. *The algorithmic beauty of plants*. 1st ed. Springer-Verlag, 1996. ISBN 978-1-4613-8476-2. Available at: http://algorithmicbotany.org/papers/abop/abop.pdf.

[11] Quigley, E.; Yu, Y.; Huang, J.; Lin, W. and Fedkiw, R. Real-Time Interactive Tree Animation. *IEEE Transactions on Visualization and Computer Graphics*, May 2018, vol. 24, no. 5, p. 1717–1727. Available at: https://doi.org/10.1109/tvcg.2017.2661308.

[12] Vince, J. Mathematics for Computer Graphics. In: Mackie, I., ed. 6th ed. Springer, 2022, chap. Bézier Curves, p. 303–307. ISBN 978-1-4471-7520-9. Available at: https://doi.org/10.1007/978-1-4471-7520-9.

[13] Wesslén, D. and Seipel, S. Real-time visualization of animated trees. *The Visual Computer*, Jul 2005, vol. 21, no. 6, p. 397–405. ISSN 1432-2315. Available at: https://doi.org/10.1007/s00371-005-0295-1.

Appendix A

Sample images of the application

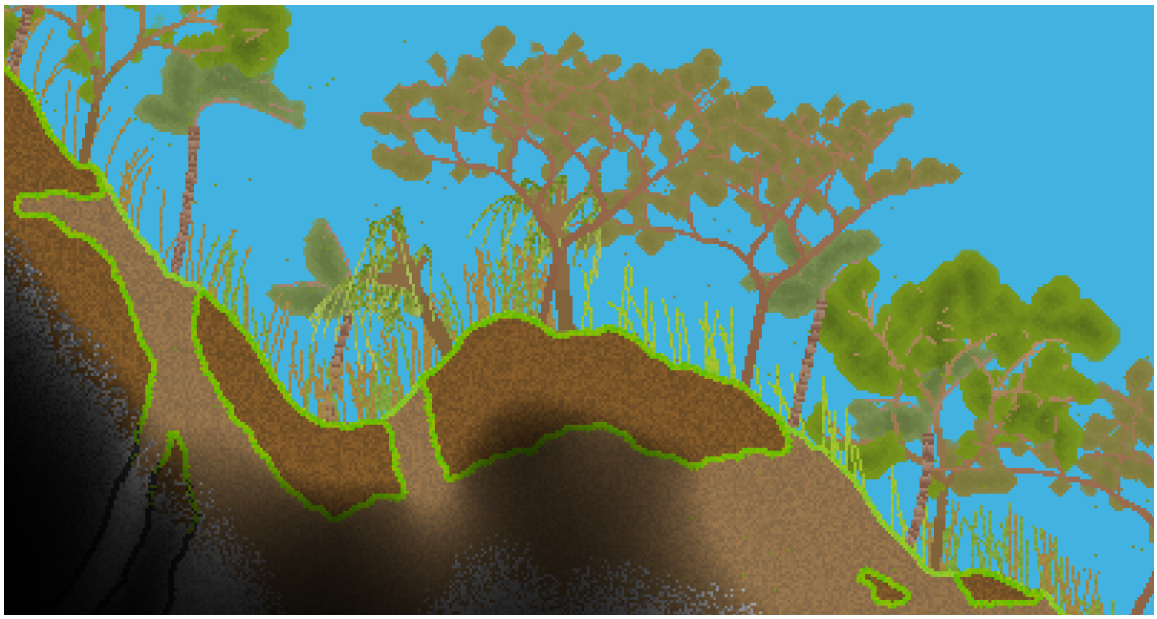


Figure A.1: The picture shows a jungle biome composed of palm trees, acacias, willows, oaks and tall grasses.

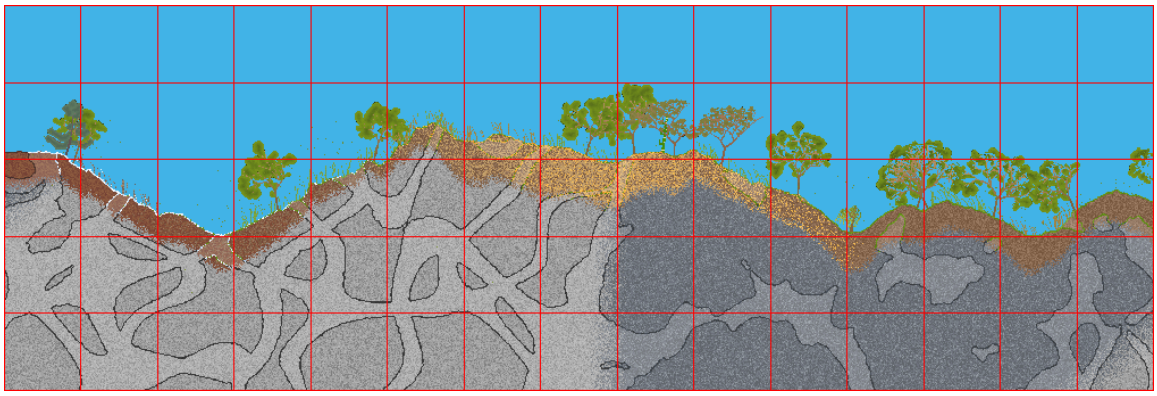


Figure A.2: The picture shows a map of a generated world. Red lines highlight boundaries between chunks. The map does not show shadows that would normally be visible.

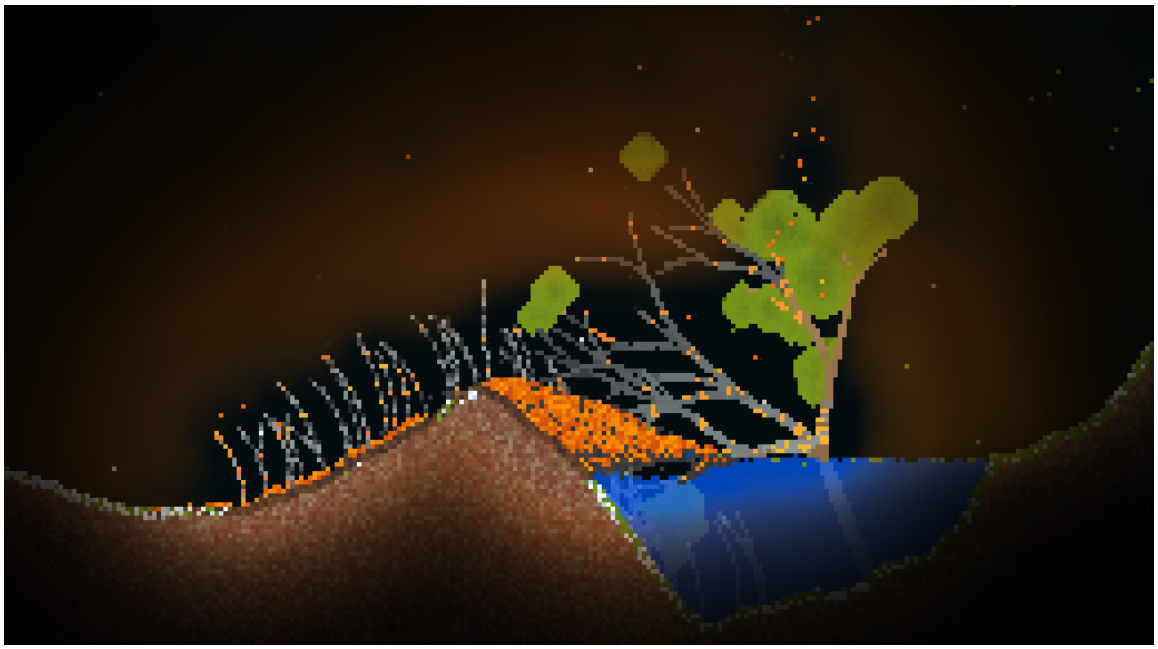


Figure A.3: The picture shows burnt tall grass and partially burnt oak tree. The fire is caused by the spilled lava. Branches submerged in water do not burn.

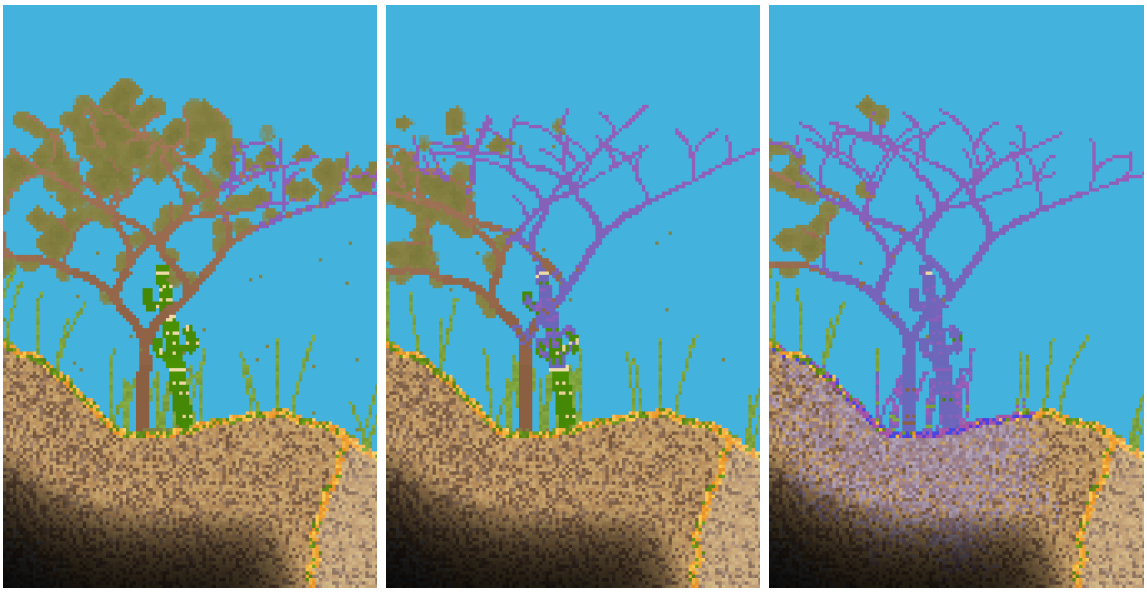


Figure A.4: The images shows purple mold spreading from acacia to cactus and nearby grass. The images are taken about ten seconds of game simulation time from each other. It can be seen that the infected branches loose leaves.