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

GRAPHICS INTRO 64KB USING OPENGL

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **IVO MEIXNER**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET**
VEDOUCÍ PRÁCE

BRNO 2021

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Bachelor's Thesis Specification

23692

Student:	**Meixner Ivo**
Programme:	Information Technology
Title:	**Graphics Intro 64kB Using OpenGL**
Category:	Computer Graphics

Assignment:

1. Familiarize yourself with the topic of graphic intro with limited size.
2. Study the OpenGL library and its extensions
3. Describe selected techniques applicable in a limited-size graphics intro.
4. Implement a graphical intro using OpenGL. The size of the executable must not exceed 64kB.
5. Evaluate achieved results and suggest possibilities for the continuation of the project; create a video to present the project.

Recommended literature:

- according to supervisor instructions

Requirements for the first semester:

- Items 1, 2 and 3 and the core of the application.

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

Supervisor:	**Milet Tomáš, Ing.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 12, 2021
Approval date:	April 22, 2021

Abstract

The goal of this thesis was to implement and describe a computer graphics intro using OpenGL, with an executable file size no greater than 64 KiB. It renders a 3D animation imitating a natural environment by combining various techniques into a single scene. The intro was implemented to run on Microsoft Windows using simplex noise, L-systems and several shader programs written in GLSL, including a compute shader for L-system instruction processing. The final executable size is below 8 KiB. It is capable of rending a simple nature-like scene of a mountainous terrain with scarce vegetation, reflective bodies of water and a sky that changes based on a time of day.

Abstrakt

Cílem této práce byla implementace a popis grafického intra pomocí OpenGL, s velikostí spustitelného souboru do 64 KiB. Vzniklý program vykresluje 3D animaci imitující přírodní prostředí pomocí kombinace různých technik do jedné scény. Intro bylo implementováno pro spouštění na Microsoft Windows s využitím simplexního šumu, L-systémů a mnoha shaderových programů napsaných v GLSL, včetně compute shaderu pro zpracování instrukcí L-systémů. Výsledná velikost spustitelného souboru je nižší než 8 KiB. Tento program je schopen vykreslovat jednoduchou scénu na styl přírodního prostředí s hornatým terénem, řídkou vegetací, vodními plochami odrážejícími okolní scénu a oblohou měnící se v průběhu dne.

Keywords

intro, demoscene, computer graphics, OpenGL, GLSL, simplex noise, L-systems, procedural generation

Klíčová slova

intro, demoscéna, počítačová grafika, OpenGL, GLSL, simplexní šum, L-systémy, procedurální generování

Reference

MEIXNER, Ivo. *Graphics Intro 64kB Using OpenGL*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet

Rozšířený abstrakt

Cílem této práce bylo vytvořit grafické intro s maximální velikostí 64 KiB. Intro je druh počítačové aplikace vykreslující animaci v reálném čase se striktním omezením maximální velikosti spustitelného souboru. Typicky je vykreslována trojrozměrná scéna s různými vizuálními efekty, často doplněná o hudební podklad syntetizovaný za běhu programu. Velikost je obvykle omezena na mocninu dvou bajtů, přičemž nejčastější hodnoty jsou 256 B, 1 KiB, 4 KiB a 64 KiB. Pro tento projekt byla zvolena přírodní scéna, které umožňuje aplikaci mnoha různých technologií počítačová grafiky. Syntetizace zvuku byla z tohoto projektu vynechána, jelikož techniky používané pro ozvučení intra nesouvisí s grafickými algoritmy, na které je práce zaměřena.

Jako cílová platforma byl zvolen operační systém Microsoft Windows na osobních počítačích s použitím OpenGL pro vykreslování. Hlavním důvodem byla prakticky univerzálně dostupná zabudovaná knihovna pro práci s OpenGL v instalacích Windows a velmi dobrá podpora nejnovějších ovladačů grafických karet, které umožňují používání nové OpenGL funkcionality. Intro je založeno na šabloně `i1k_OGLShader`[1], kterou zveřejnil Inigo Quilez na svém webu pro volné použití. Pro implementaci byly použity programovací jazyky C a OpenGL Shading Language (GLSL).

Jedna typická digitální fotografie zabírá stokrát až tisíckrát více místa na disku než běžné intro. Aby bylo možné dosáhnout takto nízké velikosti spustitelného souboru, je nutné omezit používání knihoven při vývoji na absolutní minimum a binární objektové soubory komprimovat. Běžné instalace Windows na osobních počítačích obsahují zabudovanou aktuální verzi OpenGL knihoven. Díky této vlastnosti není nutné využití žádné externí knihovny a celé vykreslování lze založit čistě na této zabudované knihovně. Pro lepší omezení velikosti nevyužívají intra typicky ani standardní knihovnu jazyka C, ani matematickou knihovnu, což značně komplikuje implementaci oproti běžným uživatelským aplikacím. Minimalizace velikosti byla u tohoto intra provedena především pomocí nástroje Crinkler[2], který provádí linkování objektových souborů do jednoho spustitelného souboru.

Vykreslovaná animace simuluje časosběrný průlet přírodní krajinou obsahující hory, rostliny, oblohu měnící se v závislosti na čase a vodní plochy odrážející okolní scénu. Terén připomíná hornatou bažinu díky velkému počtu malých vodních ploch. Pro zachování minimální velikosti intra nebyly do spustitelného souboru zabudovány žádné textury, které by byly u obdobných grafických aplikací typicky použity pro oblohu, travnaté plochy, kamení a jiné povrchy. Z tohoto důvodu jsou všechny části scény generovány procedurálně s pomocí šumových funkcí a dalších technik, jako jsou například L-systémy. Zbarvení terénu je dáno jeho výškou, s přidaným simplexním šumem pro méně viditelné a přirozeněji vypadající přechody mezi druhy povrchu.

Terén je pokryt mnoha tisíci drobných rostlin vykreslených pomocí kombinace mnoha technik počítačové grafiky. Jako základ slouží L-systém známý jako *fractal plant*, neboli fraktálová rostlina. V procesorovém kódu je provedeno několik iterací nahrazování řetězců, čímž jsou vygenerovány instrukce pro vykreslování. Vzhledem k chybějící matematické knihovně byl pro interpretaci těchto instrukcí použit compute shader spuštěný na grafické kartě, jehož výstupem jsou souřadnice přímek. Přímé vykreslení těchto přímek do scény by bylo vzhledem k jejich vysokému počtu příliš výpočetně náročné. Přímky jsou proto vykresleny při inicializaci intra do textury, která je následně umístěna mnohotisíckrát na povrch terénu.

[1]https://iquilezles.org/code/isystem1k4k/isystem1k4k.htm
[2]https://github.com/runestubbe/Crinkler

Pomocí šumových funkcí je rostlinám přidán pseudo-náhodný pohyb, který napodobuje efekt větru.

Simulovaný den trvá $16\pi \approx 50.3$ sekund. Během dne se vykresluje modrá denní obloha s mraky, která postupně přechází v noční hvězdnou oblohu a poté zase zpět. V obou případech je pro generování vzorů použita kombinace několika simplexních šumů.

Vodní plocha je jeden velký lichoběžník protínající terén, jehož šířka se zvyšuje se vzdáleností od kamery, čímž kopíruje základní tvar terénu. Plocha je částečně průhledná a odraz okolních objektů se provádí zrcadlením jejich vrcholů podle vodní plochy a následným opakovaným vykreslením pod úrovní vodní hladiny. Odrazy objektů jsou tedy ve skutečnosti vykreslovány skrz hladinu. K barvě odrazů je přidán modrý odstín vody a simplexní šum, čímž je dosažen přirozenější vzhled vodních ploch.

Výsledné intro má velikost spustitelného souboru menší než 8 KiB, což je podstatně méně než stanovené maximum pro tento projekt. V reálném čase je vykreslována animovaná imitace přírodní scény, ve které jsou aplikovány mnohé techniky počítačové grafiky. Výsledek tudíž splňuje cíle stanovené zadáním práce.

Alespoň jedna z dostupných verzí intra by měla být spustitelná na osobním počítači s moderním hardware a aktualizovaným operačním systémem Microsoft Windows a ovladači grafické karty, která musí podporovat minimálně OpenGL 4.3. Vývoj byl proveden na počítači s grafickou kartou z řady Nvidia GeForce, na kterých byla při testování pozorována vyšší podpora náročnějších verzí intra. Na grafických kartách AMD je doporučeno použít méně náročnou verzi kvůli jejich odlišným vlastnostem, které dramaticky snižují výkon při mapování grafické paměti do prostoru procesorové paměti.

Navazující práce by se mohla zabývat implementací dalších L-systémů pro více různých typů vegetace, vylepšením generování terénu pro přirozenější vzhled nebo přidáním mraků na noční oblohu. Dále by bylo možné optimalizovat kód pro vyšší výkon, kompatibilitu s různým hardware a menší velikost než bylo požadováno v rámci této práce.

Graphics Intro 64kB Using OpenGL

Declaration

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

....
Ivo Meixner
May 10, 2021

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Milet for his insightful suggestions, my colleague Martin Kostelník for his assistance with testing and my family for supporting me.

Contents

Chapter 1

Introduction

The goal of this thesis was to create and describe an animation rendered in real-time on a modern personal computer. The program rendering this animation must have the form of a single executable file, which can be run repeatedly without a need to install any extra third-party software that is not a part of a common Microsoft Windows desktop installation.

The size of this file must not exceed 64 KiB[1]. To put this number into perspective, pictures taken using modern digital cameras and smartphones have a typical size between 5 MiB and 10 MiB. This means that a relatively complex animation must be rendered by a program approximately a hundred times smaller than a common digital picture, in terms of disk size. An even more relevant comparison would be with modern video games. They also perform similar real-time rendering and often exceed 50 GiB in disk space, which makes them approximately a million times larger than the size limit set for this project.

My motivation for this project was a long-term interest in computer-generated 3D animations, especially those rendered in real-time on personal computers available to everyday consumers. My first introduction to the demoscene was approximately 15 years ago, when I watched a production called Second Reality[2], released by Future Crew in 1993. It featured many unusual rendering techniques and a striking soundtrack by Purple Motion. Another demo that deepened my interest in computer graphics was fr-025: the.popular.demo[3] by Farbrausch, with its striking visuals and a catchy soundtrack. I wanted to learn more about this topic and challenge myself to create an intro at least remotely comparable to some of these old classics.

The technologies used or considered when making this project are described in chapter 2. Chapter 3 outlines the design decisions behind this project. The final implementation, encountered issues, examples of rendered frames and decomposition of the created scene can be found in chapter 4.

[1] 1 KiB = 1024 B, 1 MiB = 1024 KiB, 1 GiB = 1024 MiB
[2] https://www.pouet.net/prod.php?which=63
[3] https://www.pouet.net/prod.php?which=9450

1.1 Demoscene

In the 1970s and 1980s, when the first personal computers and home gaming consoles started appearing, the hardware resources were significantly limited. This lead to a subculture among the gaming community, dedicated to creating the most complex short animations possible in real-time given the constraints of the contemporary gaming hardware available to general consumers[11]. These served as demonstrations of their ability to push these limits and utilize the scarce resources to the maximum, which is where the name of genre, *demo*, originates from. Due to the limitations of the time, it was often necessary to come up with numerous clever tricks to overcome them.

Many of the groups and individuals behind these early productions were involved in various kinds of computer hacking, modifying third-party proprietary software, often with the intent of reverse-engineering and circumventing the licence checks. Software cracking largely relies on in-depth knowledge of low-level programming, which meant that these people were well-equipped for the task of writing very efficient rendering code. This community of real-time animation creators and their fan base became known as demoscene[12].

As personal computers developed over the following decades, the need to write minimalistic, performance-optimized code has been largely replaced by efforts towards long-term software maintainability. It became undesirable to utilize these kinds of hacks, which tend to make code difficult to understand by anyone unfamiliar with them and prone to errors. Both the disk and memory space constrains were no longer as significant limiting factors as they used to be in the early times of personal computers. The code of end-user applications became largely composed of error checks and alternate execution paths to resolve runtime issues. This paradigm shift and the need to make computer programs as user-friendly as possible meant that the size of software grew rapidly. The demoscene was also affected by this and their productions have split into two divergent categories.

The first category remained true to the original idea of utilizing all the available resources, creating increasingly photorealistic and cinematic animations. This category remained known as *demos*. Modern demo productions commonly have a disk size in the order of units, or even tens, of MiB. The other one instead focused on creating minimalistic productions, artificially limited by the final executable file size. These are known as *intros*. Several subcategories have evolved, with different maximum executable disk sizes, typically set to powers of two in bytes. Some of the most commonly used limits are 256 B, 1 KiB, 4 KiB, 64 KiB. In order to minimize the size, it is still very common to use various, often platform-specific tricks in intros, especially in the smallest ones.

Most demos and intros larger than 1 KiB include audio alongside the animation. In the case of large demos, it may be in the form of a common audio track in MP3, WAV, AAC, Ogg or any other standardized format. When disk size is an important factor, the soundtrack is typically synthesized at runtime, either from MIDI instructions stored in the binary or even completely procedurally generated. This used to be the case even for demos unrestricted by size in the early times of the genre, which then required a substantially powerful sound card in order to play the demo with sound.

One of the most recognizable groups in the demoscene community is Farbrausch. It has been active for two decades and released well over a hundred productions in a wide range of categories, with the majority being either regular demos or 64 KiB intros. Five out of the current top ten demoscene productions of all time[4] were released by this group.

[4]http://www.pouet.net/prodlist.php?order=views

Demoscene productions are traditionally presented at community gatherings called *parties*[5]. Historically, some of the most significant parties used to be The Party[6] hosted in Denmark from 1991 to 2002 and Breakpoint[7] hosted in Germany between the years 2003 and 2010. Other recognizable parties include Assembly, The Gathering and Revision, all of which are hosted in Europe. They are typically held at an annual basis and referred to by their name followed the year. Productions presented at parties are usually ranked within their category based on the number of votes received from attendees.

One of the largest online demoscene communities gathers around a website called Pouët[8]. It hosts over 85 000 productions and a user forum. There are many other websites with useful resources on all important demoscene topics. A categorized list of these case be found at demoscene.info.

[5]https://www.demoparty.net
[6]https://en.wikipedia.org/wiki/The_Party_(demoparty)
[7]https://en.wikipedia.org/wiki/Breakpoint_(demoparty)
[8]http://www.pouet.net

Chapter 2

Technologies

This chapter describes algorithms and approaches related to the creation of real-time rendering software focused on minimal disk size.

2.1 OpenGL shaders

This section does not describe the entire OpenGL rendering pipeline in detail[1]. Instead, it only focuses on the stages and aspects relevant to this project. There are several types of programs executed on the graphics card both during different stages of the rendering process and independently of it[6]. They are called shaders because they are commonly used to control the lighting and shading of objects in a rendered scene. Shaders are written in a special programming language called OpenGL Shading Language (GLSL) and typically compiled in the initialization stage of program runtime.

In modern OpenGL, the rendering pipeline typically uses a shader program composed of at least two shaders, a vertex shader and a fragment shader. There are also other types of shaders which can be used to apply additional effects in other stages of the rendering pipeline when necessary. A simplified diagram of the OpenGL pipeline can be seen in figure 2.1. Each type of object in the scene can have its own set of shaders compiled into a shader program, which can then be applied during its rendering. It is even possible to draw the same object multiple times using different shader programs, resulting in distinct renderings (e.g., using different shading algorithms or performing different transformations).

Vertex shaders, as the name suggests, are called for each vertex that was sent to the rendering pipeline, including those that never end up being rendered due to being outside the field of view or obscured by another object. There may be some exceptions to this, but this is one of the best ways to imagine how it works. It takes whatever data are available about the vertex, such as its location, texture UV coordinates and other relevant properties, as its input. The primary output are screen coordinates, typically obtained by multiplying the world position by various transformation matrices and the perspective projection matrix, but many other properties can be passed from the vertex shader to the remainder of the rendering pipeline.

Texture coordinates are often passed through the vertex buffer without any modification. The location coordinates of a vertex can also be transformed nearly arbitrarily, allowing the scene to be completely modified in terms of its shape even after the rendering process has already begun. Thanks to this property, a large plane composed of tiny triangles can

[1]https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

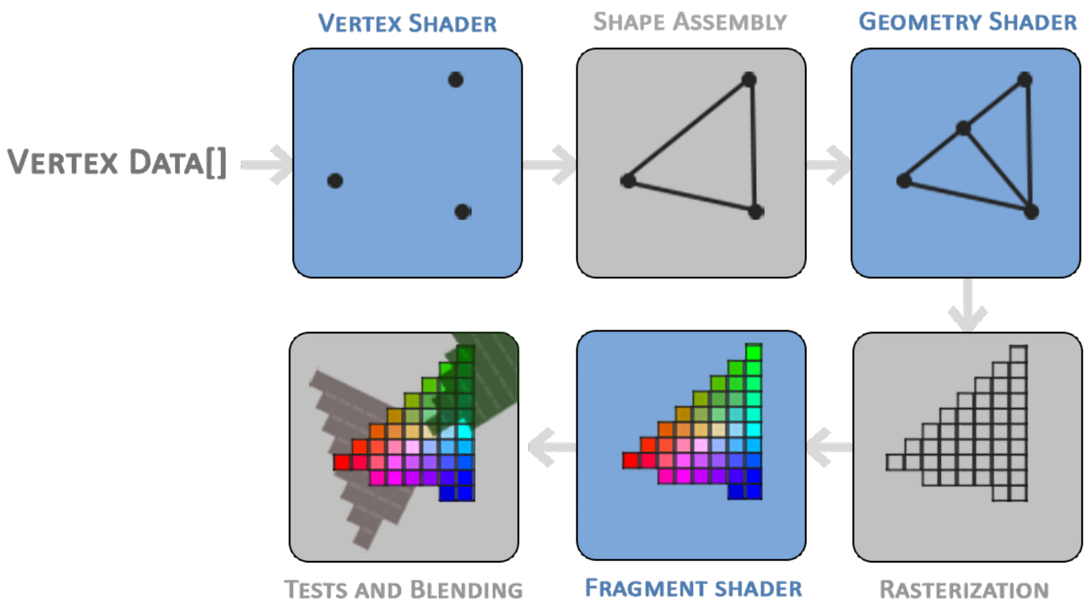


Figure 2.1: Simplified diagram of the OpenGL rendering pipeline[2], with stages programmable via GLSL shaders highlighted by a blue background

be constructed in processor code, written into a vertex buffer, passed to the graphics card and then, when the rendering gets to the vertex stage, a height-map can be applied onto the vertices to give the plane a much more complex shape. The advantage of this approach, when compared to applying the height-map in the processor code, is that shaders are generally executed in parallel and graphics card processors are better optimized for floating-point arithmetic operations, making the execution of such code potentially significantly faster.

Fragment shaders are used in another important stage of the OpenGL rendering pipeline. It is easiest to imagine fragments as pixels on the screen, even though it is not necessarily always true. The transformed coordinates from the vertex shader can be used to determine which polygons are visible and which are obscured or out of sight. Therefore, the fragment shader is only executed for fragments that are likely to be rendered onto the screen (i.e., visible fragments in the foreground). Again, there are exceptions to this, especially when drawing partially transparent objects.

Input to the fragment shader is taken from the output of the vertex shader. However, a triangle composed of three vertices can result in thousands of rendered fragments. Each vertex typically has different output variable values and the majority of fragments are somewhere between the vertices of the polygon they belong to. It is almost always desirable to interpolate the values from the individual vertices, which is the default OpenGL behaviour, but it can be configured differently if needed[3]. This way, each fragment receives input variables corresponding to its position within the polygon and the developers do not have to implement the interpolation themselves.

[2]Image taken from https://learnopengl.com/Getting-started/Hello-Triangle on 2021-05-08
[3]https://www.khronos.org/opengl/wiki/Type_Qualifier_(GLSL)#Interpolation_qualifiers

Fragment shaders are responsible for determining the colours of individual fragments based on the variables passed and interpolated from the relevant invocations of a vertex shader linked to the same shader program. These typically include the location of the fragment in world space, its texture UV coordinates and information about lighting. The shader uses this data to calculate the colour of the fragment. Many kinds of effects, especially those related to lighting, shading and colour filters, can be applied here.

While not directly involved in the rendering process, compute shaders are another useful type of program executed on graphics cards[2]. They are typically used to perform parallel computation tasks, such as generating textures or applying filters to them. Outside the context of visual applications, in areas like machine learning, big data processing or crypto-currency mining, they can be utilized for an even wider range of tasks. Graphics card processors are optimized for floating-point operations and compute shaders have access to all the mathematical GLSL functions, which makes them much faster at processing large quantities of data in parallel than traditional processors.

Compared to running code on the processor, there are some challenges associated with using compute shaders. For example, passing data to and from a compute shader typically requires the allocation of buffers in the graphics card memory and copying data from the processor memory to the graphics card and then back once the computation is finished. Depending on the complexity of the program, degree of parallelization and the size of data, there may be a significant overhead due to the time it takes to move data between the processor and graphics card memory. It is possible to implement other kinds of programs that do not involve any complex computations using compute shaders as well, even if they cannot be parallelized at all. This is rarely more efficient than running equivalent code on the processor, but there are situations where access to the mathematical functions outweighs the performance drawbacks.

2.2 Noise

In computer graphics, noise refers to a pseudo-random N-dimensional sequence of values. These are often used in procedural generation of textures, terrain, clouds and anything else that appears random in nature. To make the noise functions easier to understand and visualize, examples and explanations are presented in the form of two-dimensional textures. This is one of the most common use cases for noise in practice as well. It is typically generated by a function that takes a vector of N-dimensional coordinates as its input and returns a single floating-point value within a certain rage (typically either from -1 to 1 or from 0 to 1, depending on implementation).

The most primitive kind is random noise, also commonly known as white noise. Each sample is randomly generated and unrelated to any previous sample, with a mean value of 0 or 0.5 depending on the implementation. When applied to 2D textures, the value depends on pixel coordinates and a seed. Most implementations attempt to achieve a uniform distribution of generated values. This noise is rarely used on its own because it does not commonly appear in nature.

In computer graphics, a much more natural-looking pattern is desirable. Sudden changes are scarce in nature. The noise function must be continuous, with the returned value changing smoothly based on the change of the input, without any apparent geometric artefacts or repetitive patterns. A small change in the input vector results in an accordingly small, change of the output value. A change greater than a certain, implementation-specific threshold yields a seemingly unpredictable output value unrelated to the previous one. To control this behaviour, it is necessary to choose an appropriate scaling factor for the input coordinates in order to achieve a suitable pattern density. The input scaling factor is called persistence. In order to be usable in real-time rendering, the algorithm must not be computationally expensive. For many purposes, it must yield consistent results when given the same input parameters. For example, it would be unacceptable for the noise pattern to be different with each rendered frame.

One of the easiest ways to create a noise function is to interpolate values in a fixed lattice pattern obtained via common random number generation functions. This is known as value noise. Linear interpolation is both easy to implement and computationally inexpensive. The drawback is a constant rate of change between the randomly generated points. This makes it look very unnatural in many scenarios because a constant rate of change is scarce in nature. Higher-order interpolation generally yields betters results, but without applying other algorithms on top of it, sharp edges between the randomly generated points may remain visible. Expanding this approach into higher dimensions becomes increasingly difficult and the results are degraded. Due to these reasons, using a plain interpolation of any order is not very common when simulating nature in modern computer graphics applications. Instead, more complex methods of generating less unnatural noise have been devised.

Perlin noise is a function for generating continuous pseudo-random noise. The algorithm resembles that of value noise, but instead of generating a single value per lattice point, a gradient vector is generated instead. These vectors are then interpolated to obtain the final value. It is one of the best known and most commonly used noise functions throughout the field of computer graphics, belonging to the category of gradient noises. It was created and published by Ken Perlin in 1985, originally for use in the film industry for creating computer effects[7].

Simplex noise was released by Ken Perlin in 2001 as an improvement upon Perlin noise in order to resolve some of its issues[8]. It has less noticeable directional artefacts thanks to the use of more complex underlying geometrical shapes. The simplex noise function is also less computationally expensive in higher dimensions. However, its higher-dimensional implementations have been patented by Ken Perlin, which limits its usability in commercial software. For this reason, the availability of implementations in majority of popular programming languages and thanks to its relative simplicity and general popularity[4], Perlin noise was never completely replaced by simplex noise and remains commonly used.

Popular implementations of Perlin noise and simplex noise for execution on graphics cards were created by Stefan Gustavson and are currently freely available in his GitHub repository[5] under the MIT licence. He also published an article with his explanations of these noise functions[3], including their additional implementations in Java. In 2012, Ian McEwan et al. from Ashima Research released an article on the topic of Perlin and simplex noise implementation for real-time use in rendering on graphics cards[5].

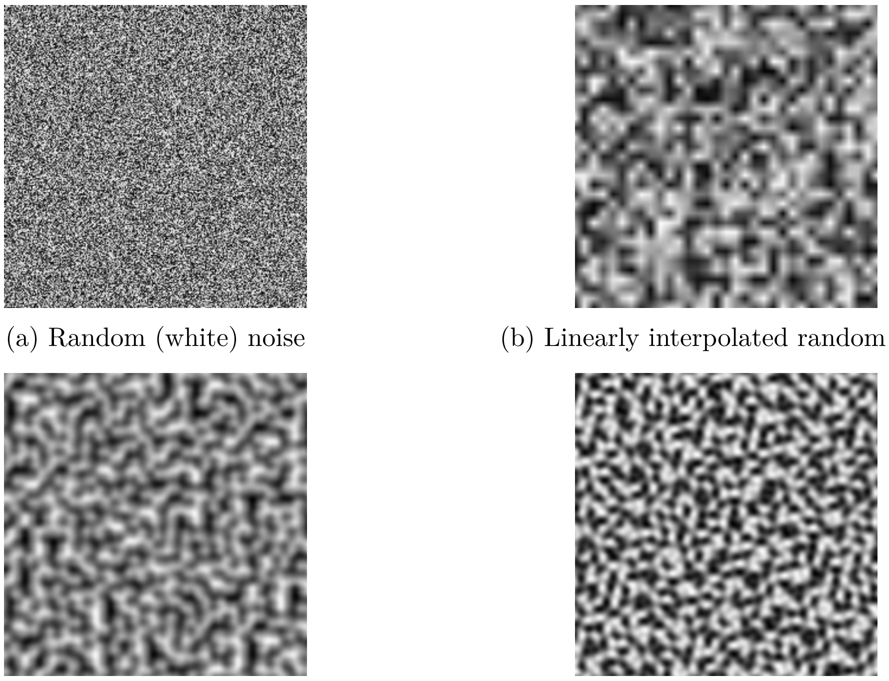
Alternative source of noise function implementations for graphics cards is a GitHub Gist by Patricio Gonzalez Vivo titled GLSL Noise Algorithms[6].

Renderings of the four aforementioned noise functions in the form of 2D textures can be seen in figure 2.2. Random noise could not be scaled to match the other three because it would be indistinguishable from the linearly interpolated random noise.

[4]`https://computergraphics.stackexchange.com/a/26/15488`

[5]`https://github.com/stegu/webgl-noise/`

[6]`https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83`



(a) Random (white) noise (b) Linearly interpolated random noise

(c) Perlin noise (d) Simplex noise

Figure 2.2: Examples of 2D textures generated by some of the most commonly used noise functions in computer graphics, scaled for a better comparison

More complex noise functions can be easily created by combining multiple simpler ones with different input and output scaling factors. This can produce patterns with much finer details without changing the scale of the base noise, which is typically the one with the greatest output scaling factor. An example of this is fractal noise, also known as pink noise. Uniform distribution of generated values is typically not guaranteed for any of the more complex functions. Higher dimensional gradient noise functions tend to produce more values closer to the mean value[7].

2.3 L-systems

A Lindenmayer system, commonly referred to as an L-system, is a formal grammar, consisting of an alphabet, an axiom and a set of substitution rules[9][10]. It can be used to describe complex self-similar shapes, such as fractals, with millions of vertices, using just a couple of bytes of disk space. Even very complicated L-systems rarely exceed a hundred bytes when stored efficiently.

The alphabet is a set of characters, which function as instructions for the rendering engine. Most commonly used characters are symbols of the Latin alphabet, arithmetic operators and brackets. In the program, the alphabet appears in the form of instruction processing logic. Each character of the alphabet has its behaviour implemented there. This is used to turn a sequence of L-system instructions into an image that can then either be rendered directly onto the screen or stored in a texture for later applications onto other objects in the scene.

[7]https://computergraphics.stackexchange.com/q/4212/15488

Common instructions include moving forward, turning clockwise by N degrees, turning counter-clockwise by N degrees, savings the current position onto a stack and popping the latest position back from the stack. Movement forward, or in any other direction, implies drawing a line in the context of rendering logic. There are usually between four and eight characters in the alphabets of common L-systems. When correctly combined, they can be used to construct geometrically complex, self-similar shapes, such as fractals.

The axiom is a string of characters containing the initial sequence of instructions before any substitution iterations are performed. The usual length is in the order of units of characters. Sometimes, it is just a single character, typically the instruction to move forward.

The substitution rules are a set of string pairs used for replacing parts of the axiom. When the rules are applied to the axiom, a new string is produced. This is the first iteration. For the majority of L-systems, the rules can be applied repeatedly any number of times, gradually increasing the length of the string and thus the complexity of the rendered structure. One iteration typically increases the length of the string up to ten times, with the usual rate of growth being between two and three times. Thanks to this, it is usually sufficient to perform only a few iterations in order to obtain enough instructions for a very detailed rendering. Each iteration typically increases the size of the rendered geometry, which means that it needs to be scaled down accordingly, in order to preserve consistent dimensions of the shape, independent of the number of performed iterations.

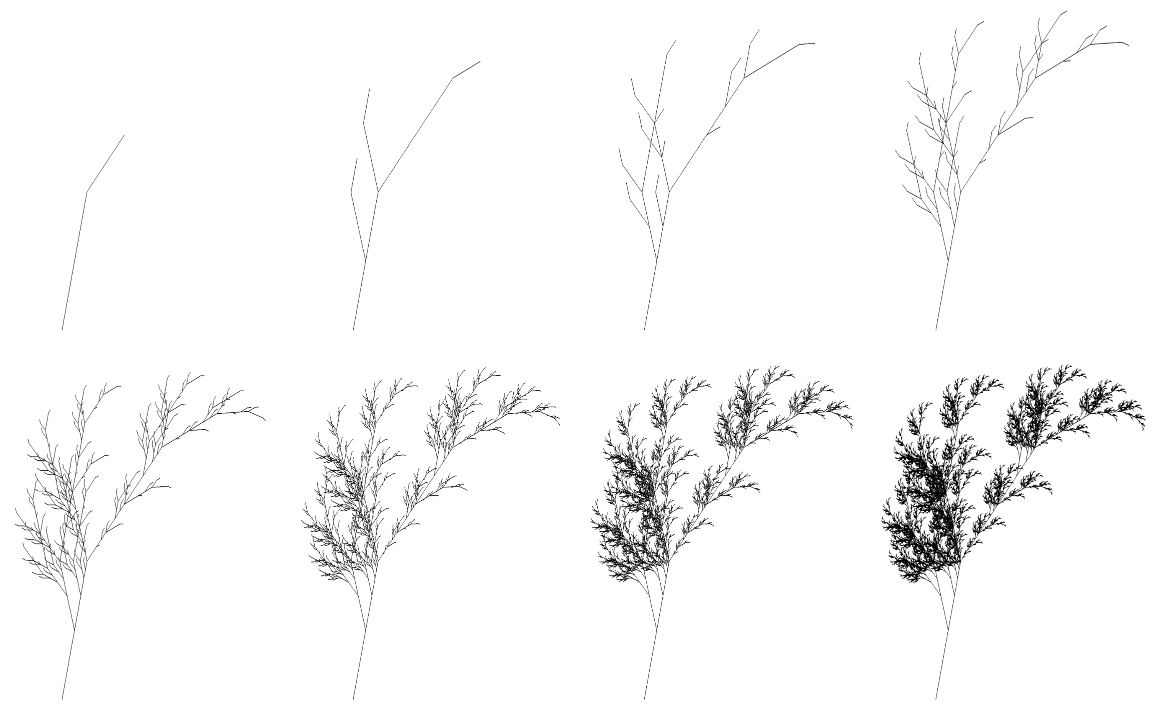


Figure 2.3: First eight iterations of the fractal plant L-system showing the gradual increase in shape complexity with each application of the substitution rules

Implementations of a string substitution and an L-system rendering engine are relatively simple and can be reused for any number of different L-systems. The initial data of each system (an axiom and substitution rules) can usually fit under a hundred bytes. These properties are convenient for software focused on minimal disk size, but not constrained by memory space limits during runtime.

There are many popular fractals and other self-similar shapes that can be described using an L-system definition, such as the Sierpinski triangle, the dragon curve, the Koch curve and the fractal plant[8] visualized in figure 2.3.

2.4 Executable file size minimization

The primary constraining factor in intro development is final executable file disk size. In order to minimize the binary size of intro productions, a special linker focused specifically on achieving this task was developed by Rune L. H. Stubbe and Aske Simon Christensen. It is called Crinkler[9] and is currently available for public use on GitHub[10]. Is it commonly used for 1 KiB and 4 KiB intros, but it can be used for larger productions as well. When used, it attempts to reorder and compress both the executable and the data parts of binary object files in order to create the smallest possible linked executable file.

For intro productions with a higher size limit, such as 64 KiB, different tools are used. A very popular packer created and used by the German demoscene group Farbrausch called *kkrunchy*[11] was used to minimize the disk size of a great number of 64 KiB intros. It is publicly available on their GitHub[12] and remains one of the most commonly used tools for binary minimization many years after its initial release.

Another tool related to executable file minimization is *Sizer*[13], created by Aras Pranckevičius. It can be used to analyse binary files and determine which of their parts are most responsible for their disk size. It has some common code with kkrunchy, but it does not perform any minimization itself. It is also publicly available on GitHub[14].

2.5 Shader code minimization

Unlike code written in C/C++, which can be significantly compressed during compilation and linking before program execution, GLSL (OpenGL Shading Language) code is compiled during application runtime and usually must be included in the executable file in its text form. This means that large, complex shader programs can cause a major increase in disk size, which is undesirable when developing an intro. The impact of shader code on the final binary size can be reduced by removing unessential white-space characters, comments, unused code and shortening variable names. This can be done both manually or automatically using an external tool, and it commonly reduces the size of shader code to approximately 10–20% of the original. Real impact on the final size tends to be smaller, as there are usually also other minimization processes, such as data section compression performed by a linker, included in an intro build pipeline.

[8] https://en.wikipedia.org/wiki/L-system#Example_7:_Fractal_plant
[9] http://code4k.blogspot.com/2010/12/crinkler-secrets-4k-intro-executable.html
[10] https://github.com/runestubbe/Crinkler
[11] http://www.farbrausch.de/~fg/kkrunchy/
[12] https://github.com/farbrausch/fr_public/tree/master/kkrunchy
[13] http://aras-p.info/projSizer.html
[14] https://github.com/aras-p/sizer

Chapter 3

Design

This chapter describes the individual design decisions of this project and the reasoning that motivated them.

3.1 Target platform

Microsoft Windows running on x86 processors are the platform of choice for the majority of intros. Historically, video games existed on MS-DOS since the 1980s and Windows remain one of the most popular gaming platforms to this day. Thanks to this, they have a great support for the latest computer graphics features built directly into the operating system. Many Linux distributions may not include graphics card drivers and OpenGL libraries, which makes developing intros without focusing on a very specific distribution complicated. There are many commonly used desktop and window managers on Linux, which further increases the difficulty of creating a universally compatible intro without relying on libraries that may not be present in many installations.

All modern desktop installations of Windows include a relatively recent version of OpenGL libraries and basic drivers for all major graphics cards. Having a guaranteed access to all important OpenGL functions is very convenient when linking an additional library to the intro binary would have been a significant inconvenience. Windows also provide a consistent API for creating and managing windows and their rendering contexts.

Among other popular platforms are retro gaming consoles, such as Commodore 64 and various products from Atari and Amiga. However, in order for anyone to be able to run such demo productions, they must either own the console or its emulator. Compatibility and ease of evaluation are important factors for this project, so using an old platform would be inconvenient. There are fewer resources available about them and it would be also practically impossible to utilize modern technologies, such as compute shaders, which was one of the goals of this thesis.

In conclusion, development of intros for Microsoft Windows is typically more straightforward and results in smaller binaries with better compatibility than most of its alternatives. Ample resources are readily available about OpenGL intro development for Windows on the Internet thanks to its long-lasting popularity on personal computers.

3.2 Choice of rendered scene

From the beginning, the goals of this thesis included utilization of procedural generation techniques, such as noise functions and L-systems. A time-lapse flight through a natural environment allows a convenient combination of both into a single scene. The intended template used by the intro was released by a co-author of *elevated*[1], one of the most popular intros of all time, featuring mountains and a large body of water. This intro served as an initial inspiration, but the theme has slightly shifted since then.

The final design of the scene is an imitation of a natural environment, resembling a mountainous swamp. The main features are mountains, numerous bodies of water and a large quantity of plants.

The intro must be capable of running at a stable frame-rate on common graphics cards in personal computers. This limits the number of vertices that can be rendered with each frame and the complexity of shader programs. Making too many noise function calls prolongs the execution and decreases the frame-rate. An endless open-world terrain covered in small-sized vegetation requires a lot of vertices to be constructed with a sufficient degree of detail. To make it possible to run the intro for a long period of time, objects that are no longer visible must be seamlessly recycled back into the scene. They could also be destroyed and replaced by new objects instead of being reused. However, the movement of all objects in the scene is implemented in the relevant vertex shader code, so it is easier to keep using the same objects and moving them back into the visible part of the scene.

3.3 Terrain generation

Terrain is based on a single seemingly infinite height-map, spanning across the entire scene. The height is computed using a noise function with suitable properties in a vertex shader, to imitate shapes of terrain commonly found in nature. The majority of terrain is covered in grass, with high grounds on mountains having a more stone-like appearance. The generated terrain must allow a seamless transition between dry, grassy land and bodies of water without sudden sharp edges.

The underlying structure, upon which the terrain is rendered, is a large trapezoid, narrow near the camera and widening with increased distance. The terrain must fill the entire width of the screen, even at its maximum rendered distance from the camera. This is why a trapezoid shape is more suitable than a regular rectangle. In order to apply the height-map to Y coordinates of the base shape of terrain, this trapezoid is divided into a sufficiently large number of triangles. It is necessary to find a compromise between smoothness of terrain surface and the frame-rate drop caused by rendering large quantities of polygons.

3.4 Vegetation placement

To make the scene appear livelier and less flat, plants are added onto the terrain. To make the implementation easier, they can be uniformly distributed throughout the scene, regardless of the type of underlying terrain. The fractal plant L-system is used for the vegetation. Due to the fact that it is only two-dimensional and resembles simple small

[1]https://www.pouet.net/prod.php?which=52938

plants more than tall trees, it is necessary to render many thousands of instances to achieve a somewhat natural appearance.

First, several substitution iterations must be performed to generate enough instructions for a sufficiently complex structure. These instructions must then be converted into lines. This requires trigonometric functions, which are not available from the processor code due to the missing mathematics library. It would be possible to add their implementation, but in order to utilize more features of modern OpenGL, the instruction processing is implemented via a compute shader instead.

While it would be theoretically possible to place the lines directly into the scene, it would have a vast negative impact on performance. Depending on the number of iterations, each plant may be composed of tens of thousands of lines. The scene is populated with thousands of these plants and rendering tens of millions of lines with each frame would result in a very low frame-rate, assuming a common consumer-grade graphics card.

Instead, the plant is only rendered once onto a texture during the initialization. This texture is then be applied onto billboards thousands of times throughout the scene. The positions of billboards are randomly generated within a pre-defined area in the processor code. Their size and aspect ratio are also randomized within suitable limits. The billboards must move in sync with the height-map on top of terrain. Once a plant gets behind the camera, it is moved to the back of the plant placement area. This happens repeatedly, resulting in a seemingly infinite number plants in the scene. Each billboard is composed of only two triangles, which are much faster to draw than the individual lines.

Having thousands of exactly equal plants in the scene would look very unnatural, so it is necessary to add noise to the billboard vertex locations and the texture colour. In nature, plants are often moved by wind, so the noise changes not only based on coordinates but also with the passage of time. This is where a 4D noise function becomes very convenient.

3.5 Sky rendering

A common practice for rendering a sky is to use a sky-box texture built into the executable file and rendered onto the inner faces of a cube around the scene. This technique is easy to implement and offers a relatively natural-looking appearance. There are, however, significant drawbacks as well. Unless the texture is constructed correctly, the edges of the sky-box cube may become visible, which breaks the illusion of a distant sky. More importantly, a static texture increases the executable file size and it cannot change depending on time. These properties make such sky-boxes unsuitable for an intro that renders an endless time-lapse with a permanently visible sky.

The sky patterns are rendered using a combination of noise functions with different persistence levels in a fragment shader. Since the rendering is implemented on a per-pixel basis, similarly to ray-tracing, the shape of the object onto which the sky is rendered can be arbitrary. One of the simplest solutions is to place a large rectangle or trapezoid into the distance, to cover the entire screen above and behind the terrain. The world-space location of each rendered fragment, relative to the camera, is normalized and then scaled up to get a point on a hollow sphere. This way, the sky can be rendered on a spherical surface without having to actually construct one out of many polygons in the processor code, which would be challenging without the mathematical library and its implementations of trigonometric functions.

The constant distance of a sky surface from the camera allows the sky to relatively easily revolve around the scene, which imitates the planetary rotation similarly to the geocentric model of the universe. This is necessary for a more natural sense of time passing at a fast pace. In the real world, individual stars and clouds can be very differently distant from the observer. Placing millions of objects uniformly around the scene and transforming them would be too demanding. Replacing them with a single surface and simulating the objects in a fragment shader is more efficient because only visible fragments require computation.

3.6 Water surfaces and reflections

The terrain is intersected by a single flat trapezoid, widening with increasing distance from the camera, similar to the base shape of terrain described in section 3.3. This surface must be at a constant height, near the bottom of the terrain, where the slope is not too steep. The constant height level of the water surface imitates the sea level found on Earth, but it also significantly simplifies the logic required to determine which parts of the terrain are underwater. The primary reason for placing water into the scene is its reflective surface, which relies on a combination of rendering techniques.

One of the simplest ways of implementing reflections on flat surfaces is a two-pass rendering system using the stencil buffer[2]. Placement of visible water surfaces is stored in the stencil buffer[3], without affecting the colour buffer. The scene is first rendered without transformations and then again upside-down, below the water level. By making the surface partially opaque, a shade of blue can be added to the reflections for a more natural appearance. The addition of bodies of water into the scene makes it appear less flat and artificial.

All types of objects rendered above water also appear in the reflections. This requires the objects to be rendered twice, at least in some parts of the screen, which results in a lower frame-rate. Drawing to the stencil buffer to determine which of the rendered fragments are above water and which belong to the reflections also requires the same drawing calls to be made. This means that the addition of reflections has a significant negative impact on the overall performance. However, it is necessary for aesthetic reasons to liven up the scene. Optimizing the order of draw calls and rendering to the colour and depth buffers while creating the stencil can help with minimizing this impact, but it is not always practical or even possible.

[2] https://open.gl/depthstencils

[3] https://en.wikibooks.org/wiki/OpenGL_Programming/Stencil_buffer

Chapter 4

Implementation

In this chapter, implementation details specific to this project and complications encountered throughout its development are outlined.

4.1 Intro template

As a base for the code of this project, a template called `i1k_OGLShader`, published by Inigo Quilez on his website[1], was used. This template was originally intended for creating 1 KiB productions based primarily around a single fragment shader rendered across the whole screen. Initial development of this project was focused on ray-tracing performed in a full-screen fragment shader. Over time, it diverged from this path and focused more on the implementation of multiple different algorithms and merging them together. The core of this intro, such as the build configurations and initialization code, is still based on the template, but the majority of the code was written from scratch.

Snippets of code directly taken from or heavily inspired by other sources are preceded by a link to the original source. These include the simplex noise implementation in GLSL by Ian McEwan and Stefan Gustavson[2], LFSR113 random number generator[4][3] and a GLSL implementation of 4D coordinate transformation matrices for rotation around an arbitrary axis[4].

The majority of recent intro productions are created in C/C++, with occasional Assembly code used in the smallest ones. Thanks to the relatively large size limit of this project and the used template, the implementation could be done entirely in C and GLSL. The significant minimization effect achieved by linking the program via Crinkler resulted in plenty of space to work with before the 64 KiB limit would be reached.

4.2 Accessing OpenGL functions

Windows include a recent version of OpenGL libraries, and they are loaded into memory by the operating system. However, by default, even with the correct headers, containing the declarations of all OpenGL functions, most of them cannot be used directly. Intros

[1]https://iquilezles.org/code/isystem1k4k/isystem1k4k.htm
[2]https://github.com/ashima/webgl-noise/blob/master/src/noise4D.glsl
[3]http://www.iro.umontreal.ca/~simardr/rng/lfsr113.c
[4]http://www.neilmendoza.com/glsl-rotation-about-an-arbitrary-axis/

only link a minimal Windows library to their binary, which means that only the most basic functions from an old version of OpenGL can be accessed the usual way.

In order to call any other function, it is necessary to find their memory addresses using a function from the Windows library: `wglGetProcAddress`. It takes the name of a function from one of the built-in libraries as a string and returns its memory address. Given a header with function declarations, it is possible to turn this address into a callable function pointer. Using this method, it is possible to gain access to all modern OpenGL functions by looking them up during the intro initialization, storing their addresses as globally scoped function pointer variables with their respective types. These can be then exposed throughout the program by adding the declarations of the global variables into a header, just like any other user-defined variable or function. This can cause a lot of code duplication and keeping two big chunks of code synchronized is often tedious and prone to errors.

Modern graphics software, such as video games or 3D animation tools, typically use a library that automatically loads all the OpenGL functions without having to manually specify the particular functions which are used. Examples of popular OpenGL function loading libraries include GLEW, GL3W and glad[5].

In this project, all the additional OpenGL functions are listed in a single header file. It is then included in both a global header file with other declarations, which is used throughout the code, and the function-lookup initialization code. The different behaviour of each inclusion is achieved through two preprocessor macros with the same name, each used in their respective case and then undefined again.

4.3 Summary of the rendered scene

This intro renders a time-lapse of a simple scene imitating a natural environment with hills, meadows and bodies of water. It resembles a mountainous swamp with occasional plants scattered throughout the scene. Each day in the animation is $16\pi \approx 50.3$ seconds long in the real world. The sky changes its pattern based on a simulated time of day, with clouds rendered during daytime and a starry sky throughout the night. Water surfaces reflect the scene above them, while remaining partially opaque. The camera seemingly moves straight forward through the terrain at a constant speed. No texture was hard-coded into any part of the intro; everything is generated procedurally at runtime.

Despite the original executable size limit of 64 KiB, a decision was made to minimize the size instead of filling the program with third-party code, textures and other data to enhance the existing functionality, which is a common practice in intro productions. This approach resulted in a final size below 8 KiB, at the cost of slightly less aesthetic scene components.

Figure 4.1 is a side-view diagram of the scene, showing the relationship between the different surfaces. To complement this, figure 4.2 shows a top-down view of the scene, visualizing the shapes and relative sizes of individual components. Decomposition of a single rendered frame into individual components implemented within this project can be seen in figure 4.7.

[5] https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

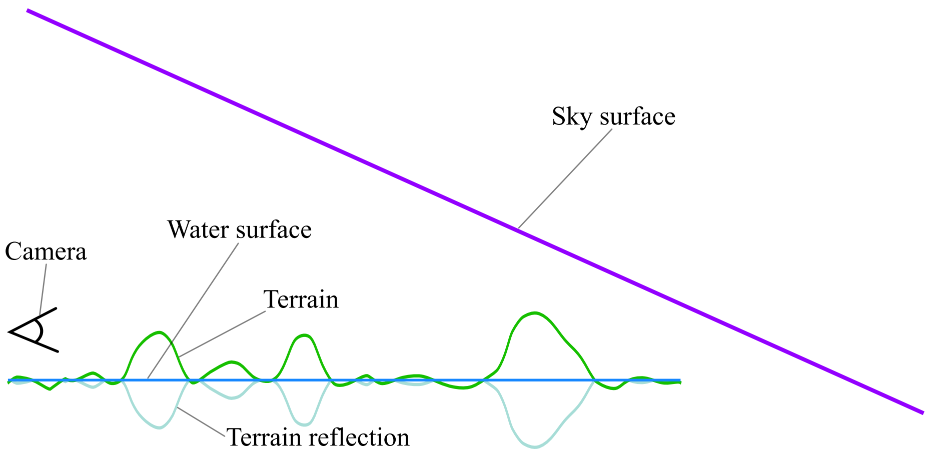


Figure 4.1: Side-view diagram showing a visual approximation of the spatial relationship between the main surfaces of the scene rendered in this intro – terrain, water and sky

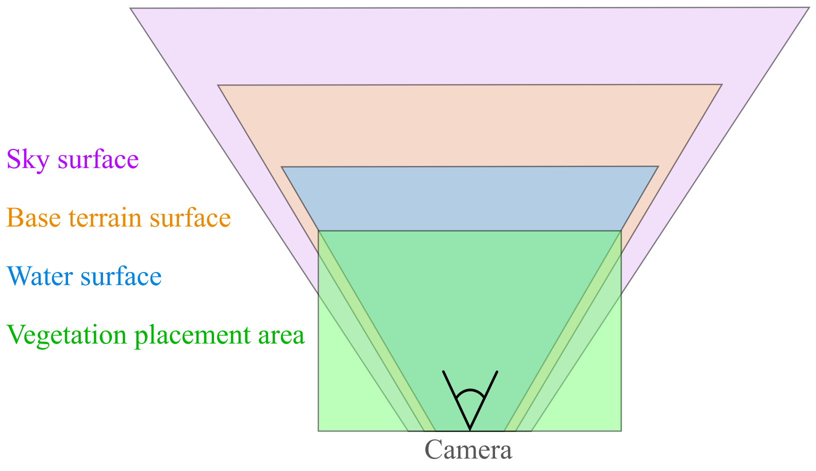


Figure 4.2: Top-down view diagram visualizing the approximate shapes and relative sizes of individual components of the scene and how they overlap

4.4 Terrain rendering

Terrain is procedurally generated using simplex noise[1]. The base shape of the terrain is a flat trapezoid, which widens with distance from the camera, to fill the perspective frustum. This trapezoid is divided into thousands of triangles, in order for the vertex shader to be able to modify its shape. Multiple layers of noise with different levels of persistence are added together to form the height-map, which is applied to the Y coordinates of the underlying triangles.

The terrain is coloured based on its height, with extra noise applied for a more natural appearance. Low areas near water have a blue shade to better blend with water surfaces. The majority of terrain is covered in grass-like shades of green. Higher areas are coloured in dark brown and grey to resemble mud and stones. The colours are more flat, brighter and more saturated than those commonly found in similar environments in nature because real grass and rocky surfaces are composed of millions of tiny objects. Achieving more naturally looking surfaces would be difficult without a significant negative impact on frame-rate or increasing the binary size by using pre-existing textures.

The triangles remain stationary throughout the entire runtime of the intro. Instead, the height-map moves along the Z axis towards the camera at a constant rate. This behaviour allows the intro to run for a very long time without having to modify the base coordinates of the terrain. Given its trapezoid shape, moving it without breaking the illusion of an endless terrain would be more complicated than moving the height-map on top of it.

The normals computed to determine the lighting conditions of a vertex are based on the vertices of triangles artificially constructed around each terrain vertex by applying constant coordinate offsets. One of the greatest drawbacks of using the approach of a moving height-map on a static triangle mesh is related to these offsets. If they are too great or too small, lighting glitches become visible on the ground, especially in areas with a steep slope near the camera. These are mainly caused by floating-point value rounding errors[6], which become increasingly apparent with a long runtime of the intro.

The longer the intro runs, the farther the terrain height-map moves along the Z axis. Adding small offsets to large coordinate values eventually results in at least two vertices of the triangles used for computing normals having exactly equal coordinates. When this happens, the normal vector calculation yields invalid results. A similar unintentional behaviour of a procedural terrain generator was well-known in early versions of Minecraft as *far lands*[7], named by the fact that the effect only appears with very high coordinate values.

Despite numerous attempts to tweak various constants throughout the program, some directional artefacts remain visible, especially on low ground and when the plants align. Attempts were made to minimize them by modifying the noise function calls, but they can be still observed when paying attention to uncanny repetitive patterns. The higher the density of objects, the more apparent these patterns become. Lowering the number of plants in the scene and removing layers of noise with the lowest persistence from the terrain both greatly reduce the number of undesirable artefacts. However, doing that has the downside of making the terrain too flat and smooth and the plants too scarce.

4.5 Vegetation rendering

Addition of vegetation into the scene relies on a combination of many technologies. The plants are rendered by performing L-system substitution iterations on the processor and subsequently processing the L-system instructions into line coordinates in a compute shader. The lines are rendered onto a texture, which requires the use of a custom frame buffer with the texture bound to it. The placement of plants is performed using a conventional pseudo-random number generator on the processor. However, their vertical position depends on the height of terrain, which is generated using simplex noise in the terrain vertex shader.

To imitate a natural scene, many thousands of relatively small plants placed directly on top of the terrain are necessary. Since the scene resembles a swamp, it is not a problem for plants to appear within the shallow bodies of water.

After eight substitution iterations of the fractal plant L-system, each instance is composed of 60 074 lines. The scene contains tens of thousands of plants, which would require over a hundred million lines to be rendered. If rendered directly into the scene, the frame-

[6]https://floating-point-gui.de/errors/rounding/

[7]https://minecraft.fandom.com/wiki/Java_Edition_Far_Lands/Infdev_20100327_to_Beta_1.7.3

rate on a common graphics card would likely be in the units of frames per second. That would be unacceptable for an intro.

Due to the quantity of plants required for a more natural appearance, it is necessary to render them as textures on billboards placed into the scene. This significantly limits the possibility of applying transformations to the individual plants. The texture is generated during initialization and remains unmodified throughout the runtime of the intro.

The X and Z coordinates of billboards, onto which the fractal plant texture is applied, are pseudo-randomly generated in the processor code using the LFSR113 random number generator[4]. The Y coordinates are determined in the vertex shader based on the height of terrain below the plants.

A waving effect is applied onto the billboards in their vertex shader using simplex noise. The texture is also modified by applying noise to fragment brightness levels and giving the bottom part a brown shade, while keeping the top green. This makes them seem slightly less artificial because plants in nature tend to be more brown near the bottom and green near the top.

4.6 Sky rendering

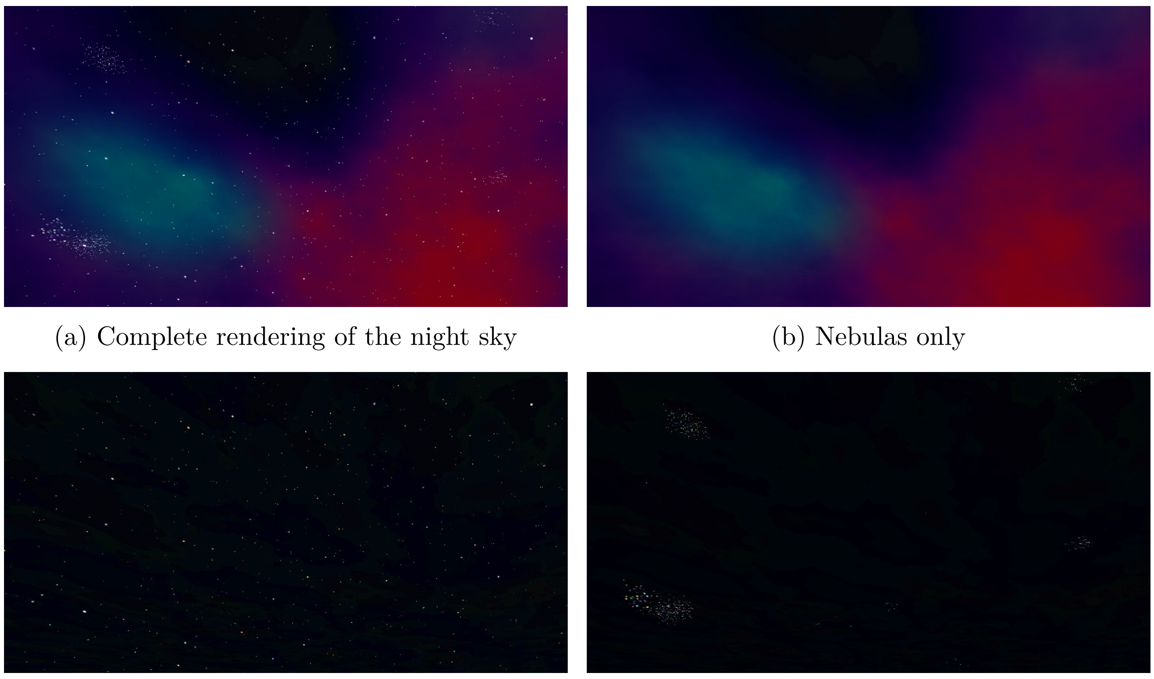
The sky is rendered as a hollow sphere around the entire scene. However, due to the high memory and computational cost of rendering a sphere approximation with thousands of vertices, a simplification was implemented instead. A large trapezoid plane is rendered high above the terrain, with the distant side much longer and lower than the near side, thus covering the entire screen above and behind the terrain.

In the fragment shader, each coordinate vector is normalized and then scaled to find an intersection with the surface of a hollow sphere enveloping the scene. This way, the sky can be rendered as if it were on the inner surface of a sphere without having to generate the actual vertices of such sphere in the code executed on the processor. This helps to improve performance and memory consumption, while making the spherical shape appear smoother than it would have if it were composed of thousands of small triangles.

The sky slowly revolves around the terrain, imitating the passage of time and airflow. The daytime sky features procedurally generated clouds on a light blue background, resembling the sky on a sunny day with occasional thin clouds. Multiple simplex noise calls with varying degrees of persistence and output scaling are added together in order to determine the mask, opacity and colour of the cloud-like patterns. These are, unlike in the real world, rendered at the same distance as stars on a flat surface, which means they have no real volume in the scene. This makes the rendering significantly simpler. The entire sky is described using a single mathematical expression.

The night sky is rendered onto the same surface, but using separate simplex noise function calls. There are three main components layered on top of each other to form the night sky patterns. The most visible are large cloud-like formations of noise imitating nebulas, with colours ranging from cyan to orange. These serve as a background for stars rendered individually and also clusters resembling galaxies. Stars are also generated using simplex noise, but the persistence used is significantly lower than in the case of large nebulas. The colour of stars is determined by their position in a way that allows practically any RGB value to be generated. Figure 4.3 shows a decomposition of the night sky into its individual parts.

From the perspective of the camera, the sun rises on the left side and sets on the right. It also moves slowly back and forth to achieve various angles of lighting, just like the angle


(a) Complete rendering of the night sky (b) Nebulas only

(c) Individual stars only (d) Star clusters only

Figure 4.3: Decomposition of the night sky rendered by the created intro into separate sky formations generated by applying several simplex noise calls with different persistence values and offsets added to the coordinates

of sunlight changes throughout the year in the real world. This allows the viewer of the intro to see the terrain under more than just one kind of lighting conditions.

4.7 Water rendering

Water is implemented as a single large trapezoid, approximately copying the shape of terrain. However, unlike terrain, it is not divided into smaller polygons. The water surface has a constant height and a semi-transparent blue colour. More accurately, from the point of view of a person watching the intro, the surface is partially reflective. Using the stencil buffer, the scene is first rendered without any additional transformations and then again, transformed upside-down, below the water level. This results in seemingly reflective surfaces, as can be seen in figure 4.4.

However, there are no real reflections computed in this project. The partially transparent water surface allows the objects to be rendered below the terrain, through the water, with their vertices mirrored by the surface. For this approach to work correctly, it is essential that the water is a single flat surface. In case of a more complex shape, the reflected objects could not be transformed as easily as by inverting their Y coordinates. The lighting of reflected objects must be calculated based on their original position, so it was necessary to send the original vertex locations to the fragment shader rather than the reflected.

There were many challenges involved in making this approach work. Among the most problematic was blending, which had to be enabled to add a shade of blue to the water reflections. This also improved the quality of vegetation, but it required the objects to be rendered in a correct order, based on their distance from the camera. Another neces-

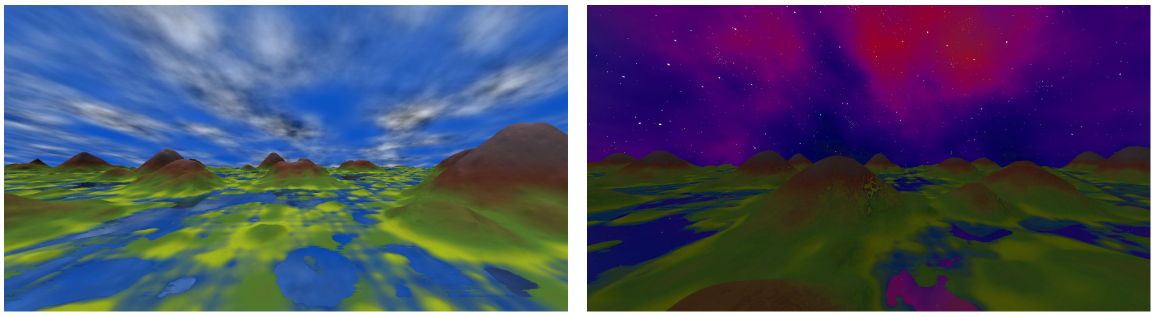

(a) Daytime reflections of clouds and a blue sky (b) Night-time reflections of a starry sky

Figure 4.4: Example frames from the created intro with visible water reflections of the sky and mountains (the vegetation rendering was disabled and image brightness was increased for better visibility of the reflections)

sary modification was the removal of parts of terrain rendered on the wrong side of water surfaces due to the upside-down transformation. This revealed further issues with time synchronization between types of objects, which had to be fixed. Using the stencil buffer required a lot of experimenting before the desired results were achieved.

In the final implementation, noise is applied to colouring of the water surface to make it appear less flat. This should result in slightly more natural looking reflections. Unfortunately, this effect sometimes looks more like a rendering glitch, especially in static frames. This can be observed in the daytime part of figure 4.4 as thin, barely visible horizontal lines around the borders of water surfaces. With plants added into the scene, the negative effect become much less apparent and the more natural impression of a constantly moving water surface remains.

4.8 Limitations of this intro

Due to a lack of proper Windows event handling in the minimized version of the demo, the message queue fills up with each action performed by the user and none of them are resolved. This is not a problem in the case simple mouse movement, but clicking, pressing a key on the keyboard, switching windows and many other events can cause the intro to freeze because Windows will consider it as not responding and it will eventually crash due to this. If left untouched, the program should continue running without any such issues.

Typical demoscene productions begin with a loading screen and an introduction of its authors and end with credits with the names of all contributors. To save as much binary size as possible, given the previously mentioned constraints, and avoid having to implement text rendering just for the short introduction and credits, these parts have been omitted and focus was put onto the actual scene.

For a similar reason, this intro has no soundtrack or audio effects. Addition of sound would require a third-party library and experience with music composition. It would increase the executable file size without utilizing another relevant computer graphics technology, which would go against the goals of this thesis.

Intro code is mostly focused on minimalism, which means that debugging can often become particularly challenging. It is not uncommon to omit any unessential error checks from the code and techniques used in regular software, such as debug printing, are unavailable due to the missing standard libraries.

It is possible to keep two versions of the code, one for debugging and one properly minimized, for example using conditional preprocessor directives. In practice, this often leads to issues that only appear in the minimized version, with the debugging version working as intended, resulting in a necessity to frequently check that the two versions behave the same. There are other areas of intro development where printing is impossible and conventional debugging using breakpoints can be very difficult. For example, fragment shader code can present debugging data through the output colour of pixels, which can either be observed in real-time or their exact values can be examined by taking a screenshot and investigating the RGB values of each pixel.

4.9 Lack of standard libraries

In order to reduce the executable file size of intro productions, it is a common practice to link only the truly essential libraries required for the rendering. This results in several drawbacks that are very unusual when developing desktop applications. In the case of this project, both the standard C library and the mathematical library are omitted, which severely complicates writing more complex code.

Without the standard library, dynamic memory allocation at runtime in the typically used processor memory space cannot be performed. This is due to the lack of functions like `malloc` and `calloc`. In many cases, it is possible to substitute this functionality with static memory, but this can result in a larger binary size, especially if the memory space is initialized to non-zero values. A common workaround is the use of global arrays in place of dynamic allocation. They must be large enough to fit their longest possible contents, usually generated during the initialization of the program or throughout its runtime.

Without the mathematical library, it is impossible to access the majority of advanced operations and trigonometric functions, which are often required by common computer graphics algorithms.

In order to overcome these limitations without having to either increase the binary size by linking the standard libraries or by reimplementing their functionality, it is necessary to find unconventional alternative solutions to many common problems. For more complex mathematical computations, it is possible to utilize OpenGL compute shaders. The graphics card can perform all common mathematical operations. In many cases, implementing the calculations in a compute shader and executing it on the graphics card can be significantly easier than doing so in a code executed on the primary processor. However, there are also additional challenges related to the use of compute shaders.

4.10 Graphics card memory access

One of the most complicated tasks when running code in a compute shader is the mapping between the memory space of the processor and the graphics card memory. There are various approaches and numerous configuration options, which affect the behaviour of the established mapping. If performance is not a significant concern, it is possible to create a mapping that allows the code executed on the processor to access data stored in the memory of the graphics card for both reading and writing. That way, the processor code can work with this memory very similarly to how it would work with dynamically allocated space in the regular processor memory.

Mapping is just one approach of accessing data stored on the graphics card. Another common way of working with such data, particularly in cases where the data do not need to be accessed too frequently (once per rendered frame or only during initialization), is copying them from the processor memory to the graphics card and vice versa. This only involves a single OpenGL function call in many scenarios and it usually causes a lower performance loss than persistent mapping. For example, vertex data are typically copied into the graphics card memory this way. However, in situations where it is not possible to allocate sufficient memory space on the processor side, this solution cannot be used to copy data from the graphics card.

It is also possible to generate the data entirely on the graphics card via a compute shader. The output data can then be passed to other parts of the rendering pipeline without ever being in the processor memory, or they can be accessed from the processor code by creating a mapping for reading.

4.11 Dynamic memory allocation on a graphics card

Crinkler[8], the linker used to minimize the final binary size, tends to crash when attempting to process large chucks of statically allocated memory, even if they remain uninitialized. The data section minimization repeatedly failed when attempting to handle source code with an approximately 100 MiB global array. Even though this particular instance may be slightly extreme, it is not unusual for an intro to need to store data about millions of vertices. Therefore, it is necessary to be able to allocate large memory spaces somehow, ideally dynamically at runtime.

An unconventional approach that solves this issue without much additional code is allocation of the necessary memory space at runtime on the graphics card. OpenGL allows this by making a few function calls with the right combination of parameters. In this project, the allocation of a large memory space on the graphics card was used to store the final sequence of L-system instructions after performing a certain number of iterations. The initial axiom and a list of substitution rules are stored statically in the read-only part of the executable file, but the sequence of instructions and the vertices generated from them are both stored in a memory space allocated on the graphics card.

Performance was not a key factor here because this task is only performed once during the initialization of the intro. The executable file size growth caused by this solution was in the order of units of bytes, it did not require linking the standard C library and it made the linking and compression process performed by Crinkler significantly faster and more stable. While this kind of approach would be considered obscure in any kind of standard production software, in the case of making an intro, it was a viable workaround of the unavailable regular dynamic allocation in processor memory space.

4.12 Requirements for compilation and execution

Compilation of the project requires an updated desktop installation of Microsoft Windows and Microsoft Visual Studio. It was originally developed and built using Visual Studio 2019 and it is expected to remain compatible with at least the next major version. Linking was performed using Crinkler version 2.2, which is included with the source code. There should be no other dependency on external tools or libraries.

[8] https://github.com/runestubbe/Crinkler

Later, Visual Studio Code was used for development, building and debugging instead of the regular Visual Studio due to its subjective ease of use and convenience when creating more complex build tasks. An installation of Microsoft Visual Studio or Visual Studio Build Tools with C/C++ desktop application development features enabled is still required. The build tasks used in Visual Studio Code project configuration use the C/C++ compiler from the regular Visual Studio called MSBuild.

Header files containing strings with shader code are included in with the C source code of the intro. If necessary, they can be recreated by running the included minification tool written in the Go language[9]. During development, version 1.14 of the Go compiler was used. This tool was not a part of the original plan for this thesis, and as such, it is not described in greater detail than necessary. Unlike more advanced tools with a similar functionality, it only removes white-space and comments from the GLSL code, performs substitution similar to the `#include` directive in C/C++ code and generates the header files, which are then included into the rendering code written in C. In order to run this tool, the Go compiler must be installed and configured on the host system. Visual Studio Code project configuration includes tasks for building the shader headers. The regular Visual Studio solution configuration does not include logic for rebuilding these headers.

It is necessary to include the shader code in the intro binary and writing GLSL code directly in a string in C code is very inconvenient due to missing syntax highlighting, autocompletion and other helpful text editor features. Initially, Shader Minifier[10] by Ctrl-Alt-Test was considered for this task. It is significantly more efficient at GLSL code minification, but at the time of testing, during an early stage of development of this project, it had limitations, due to which it could not be used without sacrificing multiple modern GLSL features, such as macros created using the `#define` directive. If the GLSL code was written without using any of the unsupported features and Shader Minifier was used instead of the included tool, the final binary size would likely be slightly smaller. The current design relies on data section compression performed by Crinkler instead of minimizing the GLSL code prior to compilation.

Code minimization and the use of modern OpenGL unfortunately cause numerous compatibility issues when attempting to run the resulting executable on older hardware or software. Running this project requires an updated installation of Microsoft Windows, a version of OpenGL libraries no older than 4.3 and hardware capable of supporting modern OpenGL features. It is difficult to establish specific minimal requirements. According to local testing, Nvidia and AMD graphics cards released after the year 2015 should support all necessary features, but exceptions among them are possible. Many older graphics cards are sufficient, as well as graphics chips integrated into recent processors with the x86 architecture. All minimized versions of the intro assume a screen with support for the 1920×1080 resolution. Setting Windows display scale to 100 % is recommended if the intro is not being rendered correctly or if single-coloured borders appear around it.

4.13 Executable versions

The full, high-performance version of the intro, shown in figure 4.5, was successfully tested on computers running the latest major public release of Microsoft Windows 10 and Nvidia GeForce drivers with the following graphics cards: Nvidia GeForce GTX 760, Nvidia

[9] https://golang.org
[10] http://www.ctrl-alt-test.fr/glsl-minifier/

GeForce GTX 960, Nvidia GeForce GTX 1050, Nvidia GeForce GTX 1050 Ti Max-Q, Nvidia GeForce RTX 2070. It was also tested on a few older computers unsuccessfully, most likely due to the `#version 430 core` requirement of the compute shader GLSL code used to process the L-system instructions. While support for GNU/Linux was not originally planned, a brief testing using Wine[11] revealed that installations with up-to-date graphics card drivers may be capable of running even the high-performance version of the intro at a stable frame-rate, assuming that all other conditions are satisfied.

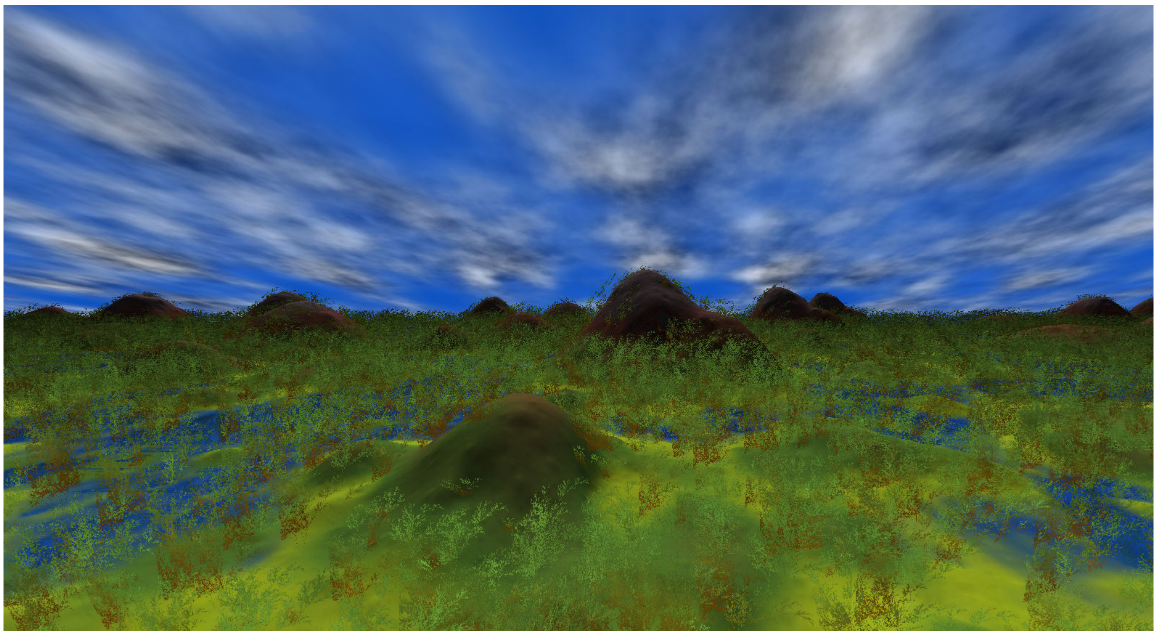


Figure 4.5: Example scene rendered using the high-performance version of the created intro, showing a complex plant shape, a dense vegetation and a larger terrain

Later testing on computers with AMD graphics cards revealed problems with L-system iteration code executed on the processor. The code accesses shader storage buffers[12] allocated on the graphics card. There seem to be significant performance differences between Nvidia and AMD cards when mapping buffers to processor memory space and repeatedly accessing them that way. This issue was discovered near the end of development because it did not affect performance on Nvidia cards.

Changing the memory allocation and access logic at that point would have required a lot of other changes. Instead, a version of the intro with lower requirements for execution was created. This allows it to run on recent discrete AMD graphics cards with a decent frame-rate, as well as higher-performance integrated graphics cards with at least a couple of frames per second. The lower-performance version renders a reduced amount of significantly less complex vegetation with a lower resolution of the intermediate texture placed onto a smaller part of the terrain. To increase the frame-rate, the terrain was also made slightly smaller. The result of these sacrifices is a better compatibility.

The difference between the high-performance and the low-performance version of the created intro can be seen in figures 4.5 and 4.6. They both contain a rendering of the same frame. The lack of mountains in the distance and simplified plants should be clearly visible.

[11] https://www.winehq.org/

[12] https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

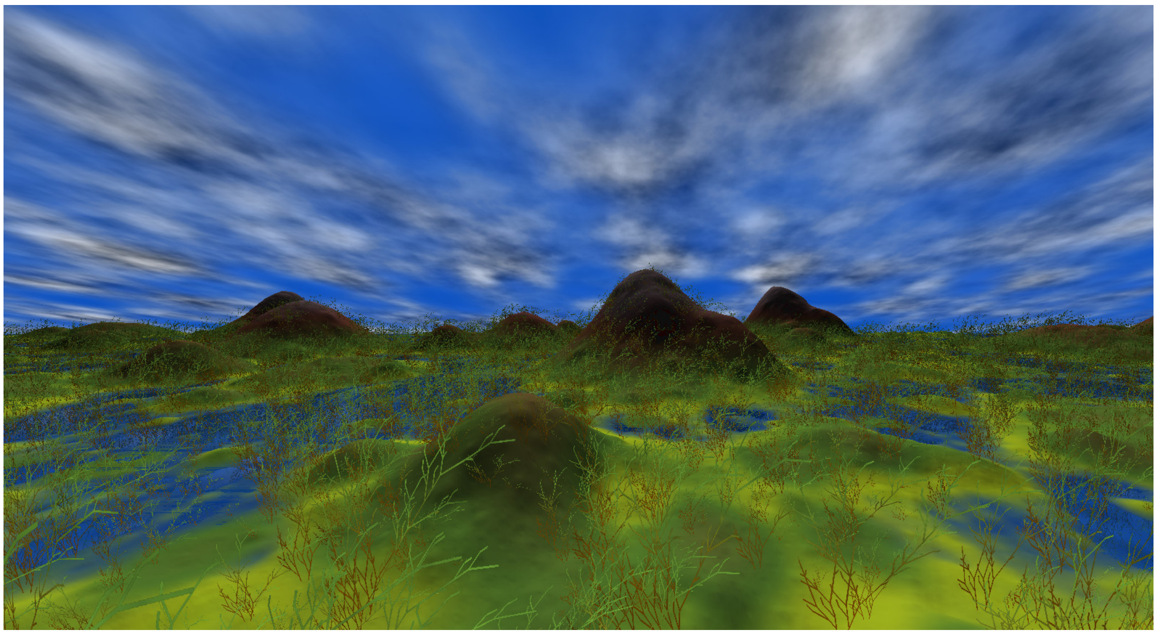
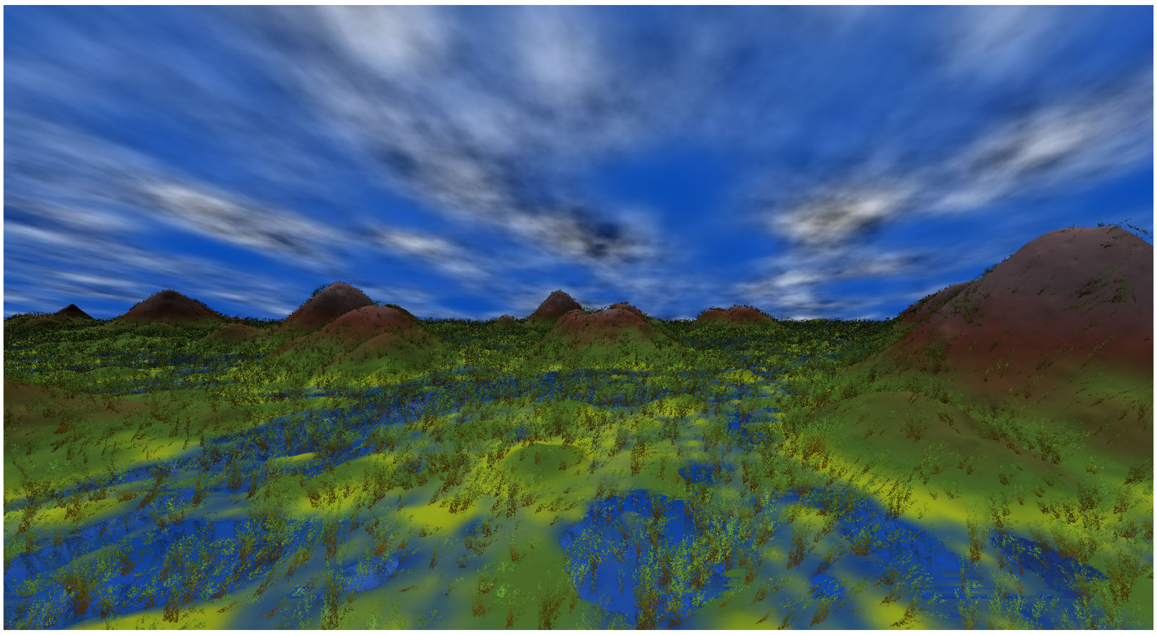
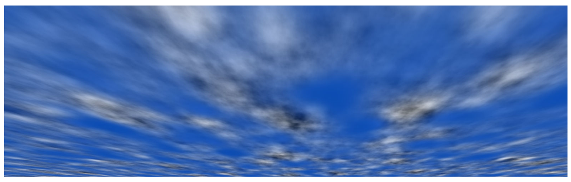


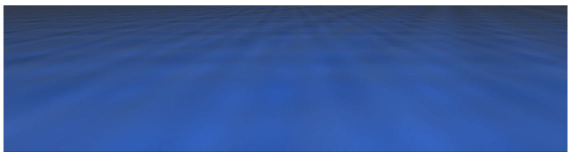
Figure 4.6: Example scene rendered using the low-performance version of the created intro, showing a simpler plant shape, a lower vegetation density and a smaller terrain (similar effect to a shorter render distance, but the distant terrain polygons are completely removed)

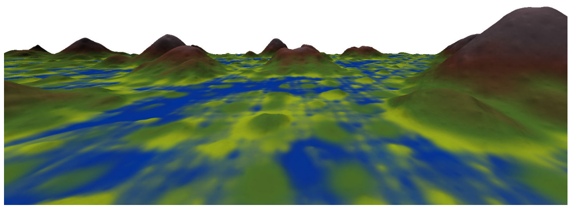
Water surfaces are more apparent in the low-performance version as a side effect of the lower vegetation density. The vegetation is also applied over a smaller area, which causes the lack of plants on distant mountains in the low-performance version. The simplifications may be subjectively perceived as more aesthetic by some people, but the high-performance version shows the scene as it was designed and intended to be viewed.

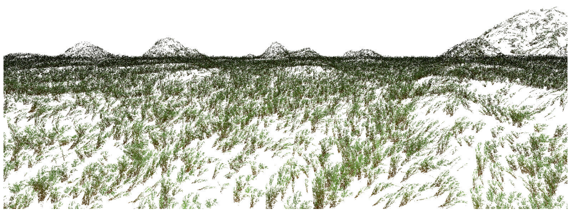
In order to avoid incompatibility with systems that do not allow the aforementioned troublesome memory mapping from the graphics card into the processor memory space, a version without any vegetation was also created. This is not the intended way for the intro to be viewed and it should only be used if all other versions fail to reach the rendering loop in an acceptable amount of time. If this version fails to render a frame in less than a minute of runtime, there is likely a significant problem preventing its execution, such as incompatible GLSL code.

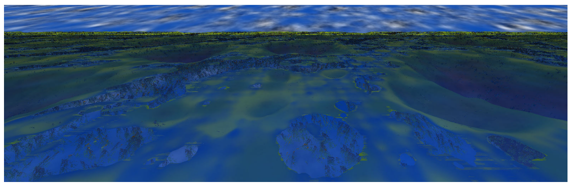

(a) Fully rendered scene including all components and water reflections, as seen during intro runtime


(b) Sky surface only


(c) Water surface only


(d) Terrain only


(e) Vegetation only


(f) Fully reflected upside-down scene

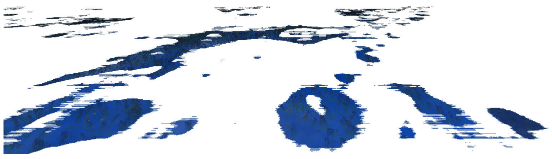

(g) Visible water reflections only

Figure 4.7: Decomposition of a sample frame of the scene rendered by this intro into its individual components and their combined reflections, using the high-performance version

Chapter 5

Conclusion

The goal of this project was to combine several computer graphics technologies into an intro production with an executable disk size no greater than 64 KiB, using modern OpenGL for the rendering. The final executable is less than 8 KiB in size and therefore it comfortably fits into the original 64 KiB limit. The rendering techniques were implemented in accordance with the specification and discussions with the supervisor.

The intro renders a simple scene imitating a natural environment with terrain, vegetation, reflective water and a sky that changes based on a time of day. A simulated day in the animation is $16\pi \approx 50.3$ seconds long. Terrain is procedurally generated at runtime using simplex noise. Vegetation is based on the fractal plant L-system, with substitution iterations performed on the processor and the conversion of instructions into lines in an OpenGL compute shader on the graphics card. The lines are rendered into a texture, which is placed onto thousands of variously transformed billboards on top of the terrain. The sky is cloudy during daytime and starry at night, with the respective patterns projected onto a hollow sphere around the terrain rather than a cubic sky-box. Reflective water surfaces are rendered using one large, partially opaque trapezoid. This required the use of a stencil buffer, alpha-blending and rendering the scene again upside-down. Discoveries made throughout the development were described in this thesis.

Program execution requires a modern graphics card and up-to-date installations of Microsoft Windows, OpenGL libraries and graphics card drivers. Assuming these conditions are satisfied, the intro can be run at 1920×1080 resolution with a stable frame-rate of more than 30 frames per second for many hours before the floating-point rounding errors significantly degrade the quality of rendered objects. At least theoretically, each frame should be unique within the runtime of the intro.

Follow-up work could include an implementation of other L-systems for a vegetation variety, an improvement of the terrain generation algorithm to make it more natural and an addition of clouds to the night sky. Performance optimizations of the existing code could help with decreasing hardware requirements. GLSL code minimizer could also support shortening of variable names and removing dead code, which would further lower executable file size. To improve the aesthetics, the camera could move along a more complex path than a straight line. Given access to a wider variety of hardware configurations and an improved debugging version of the intro, compatibility issues with currently unsupported hardware could be investigated and, in some cases, resolved.

Bibliography

[1] Archer, T. Procedurally generating terrain. In: *44th annual midwest instruction and computing symposium, Duluth.* 2011, p. 378–393. Available at: http://micsymposium.org/mics_2011_proceedings/mics2011_submission_30.pdf.

[2] Bailey, M. OpenGL Compute Shaders. *Oregon State University, http://web.engr.oregonstate.edu/m̃jb/cs557/Handouts/compute.shader.1pp.pdf.* 2016. Available at: https://media.siggraph.org//education/conference/S2012_Materials/ComputeShader_1pp.pdf.

[3] Gustavson, S. *Simplex noise demystified.* Linköping University, March 2005. Available at: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[4] L'Ecuyer, P. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation.* january 1999, vol. 68, p. 261–269. DOI: 10.1090/S0025-5718-99-01039-X. Available at: https://www.ams.org/journals/mcom/1999-68-225/S0025-5718-99-01039-X/S0025-5718-99-01039-X.pdf.

[5] McEwan, I., Sheets, D., Richardson, M. and Gustavson, S. Efficient Computational Noise in GLSL. *Journal of Graphics Tools.* Taylor & Francis. 2012, vol. 16, no. 2, p. 85–94. DOI: 10.1080/2151237X.2012.649621. Available at: https://doi.org/10.1080/2151237X.2012.649621.

[6] McReynolds, T. and Blythe, D. *Advanced graphics programming using OpenGL.* Elsevier, 2005. Available at: http://www.r-5.org/files/books/computers/algo-list/realtime-3d/Tom_McReynolds-Advanced_Graphics_Programming-EN.pdf.

[7] Perlin, K. An Image Synthesizer. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. july 1985, vol. 19, no. 3, p. 287–296. DOI: 10.1145/325165.325247. ISSN 0097-8930. Available at: https://doi.org/10.1145/325165.325247.

[8] Perlin, K. Improving Noise. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. july 2002, vol. 21, no. 3, p. 681–682. DOI: 10.1145/566654.566636. ISSN 0730-0301. Available at: https://doi.org/10.1145/566654.566636.

[9] Prusinkiewicz, P., Hammel, M., Hanan, J. and Měch, R. Visual models of plant development. In: *Handbook of formal languages.* Springer, 1997, p. 535–597.

[10] Prusinkiewicz, P. and Hanan, J. *Lindenmayer systems, fractals, and plants.* Springer Science & Business Media, 2013.

[11] Reunanen, M. Four Kilobyte Art. 2014. Available at: http://widerscreen.fi/assets/reunanen2-wider-1-2-2014.pdf.

[12] Tasajärvi, L. *Demoscene: the art of real-time.* Helsinki: Even Lake Studios Katastro. Fi, 2004. ISBN 978-9529170227.