FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Engine pro renderování a procedurální generování voxelových světů
Student:	Bc. Lukáš Hepner
Vedoucí:	Ing. Adam Vesecký
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 28. května 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Pokyny pro vypracování

Cílem práce je analýza, návrh a implementace renderovacího enginu pro voxelovou grafiku, dále pak implementace ukázkových příkladů, na kterých bude demonstrována použitelnost renderovacího enginu. Pro ukázkové příklady bude navržen a implementován procedurální generátor terénu.

Požadavky na renderovací engine:
- vykreslování voxelové grafiky
- dynamické načítání scény podle pozice hráče
- parametrizace velikosti vykreslované scény
- využití možností grafické karty pro optimalizaci vykreslování

Požadavky na procedurální generátor:
- samostatný běh s uložením vygenerované scény do souboru
- komunikace s renderovacím enginem
- deterministické chování

Obecné požadavky
- multiplatformní využití
- projekt bude obsahovat automatizované testy
- podrobná návrhová dokumentace pomocí SI metodik, která umožní jednoduché rozšiřování do budoucna

---

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 28. května 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Engine pro renderování a procedurální generování voxelových světů

Bc. Lukáš Hepner

Katedra softwarového inženýrství
Vedoucí práce: Ing. Adam Vesecký

1. května 2022

Poděkování

Tímto chci poděkovat svému vedoucímu, Ing. Adamu Veseckému, za jeho rady, vedení a skvělý předmět APH, který mě k této práci přivedl. Dále bych rád poděkoval své rodině za bezmeznou podporu během studia a psaní této práce. V neposlední řadě bych rád poděkoval svým kamarádům, s kterými jsem mohl prožít tuto éru života.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 1. května 2022 . . . . . .

Odkaz na tuto práci

Hepner, Lukáš. *Engine pro renderování a procedurální generování voxelových světů*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022. Dostupný také z WWW: ⟨`https://github.com/heppyn/MasterThesis`⟩.

Abstrakt

Tato práce se zabývá tvorbou vykreslovací části herního enginu specializujícího se na voxelovou grafiku. K jejímu vykreslení jsou připraveny shadery, na nichž je popsáno, jak zvýšit vizuální kvalitu přidáním světla a stínů. Na připravených shaderech je zároveň ukázáno, jak lze využít možností grafické karty, optimalizovat pomocí ní vykreslování a snížit zátěž procesoru.

Druhá část textu se zabývá procedurálním generováním terénu a využitím L-systémů ke generování vegetace. Pozornost je věnována šumovým funkcím zaručujícím deterministické chování generátoru.

Klíčová slova vykreslovací engine, voxelová grafika, OpenGL, procedurální generování, L-systémy

Abstract

This thesis deals with the creation of a rendering part of a game engine that specializes in voxel graphics. For rendering the graphics, a couple of shaders were created, which are used to describe how to improve visual quality by adding lights and shadows. The prepared shaders show how to utilize the capabilities of a graphics card, use it to optimize rendering and reduce the processor load.

The second part of the text deals with procedural generation and possible uses of L-systems to generate vegetation. Attention is focused on noise functions that guarantee the deterministic behavior of the generator.

Keywords rendering engine, voxel graphics, OpenGL, procedural generation, L-systems

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V posledním desetiletí zažívají velký rozkvět hry s voxelovou grafikou[1]. Mezi jejich zástupce patří Minecraft, Teardown nebo Staxel. Tyto hry se často zaměřují na příležitostné hráče, kteří se ke hře vrací, aby farmařili, budovali své příbytky nebo prozkoumávali procedurálně generovaný svět. Nejen žánry, ale i způsob tvorby her, prošel dramatickým vývojem.

První počítačové hry byly tvořeny jako jeden produkt, úzce spjatý s hardwarem, na který cílil. Při změně hardwaru bylo nutné začít s vývojem znovu, tak aby byly maximálně využity jeho zdroje. Změna přišla v půlce devadesátých let, kdy se hry začaly dělit na své základní komponenty a herní mechaniky.

Termín „herní engine“ vznikl v souvislosti s hrami jako je Doom vyvinutý společností id Software. Architektura Doomu poměrně dobře odděluje své základní komponenty (jako je trojrozměrný vykreslovací systém, detekce kolizí nebo audio systém) od grafiky, herních světů a pravidel hry. Hodnota tohoto rozdělení byla viditelná, když si ostatní vývojáři začali licencovat hry a předělávat je do nových produktů, přidáním nové grafiky, světů, zbraní, postav, vozidel a pravidel hry s pouze minimálními změnami v částech enginu [1].

Moderní herní enginy (Unity, Unreal, GameMaker, Godot, ...) umožňují vyvíjet aplikace pro desktop (Windows, Linux, macOS), mobilní zařízení (Android, iOS) i webové (HTML5) platformy. Poskytují obsáhlou sadu nástrojů umožňující programátorovi zabývat se tvorbou herního obsahu, místo znovu objevování kola [2]. Enginy často podporují tvorbu 2D i 3D her.

S rozvojem her, s otevřeným světem vznikaly větší a větší nároky na designéry, musející vytvářet detaily každého zákoutí světa, do kterého se může hráč dostat. Některé hry proto využívají metody procedurálního generování obsahu.

V rámci vývoje lze generovat drobné části světa, jako je vegetace, a ty následně vydat pro všechny hráče. Opačným směrem se vydávají hry, které

[1] Voxel je bod v prostoru, vykreslený jako krychle.

generují veškerý obsah v průběhu hry. Touto technikou se nejvíce proslavila hra Minecraft, která umožňuje hráči prozkoumávat nekončící terén, generovaný na základě seedu – semínka světa. Každý hráč tak může objevovat svůj vlastní svět, lišící se od všech ostatních.

Cíle

Cílem práce je vytvořit engine vykreslující voxelovou grafiku. Pro část výpočtů a vykreslování bude využita grafická karta.

Hráči bude umožněno pohybovat se libovolným směrem po scéně. Engine proto musí dynamicky generovat svět na základě hráčovy pozice. Části světa, které hráč opustil a jsou mimo jeho vykreslovací vzdálenost, budou uvolněny z paměti. Pro optimální běh na hardwaru s různou výkonností bude možné zvětšit či zmenšit vykreslovanou oblast.

Svět bude generovaný na základě seedu. Všechny části světa musí být stejné při opakovaném běhu generátoru se stejným seedem. Procedurální generátor bude vytvářet svět, co nejvíce připomínající venkovní prostředí, jaké lze nalézt na Zemi. Svět bude obsahovat různé biotopy, podle nichž se bude měnit vegetace, která bude taktéž procedurálně generovaná. Výsledek běhu bude možné okamžitě zobrazit pomocí enginu nebo exportovat do souboru.

Kapitola 1

Architektura a technologie

Na trhu se nachází celá řada herních enginů, jejichž cenové politiky umožňují vytvářet hry jak nezávislým vývojářům, kteří mohou využít enginy dostupné zcela zdarma jako je Godot či Cocos2D, případně jsou zpoplatněné autorskými poplatky při dosažení zisku, jako je CryEngine, Unreal Engine či Unity [3]. Herní studia často využívají placených licencí, zajišťujících podporu od tvůrce enginu.

Hry s voxelovou grafikou nemusejí vznikat jen v enginech pro ně tvořených, ale i v enginech vykreslujících polygonovou grafiku. Výhodou jejich obecnosti je větší rozsah projektů, které díky nim mohou vzniknout. To vede k větším investicím a aktivnějšímu vývoji. Tyto enginy poskytují komplexnější sadu nástrojů pro vývoj a na trhu převažují nad enginy vykreslujícími specifický druh grafiky.

I přesto na trhu existují enginy voxelové. Jedním ze zástupců open source projektů je voxel.js, určený pro vývoj her běžících v prohlížeči. Jak název napovídá, vývoj her probíhá v JavaScriptu [4].

Placeným zástupce je Voxel Farm [5]. Jeho výhodou je vyšší výkonnost, než které dosahuje voxel.js, a možnost integrace s Unity nebo Unreal enginem. Zatímco ve voxel.js lze vytvářet hry podobné Minecraftu, Voxel Farm umožňuje práci s voxely s menším měřítkem (scéna jich tím pádem obsahuje větší množství), díky němuž může svět vypadat více realisticky. Hráč je do světa vtažen díky možnosti destrukce terénu a interakce s objekty, probíhající na úrovni voxelů. Tato vlastnost je velice cenná a přitahuje proto studia jako je EA nebo Take-Two Interactive [6].

1.1 Architektura herních enginů

Herní enginy se skládají z vrstev, kde vyšší vrstvy komunikují s nižšími vrstvami. Obecný engine se může skládat z těchto vrstev, seřazených od nejnižší, po nejvyšší [7]:

1. Hardware — reprezentuje platformu, na které výsledná hra poběží. PC, Xbox, PS, iPhone, Switch.

2. Ovladače — odstiňují operační systém od detailů komunikace s HW. NVidia, Realtek, Intel HD.

3. Operační systém — zajišťuje společný běh programů, z nichž jeden je herní engine. Windows, Linux, MacOS, Android, iOS.

4. Knihovny třetích stran — poskytují algoritmy, kontejnery. Komunikují s grafickou kartou. STL, Boost, GLM, OpenGL, Vulkan.

5. Nezávislost na platformě — obaluje systémová volání. Zajišťuje přenosnost mezi platformami. Souborový systém, primitivní datové typy, vlákna.

6. Hlavní systémy — zajišťují základní funkce enginu. Aserce, matematické funkce, náhodná čísla, alokátory, textové řetězce.

7. Moduly — komplexní systémy zastřešující jednotlivé oblasti enginu. Vykreslování, manažer scény, manažer zdrojů, fyzika, multiplayer.

8. Herní subsystémy — můžou být více či méně svázané s hrou samotnou. Zajišťují společnou funkcionalitu her, jako je: zobrazení menu, mechaniky hráčem ovládané postavy, kamera, umělá inteligence, skriptovací engine.

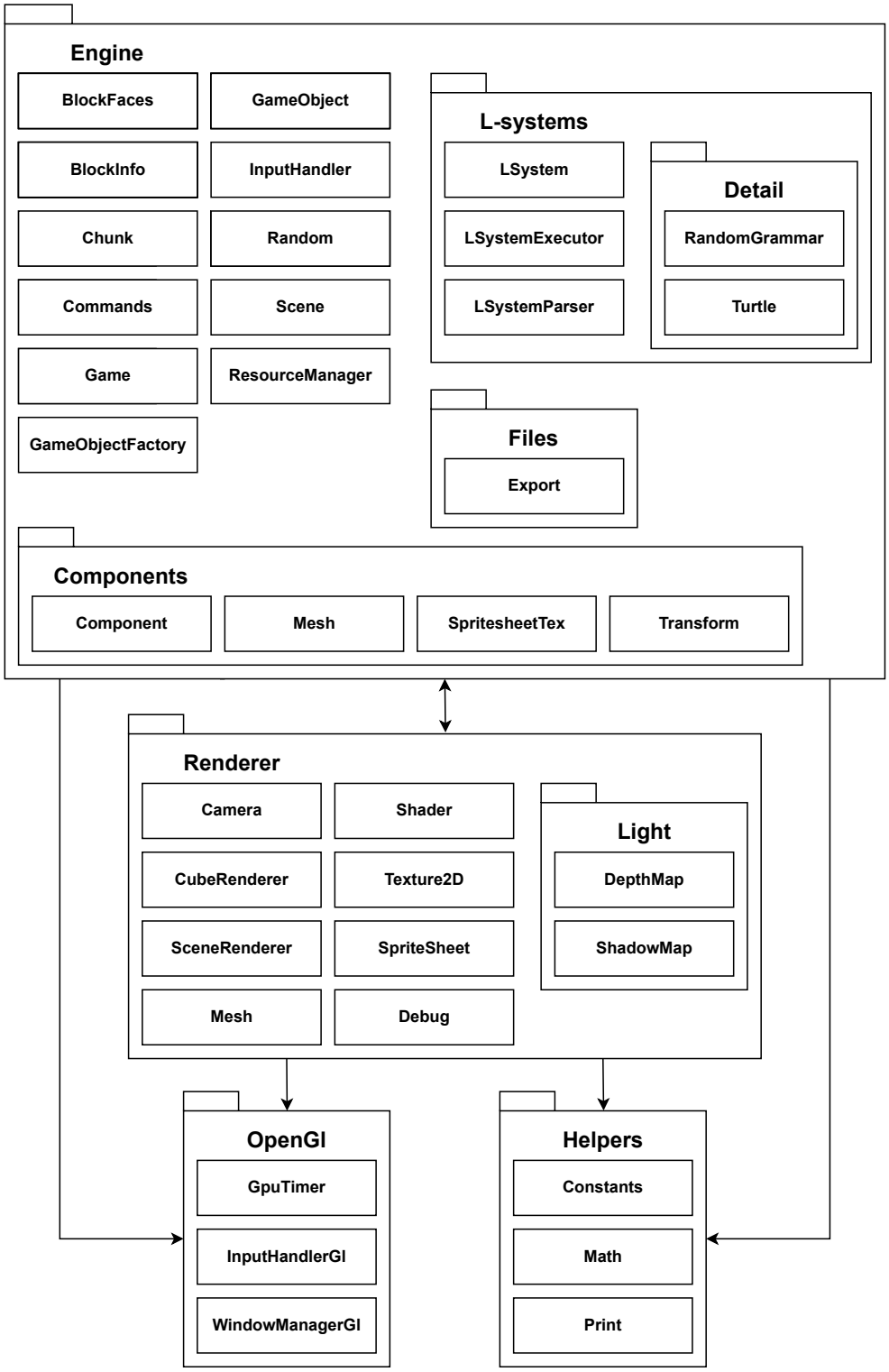
1.2 Návrh architektury

Engine je rozdělen do několika modulů a sub modulů, seskupujících funkcionalitu stejné kategorie. Diagram 1.1 ukazuje vzájemnou komunikaci mezi moduly a třídy jež obsahuje.

1.2.1 Engine

Hlavním modulem je `Engine`, zajišťující vytváření herních objektů, jejich správu v paměti, práci s náhodnými jevy, zpracování vstupů a správu zdrojů. Samostatným modulem je modul `L-systems`, umožňující načítání, zpracovávání a generování objektů na základě L-systému. Třídy zajišťující implementaci, s kterými by uživatel enginu neměl interagovat jsou umístěny v modulu `Detail`.

Herní objekty by neměly být vytvářeny přímo. Jejich tvorbu zajišťuje tovární třída `GameObjectFactory`. Herní objekty implementují ECS (Entity-Component-System) architekturu [8]. Vlastnosti objektu jsou určeny jeho komponentami. Ty určují pozici ve světě (`Transform`), tvar objektu (`Mesh`) a jeho texturu (`Mesh` nebo `SpritesheetTex`).



Obrázek 1.1: Architektura enginu

Scéna je složena z bloků terénu (třída `Chunk`), které obsahují herní objekty. Správu scény zajišťuje třída `Scene`, která ukládá chunky do *std::map* (seřazená asociativní mapa) využívající jejich pozici jako klíč. Pozice chunku – *glm::vec2* může být použita jako klíč díky porovnávacímu operátoru, který vektory řadí lexikograficky po složkách. Rozdělení na chunky zrychluje operace jako je ořezání viditelné herní plochy, po kterém třída `Scene` upraví pole ukazatelů na data objektů, které mají být vykresleny.

Každý chunk obsahuje dvě pole herních objektů, rozdělené na průhledné a neprůhledné. Herní objekty se mohou překrývat a mít libovolný posun vůči počátku scény. Z důvodu úspory paměti nejsou rozděleny na sub-voxely, které se nepřekrývají.

Velikost chunku je uložena ve statické proměnné `ChunkSize`. V základním nastavení je velikost chunku (šířka a hloubka) 16 x 16 bloků. Třída `Chunk` si pro každý blok ukládá metadata obsahující výšku terénu, biotop, teplotu, vlhkost a informaci o tom, jestli se na bloku nachází strom. Metadata jsou uložena ve třídě `BlockInfo`, která data reprezentuje jako 32bitovou hodnotu (*uint32_t*). Metadata jsou součástí exportované scény a mohou být využita pro rozdílnou interpretaci scény.

Modul `Helpers` zajišťuje převod objektů na textové řetězce, matematické operace (porovnání čísel typu float, porovnávací operátory...) a sdružuje konstanty používané v enginu.

Moduly `Engine` a `Helpers` obsahují veškerou funkcionalitu používanou generátorem terénu.

1.2.2 Renderer

Moduly `Renderer` a `OpenGl` implementují veškerou funkcionalitu spojenou s OpenGL – ostatní moduly jsou nezávislé na vykreslovací knihovně. Funkce přímo nesouvisející s vykreslováním (správa oken, zpracování vstupu, měření délky operací na GPU) se nacházejí v modulu `OpenGl`.

Modul `Renderer` zastřešuje veškerou funkcionalitu spojenou s vykreslováním a komunikací s grafickou kartou. Součástí modulu jsou i pomocné třídy umožňující snazší práci s koncepty OpenGL jako jsou shadery, textury... Hlavní vykreslovací metody obsahují třídy `SceneRenderer` a `CubeRenderer`, vykreslující celou scénu, respektive jeden či více objektů najednou.

1.3 OpenGL

K interakci s grafickou kartou bylo zvoleno OpenGL vyvíjené skupinou Khronos. Pro vývoj byla zvolena stabilní verze 4.6, vydaná v roce 2017. Tato verze přináší pokročilé funkce využívané v shaderech, ale je na trhu dostatečně dlouho, aby byla podporována většinou moderních grafických karet.

1.3.1 Vykreslovací řetězec

OpenGL představuje stavový automat. Jeho výchozí hodnoty jsou nastaveny při spuštění programu (např. způsob mísení barev, povolení testu hloubky...). Některé se musí měnit podle dat, které se mají vykreslit (např. face culling). Grafická karta má před každým voláním pro vykreslení scény uložena data objektů ve svých bufferech, přiřazené textury do texturovacích jednotek, nastavené hodnoty proměnných vývojářem definovaných programů... Tyto parametry představují stav OpenGL, podle něhož se vykreslí scéna.

Kroky vykreslovacího řetězce jsou vysoce specializované a výstup každého z nich je použit jako vstup pro další. Tyto kroky je možné masivně paralelizovat a využít tisíců jader, které může grafická karta obsahovat. Každé jádro spouští malý program, pro každý krok vykreslovacího řetězce. Tyto programy jsou nazývané shadery. Některé ze shaderů může definovat vývojář a nahradit nimi existující výchozí shadery [9].

Vykreslovací řetězec se skládá z následujících kroků [10]:

1. Specifikace vrcholů — načtení formátu vrcholů a předání dat, která budou zpracována dále v řetězci.

2. Vertex shader — provede zpracování vrcholu na základě uživatelem specifikovaného programu a předá ho k dalšímu zpracování. Zpracován může být pouze jeden vrchol a ten musí být předán do dalšího kroku. Poskytnutí vertex shaderu je povinné.

3. Teselace — nepovinný krok, který může rozdělit primitivum (trojúhelník, úsečku, ...) na několik menších primitiv [11].

4. Geometry shader — uživatelem definovaný program, zpracovávající primitiva. Výstupem je nula nebo více primitiv. Vstupní a výstupní primitiva musí být přesně definovaná. Geometry shader může změnit jeho geometrii, provést transformaci souřadnic...Tento krok není povinný.

5. Post-processing vrcholů — složená primitiva (např. pruh trojúhelníků) jsou převedena na jednoduchá primitiva (úsečky, body, trojúhelníky). Primitiva, jejichž část se nachází mimo prostor obrazovky, jsou rozdělena, aby vznikla nová, jež jsou uvnitř prostoru obrazovky. Trojúhelníková primitiva mohou být vyřazena, pokud nemíří k pozorovateli. Všechna nová primitiva jsou uložena do výstupních bufferů pro další zpracování.

6. Rasterizace — primitiva, která se dostanou do této fáze, jsou převedena na fragmenty. Fragment je soubor hodnot obsahující výstupy předchozích shaderů. Jejich hodnota je nastavena interpolací hodnot vrcholů, z kterých se primitivum skládá. Každý fragment obsahuje pozici v prostoru obrazovky [12].

7. Fragment shader — výstupem fragment shaderu je list barev, které budou zapsány do výstupních bufferů, hodnota hloubky a hodnota šablony (stencil value). Fragment shader může být definován uživatelem. Pokud není, hodnota barev není definovaná, je pouze zapsána hodnota hloubky a šablony.

8. Operace provedené na vzorcích (Per-Sample Operations) — výstupní data fragmentu jsou podrobena sérii testů (mohou být specifikované uživatelem, např.: hloubkový test), které je mohou vyřadit z podílení se na výsledné barvě pixelu. Pokud projdou, je provedeno míchání barev s barvami, které již obsahuje framebuffer.

1.3.2 Podpůrné knihovny

OpenGL je pouze specifikace, implementovaná členy skupiny Khronos. Mezi členy patří výrobci grafických karet, vývojáři operačních systémů, tvůrci her, ...[13] Implementace je napsaná v jazyce C a není specifická pro jeden operační systém. Operace jako je vytváření okna, zpracování uživatelského vstupu, se liší v závislosti na operačním systému a OpenGL od nich abstrahuje. Z tohoto důvodu je potřeba využít další knihovnu, zajišťující tyto funkce. K tomuto účelu byla vybrána knihovna GLFW, vytvářející korektní kontext – stav OpenGL [14].

Ovladače grafických karet jsou poskytovány výrobci karet. OpenGL je pouze standard, definující signaturu funkcí a jejich chování. Při překladu kódu proto není známé, jaký ovladač bude použit, a lokace funkcí se musí zjistit až za běhu programu. Funkce pro zjišťování adres funkcí jsou závislé na operačním systému. Skupina Khronos proto doporučuje využít jednu z knihoven pro načtení funkcí OpenGL [15]. K tomuto účelu byla využita knihovna GLAD.

1.3.3 Jazyk pro psaní shaderů

K programování shaderů byl zvolen OpenGL Shading Language – GLSL, jakožto hlavní jazyk určený pro OpenGL nevyžaduje žádné rozšíření. GLSL je jazyk podobný C – obsahuje stejné strukturální prvky (smyčky, větvení, ...). Avšak rozšiřuje datové typy C například o vektory, matice, textury... a obsahuje vlastní standardní knihovnu [16].

Poslední vydaná verze je shodná s verzí OpenGL (4.6). Pro vývoj je však možné používat různé verze v závislosti na požadovaných funkcích. Použitím nižší verze se teoreticky rozšíří počet kompatibilních zařízení, v praxi je však omezen verzí OpenGL, kterou používá engine.

1.4 Další knihovny

Projekt obsahuje knihovny řešící specifické problémy. První z nich je OpenGL Mathematics – GLM. Matematická knihovna určená pro práci s OpenGL. GLM poskytuje třídy a funkce navržené a implementované se stejnými jmennými konvencemi jako GLSL. Neomezuje se však pouze na funkcionalitu GLSL. Poskytuje funkce na transformaci matic, kvaterniony, komprese dat, náhodná čísla, šum, atd... [17]

Další knihovnou je *stb image* sloužící k načítání obrázků. Knihovna má omezený počet typů souborů, s kterými dokáže pracovat. Její použití je však snadné a engine používá pouze dva typy obrázků – JPEG a PNG. Její autor o ní říká, že je primárně určená pro herní vývojáře a další osoby, které se dokáží vyhnout problematickým obrázkům a potřebují pouze jednoduché rozhraní [18].

Pro generování terénu bude využit Perlinův a Simpex šum implementovaný knihovnami *siv::PerlinNoise* [19] a *SimplexNoise* [20]. Knihovny dokáží generovat 1D, 2D nebo 3D šum. Obsahují taktéž podporu pro oktávový šum. Důležitou součástí knihoven je možnost poskytnout vlastní seed, na jehož základě je šum generován.

Kapitola 2

Vykreslení scény

Vykreslovací engine má za úkol převod scény definované v 3D prostoru na 2D plochu zobrazovacího zařízení hráče. Tento proces začíná na CPU, kde jsou herní objekty převedeny na data (matice), která jsou poslána grafické kartě. Zde jsou za pomocí série kroků transformována na zobrazitelné pixely.

V každém snímku je scéna vykreslena pomocí série za sebou jdoucích vykreslovacích příkazů OpenGL. Každý z nich upraví hodnotu pixelů výstupního bufferu, jejich přepsáním nebo smícháním současné a nové barvy. Před každým vykreslovacím příkazem je potřeba nastavit stav OpenGL podle dat, která se mají aktuálně vykreslit. Před vykreslením může taktéž dojít ke změně použitého shaderu nebo výstupního bufferu (vykreslení mimo obrazovku).

2.1 Předání dat grafické kartě

Scéna je složena z kusů terénu (třída `Chunk`), obsahující herní objekty. Chunky jsou rozmístěné na ploše vymezené osami `x` a `z`. Třída `Scene` má za úkol správu chunků a serializaci jejich dat do 1D pole, předané grafické kartě.

Každý herní objekt lze reprezentovat jako matici 4x4 obsahující informace o posunu vůči počátku světa, škálování a textuře.

Převod herního objektu na matici:

```
for (const auto& o : obs) {
    assert(o.HasComponent<Components::Transform>());
    auto model =
      o.GetComponent<Components::Transform>().ModelMat();

    assert(o.HasComponent<Components::SpritesheetTex>());
    const auto& texPos =
      o.GetComponent<Components::SpritesheetTex>().GetTexPos();
    Helpers::Math::PackVecToMatrix(model, texPos);
```

```
    buffer[cube]->push_back(model);
}

[[nodiscard]] glm::mat4 ModelMat() const {
    auto model = glm::mat4(1.0f); // identity matrix
    model = glm::translate(model, Position);
    return glm::scale(model, Scale);
}
```

2.1.1 Instancing

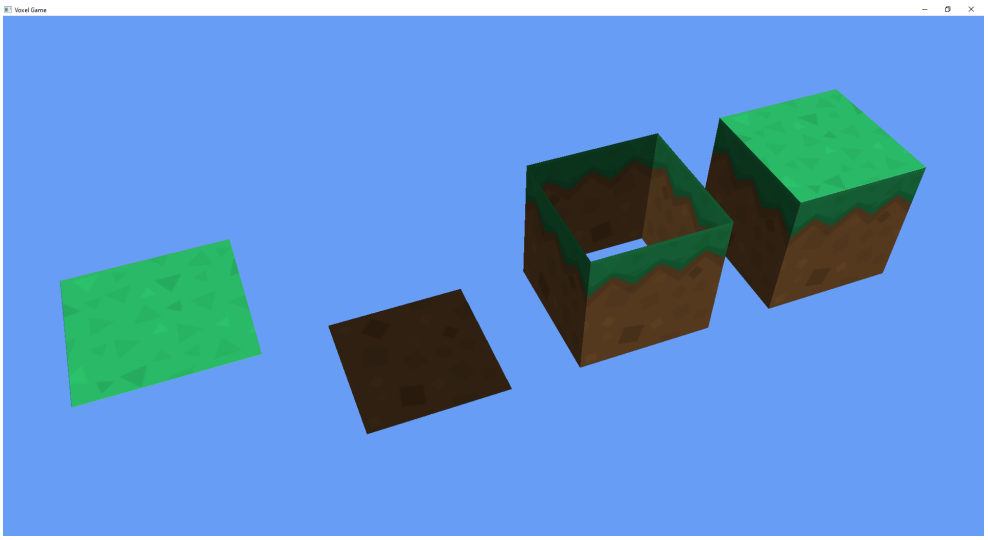
Herní scéna obsahuje velké množství objektů se stejnou geometrií (např. krychle), s různou transformací vůči počátku světa. Každá krychle je složena z 12 trojúhelníků. Pokud by pro každou z nich bylo voláno samostatné vykreslení (draw call), velice rychle dojde k drastickému snížení výkonu (komunikace s kartou přes sběrnici, uložení dat do patřičných bufferů...). Lepší řešení je poslat všechna data kartě najednou a ty následně vykreslit pomocí stejné geometrie. Toto řešení se nazývá instancing [21].

Scéna umožňuje definovat vlastní geometrie – krychle, krychle bez podstav, nebo jakákoliv jiná kombinace stěn krychle. Pro každou z nich alokuje místo na grafické kartě, uloží do něj data objektů a následně samostatně vykreslí každou z nich.

```
// bind data
glBindBuffer(GL_ARRAY_BUFFER, InstanceDataBufferIds_[cube]);
unsigned offset = 0;
for (const auto& chunk : instancesData) {
    glBufferSubData(
        GL_ARRAY_BUFFER,
        offset * sizeof(glm::mat4),
        chunk->size() * sizeof(glm::mat4),
        chunk->data());
    offset += chunk->size();
}

CubeRenderers_[cube].GetDefaultMesh().BindBatchAttribPtrs();
```

Rozdílné geometrie jsou užitečné, pokud chceme definovat krychli s rozdílnými podstavami a stěnami. Na obrázku 2.1, lze vidět blok trávy složený ze tří částí.



Obrázek 2.1: Složení bloku ze tří částí

2.2 Společná data objektů

Instancing umožňuje grafické kartě používat společná data pro všechny vykreslované objekty. Jeho protějškem v objektovém světě je návrhový vzor Flyweight [22].

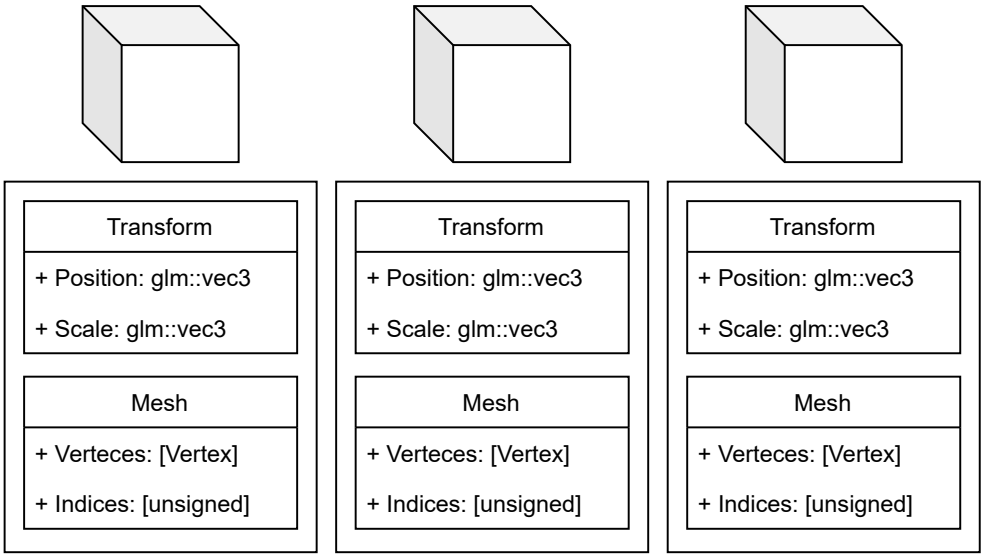
Pokud má být objekt vykreslitelný musí obsahovat komponentu `Transform` (určující jeho pozici a velikost) a komponentu `Mesh` (představující geometrii objektu) – obrázek 2.2.

`Mesh` obsahuje informace o vrcholech (`Vertex`) vykreslovaných trojúhelníků – pozici na scéně, normálový vektor k úsečce mezi dvěma vrcholy trojúhelníku, pozici na textuře. A pole indexů, které říká, z jakých vrcholů jsou trojúhelníky složeny. Každý vrchol v má velikost $8 \cdot 4 = 32$ B a `Mesh` jich obsahuje $6 \cdot 4$. Každá stěna krychle má čtyři různé vrcholy[2]. Pole indexů má velikost $12 \cdot 3 \cdot 4$ B (počet trojúhelníků, počet vrcholů trojúhelníku, velikost `unsigned`). Celková velikost `Mesh`e představujícího krychli je:

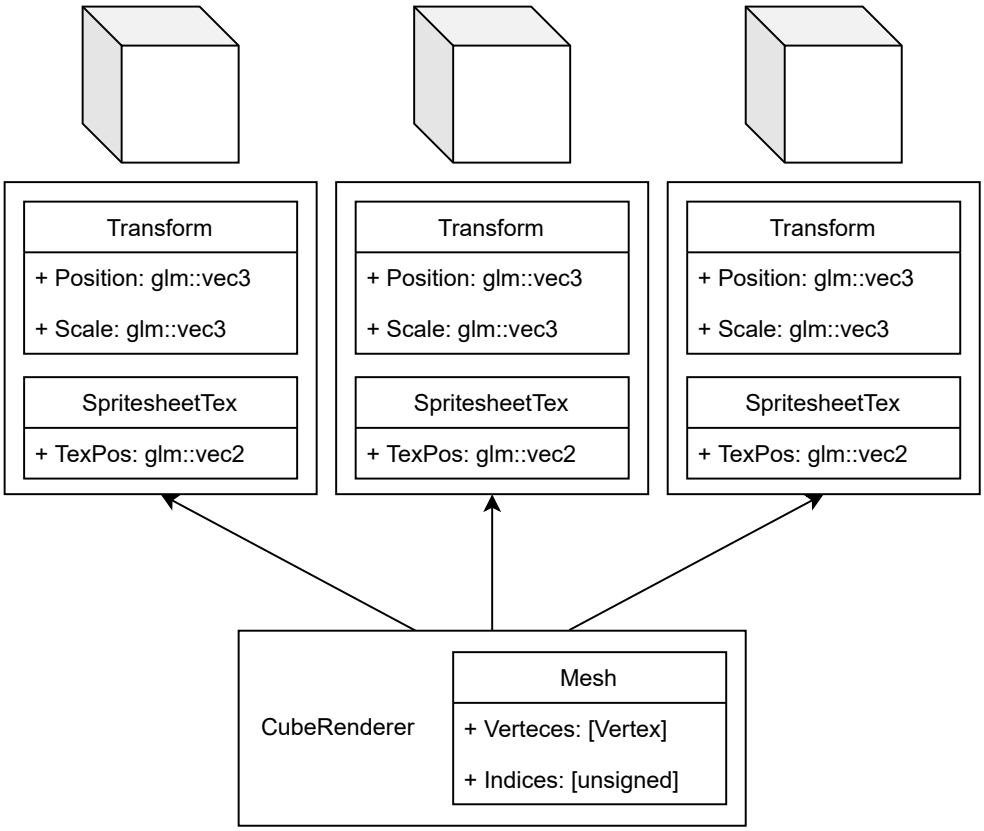
$$8 \cdot 4 \cdot 6 \cdot 4 + 12 \cdot 3 \cdot 4 = 912\ B$$

Pokud má být engine schopný vykreslovat desetitisíce objektů, je tato paměťová náročnost neúnosná. Komponentu `Mesh` není možné odebrat z herního objektu, protože obsahuje pozici na textuře, lišící se mezi objekty. Pozice na textuře představuje čtverec, který bude vybrán z textury a aplikován na povrch krychle. Všechny objekty mají texturu čtverce. Díky tomu je možné definovat čtverec na počátku textury a jeho posun uložit do samostatné komponenty – `SpritesheetTex`. Toto nové rozložení je znázorněno na obrázku 2.3.

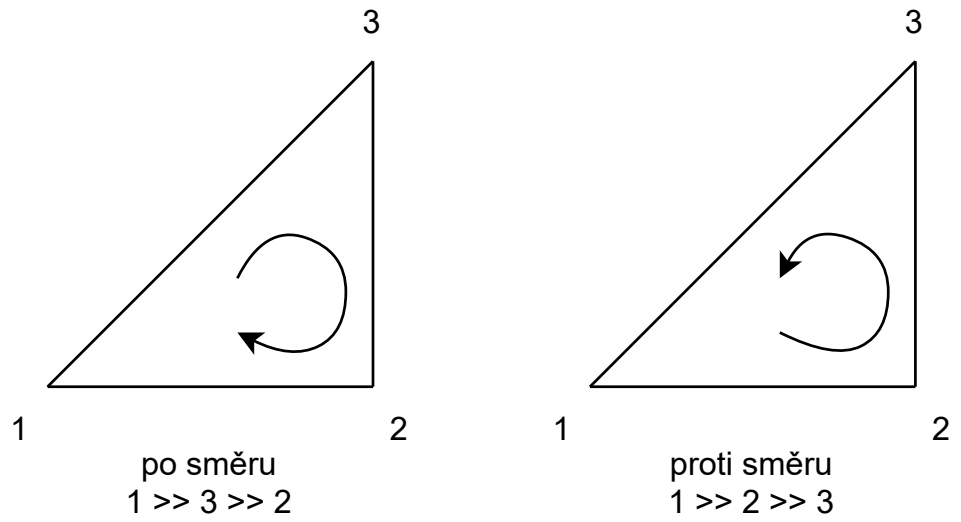
[2]Krychle má 8 vrcholů. Vrcholy jednotlivých stěn se ale liší v normálovém vektoru a po-



Obrázek 2.2: Komponenty herního objektu



Obrázek 2.3: Použití vzoru Flyweight



Obrázek 2.4: Pořadí vrcholů

Komponenta `SpritesheetTex` obsahuje dvě čísla ve formátu float (`float` je zvolen pro jednodušší komunikaci s grafickou kartou – všechna předaná data mají formát float). Komponenta v paměti zabírá pouhých 8 B. Velikost herního objektu byla redukována o 904 B.

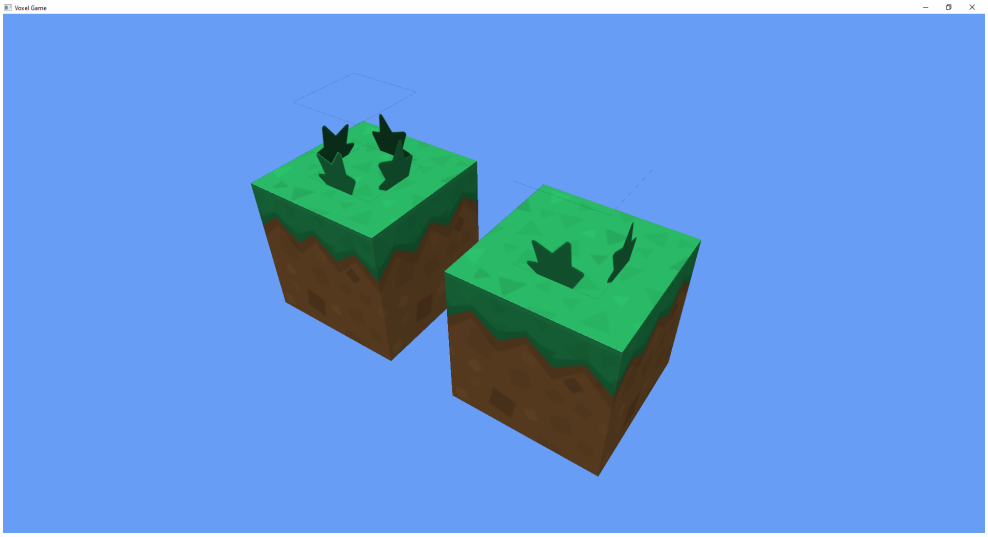
`Mesh` není dále používána jako komponenta herního objektu. Její implementace ze jmenného prostoru `Renderer`, kterou komponenta `Mesh` obsahuje jako svůj atribut, je použita pro vykreslení všech objektů se stejnou geometrií najednou. Vykreslení objektů zajišťuje třída `CubeRenderer`, viz obrázek 2.3.

2.3 Vyřazení neviditelných částí objektu

Část scény tvoří krychle mající všech šest stěn vyplněných neprůhlednou texturou. Při pohledu na krychli může hráč vidět maximálně tři její stěny najednou (z určitých úhlů pouze jednu nebo dvě). Minimálně 50 % každé krychle je vykreslováno zbytečně. OpenGL je schopno vyřadit stěny které směřují pryč od hráče z vykreslovacího procesu a snížit počet volání fragment shaderu. Tato technika se nazývá *face culling* [23].

K rozpoznání stěn (trojúhelníků), které směřují pryč od hráče OpenGL používá pořadí jeho vrcholů. Vrcholy mohou být definované ve směru nebo protisměru hodinových ručiček – obrázek 2.4. Při pohledu na grafické primitivum z druhé strany, se změní pořadí jeho vrcholů. V základním nastavení OpenGL, jsou primitiva s vrcholy definovanými po směru hodinových ručiček, považována za směřující k hráči.

zicí na textuře.



Obrázek 2.5: Face culling

Tato technika není v základním nastavení OpneGL zapnutá. Musí se povolit zavoláním funkce:

```
glEnable(GL_CULL_FACE);
```

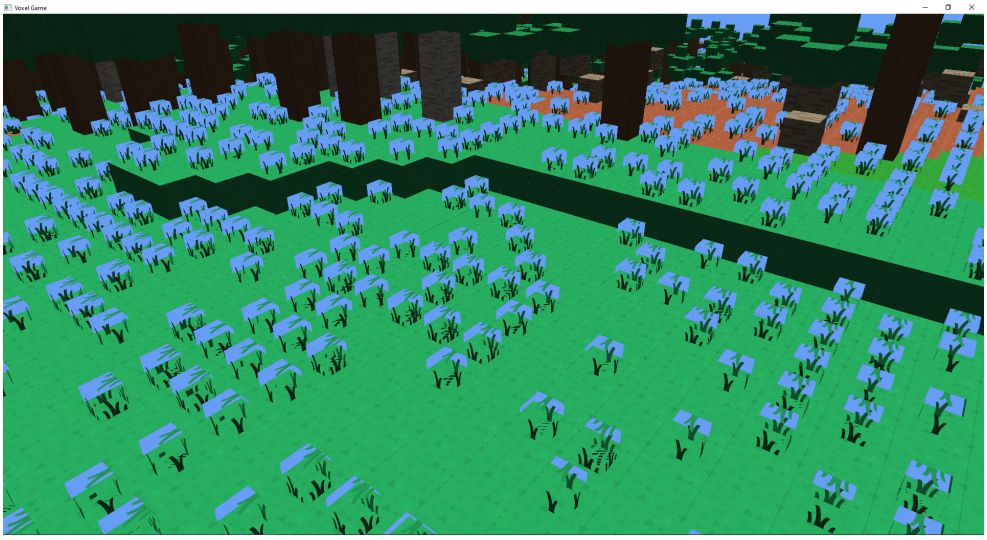
Pro objekty, které nemají všech šest stěn nebo mají průhlednou texturu, je nutné vypnout face culling. Hráči se zobrazí i vnitřek objektu, který by jinak nebyl vidět. Příkladem je vykreslování trávy – obrázek 2.5. Na pravém bloku je tráva vykreslena se zapnutým face cullingem. Na levém bloku je face culling vypnutý.

2.4 Průhledné a poloprůhledné textury

Při vykreslování textur majících průhlednost (alfa < 1)[3] může nastat několik problémů. Prvním z nich je pořadí vykreslování objektů. Pokud má textura průhlednost, musí se její barva zkombinovat s barvami textur, které překrývá. Na obrázku 2.6 jsou textury vykreslovány postupně z pravého dolního rohu, po řádcích směrem k levému hornímu rohu. Lze si všimnout, že průhledná část trávy je korektně smíchána s barvou bloku pod ní (je vykreslen první). Bloky v pozadí obrázku jsou překryté modrou barvou v místech, kde je textura trávy průhledná.

Tento jev nastává kvůli použití depth bufferu, ten šetří výpočetní výkon zahozením fragmentů, které by nebyly vidět. Jsou do něj však uloženy všechny

[3]Použité textury jsou uloženy ve formátu PNG. Alfa může nabývat hodnot 0–255, ale OpenGL ji převede do floatu v rozsahu 0–1.



Obrázek 2.6: Prolínání průhledných částí textury s pozadím

fragmenty nezávisle na průhlednosti. V ideálním případě by nejdříve měly být vykreslené všechny fragmenty, které se budou podílet na barvě pixelu, od nejvzdálenějšího k nejbližšímu [24].

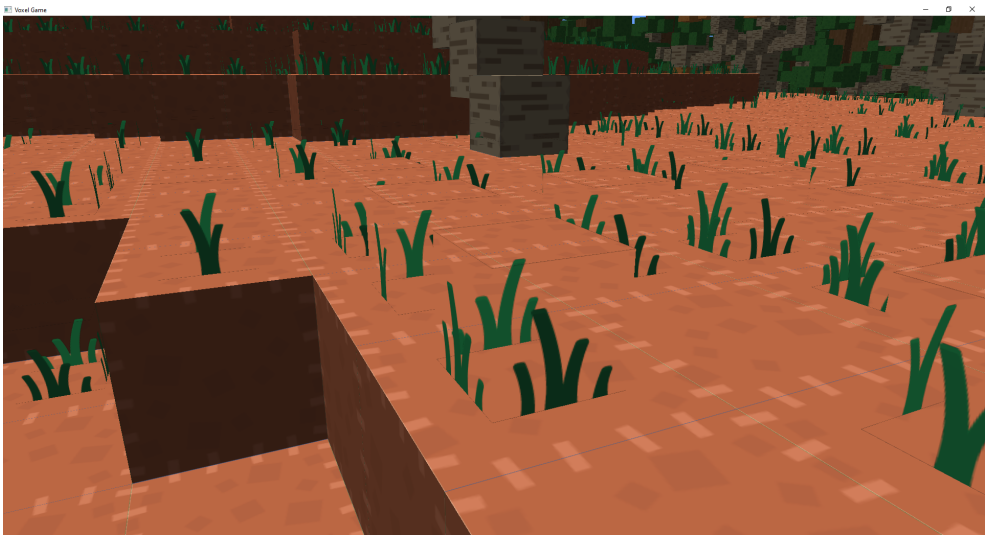
Toto řešení vyžaduje řazení objektů na scéně podle jejich vzdálenosti od kamery. Poloha kamery se s pohybem hráče neustále mění a řazení objektů by snižovalo výkonost.

Avšak objekty bez průhledných části překryjí všechny objekty nehledě na průhlednost. Scénu tak lze rozdělit na dvě části a objekty bez průhlednosti vykreslit jako první. Na obrázku 2.7 je tato technika implementována. Barva průhledné části trávy je korektně smíchána s barvou bloku v pozadí.

Problém s pořadím textur obsahujících průhlednost stále přetrvává. Na obrázku 2.7 lze jasně vidět průhlednou část textury trávy, překrývající trávu v pozadí. Pro textury mající části úplně průhledné nebo úplně neprůhledné lze problém vyřešit ve fragment shaderu. Všechny framenty mající průhlednost menší, než stanovená mez, budou vyřazeny z vykreslovacího řetězce.

```
float alpha = vec4(texture(texture_diffuse1, TexCoord)).a;
if (alpha < 0.05)
    discard;
```

Problém s poloprůhlednými texturami však přetrvává a pro jeho vyřešení je nutné poloprůhledné textury seřadit.



Obrázek 2.7: Zastínění průhlednou částí textury

Kapitola 3

Osvětlení scény

Osvětlení v reálném světě je extrémně složitý problém. Jeho přesná simulace v aplikacích reálného času je výpočetně náročná. Pro zjednodušení výpočtu existují modely, které produkují výsledky podobné reálnému světu.

3.1 Phongův osvětlovací model

Jednou z aproximací reálného světa je Phongův osvětlovací model, skládající se ze tří hlavních částí [25].

- Okolní (ambient) světlo — I za tmy jsou objekty nasvíceny světlem odraženým od ostatních objektů (měsíc) nebo vzdáleným zdrojem světla (hvězdy). Objekty většinou nejsou zcela tmavé. Nastavením konstanty pro ambientní osvětlení můžeme simulovat tento jev.

- Difúzní světlo — Simuluje dopad světla na objekt. Pokud je objekt natočen ke zdroji světla, bude ním více ovlivněn.

- Spekulární (přímé) světlo — Simuluje odraz světla od lesklého předmětu.

Výpočet osvětlení může být prováděn na CPU, nastavením světlosti každé stěny bloku. Pokud má být docíleno věrnějšího výsledku, je nutné počítat světlost pro každý fragment objektu (úhel dopadu světla se může výrazně lišit pro fragmenty na opačných stranách objektu). Výpočet je proto prováděn na grafické kartě, přesněji ve fragment shaderu[4].

Ambientní složka světla je předána grafické kartě, kde je vynásobena s barvou objektu – textury.

[4]Výpočet je možné provést i ve vertex shaderu a fragment shader by pouze interpoloval hodnoty světlosti. Tato technika snižuje počet provedených výpočtů, ale zároveň snižuje vizuální kvalitu osvětlení.

```
vec3 ambient =
    light.ambient * vec3(texture(texture_diffuse1, TexCoord));
```

Pro výpočet difuzní složky, je shaderu předána pozice světla na scéně (`light.position`) a jeho barva (`light.diffuse`). Světlo dopadající kolmo na fragment má největší efekt na jeho výslednou barvu. Pro výpočet úhlu dopadu je využit normálový vektor fragmentu[5]. Vynásobením (skalární součin) normálového vektoru fragmentu a vektoru směřujícího od fragmentu ke světlu, je získána hodnota 1 pro vektory svírající nulový úhel a 0 pro kolmé vektory. Rozsah hodnot je zaručen normalizací vektorů.

Pokud vektory svírají úhel větší než 90° je hodnota skalárního součinu záporná. Tuto situaci si lze představit jako dopad světla z opačné strany fragmentu. Ten by tedy neměl být osvětlený, `diff` je nastaven na 0. Hodnota difuzní složky je vynásobena s barvou světla a fragmentu.

```
vec3 lightDir = normalize(light.position - FragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * light.diffuse *
    vec3(texture(texture_diffuse1, TexCoord));
```

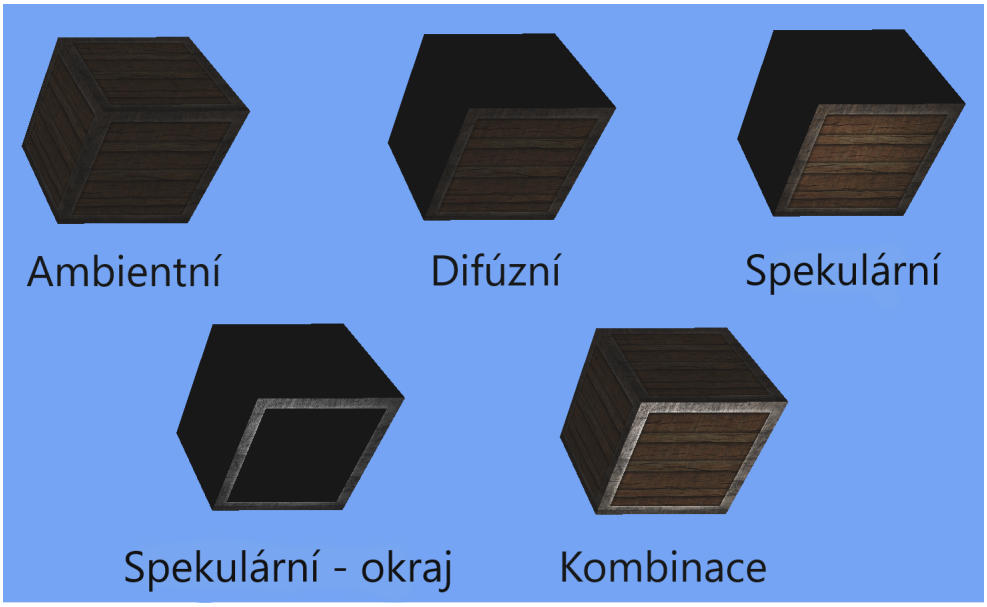
Intenzita odraženého světla (lesklá složka) je závislá na pozici pozorovatele (předána shaderu jako `view_pos`). Pokud odražené světlo směřuje přímo do oka pozorovatele, je jeho intenzita nejvyšší. Vektor odraženého světla lze spočítat pomocí funkce `reflect`. Její první parametr je vektor směřující od světla k fragmentu, tedy vektor opačný k `lightDir`. Druhý parametr je normálový vektor.

Intenzita je opět spočítána pomocí skalárního součinu, a umocněna lesklostí materiálu. Čím vyšší lesklost, tím méně je světlo rozptýleno do všech směrů a velikost efektu se zmenší [25].

```
vec3 viewDir = normalize(view_pos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec =
    pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
vec3 specular = spec * light.specular *
    vec3(texture(texture_diffuse1, TexCoord));
```

Na obrázku 3.1 je vykreslená bedna osvícena všemi druhy světel a jejich kombinací.

[5]Normálový vektor je předán vertex shaderu jako jeden z atributů vrcholu trojúhelníku. Ten ho následně předá fragment shaderu.



Obrázek 3.1: Phongův osvětlovací model

3.2 Materiál objektu

Obrázek 3.1 zobrazuje dvě možnosti použití lesklého světla. Bedna jejíž celý povrch odráží světlo působí nepřirozeným dojmem. V reálném světě dřevo světlo neodráží[6]. Lesknout by se měl pouze její kovový okraj, a to jen v místech kde není poškozen.

Tento problém je vyřešen předáním nové textury pro lesklé světlo. Řádek vyhodnocující barvu lesklého světla nebude používat texturu `texture_diffuse1`, na místo ní využije `texture_specular1`.

```
vec3 specular = spec * light.specular *
    vec3(texture(texture_specular1, TexCoord));
```

Lesklá textura nemusí definovat barvu odrazu – odražené světlo má barvu zdroje světla, ale pouze její intenzitu. Dřevěná část textury má černou barvu (žádný odraz), kovová část je převedena do černo-bílého spektra [26].

Pokud objekt nemá používat lesklé odrazy, nemusí být textura definována. Při změně nastavení textur, jsou všechny navázané textury odpojeny (*unbound from texture samplers*). Následně jsou navázané všechny definované textury.

```
unsigned int diffuseNr = 1;
unsigned int specularNr = 1;
```

[6]Dřevo s naleštěným povrchem světlo odrážet může, tato povrchová úprava však pravděpodobně nebude aplikována na přepravní bednu.

```
for (unsigned int i = 0; i < Textures.size(); i++) {
    // retrieve texture number (the N in diffuse_textureN)
    std::string number;
    std::string name = Textures[i].Type_;
    if (name == "texture_diffuse")
        number = std::to_string(diffuseNr++);
    else if (name == "texture_specular")
        number = std::to_string(specularNr++);

    // now set the sampler to the correct texture unit
    shader.SetInteger((name + number).c_str(), i);
    Textures[i].Bind(i);
}
```

Pokud dojde ke čtení z textury, která není navázána, bude vrácena černá barva, představující nulový lesk [27].

3.3 Zdroje světla

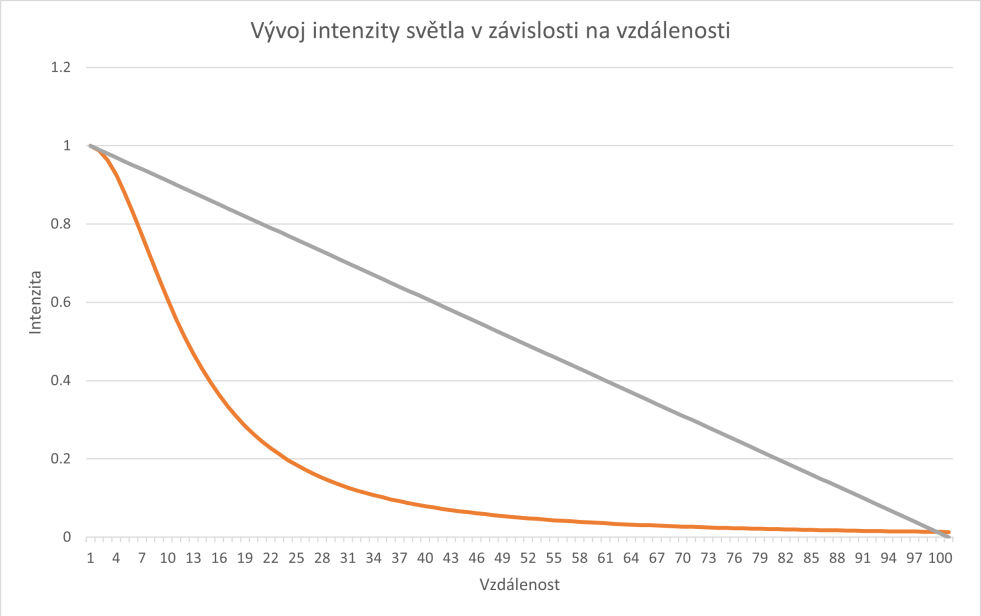
Výše popsaná implementace představuje bodový zdroj světla, jehož intenzita neklesá s uraženou vzdáleností. Tento model nereprezentuje reálné chování světla. Zavedené jsou dva nové druhy světla:

- Globální osvětlení — Představuje zdroj světla nekonečně vzdálený od scény, jehož paprsky jsou rovnoběžné a jeho intenzita se nemění. Jeho pomocí lze modelovat objekty jako je Slunce.
- Bodové osvětlení — Paprsky se šíří všemi směry. Intenzita klesá s uraženou vzdáleností.

3.3.1 Globální světlo

Pro výpočet globálního světla není potřeba jeho poloha, pouze směrový vektor jeho paprsků, definovaný směrem od zdroje světla. Při výpočtu intenzity světla byl použit směr od fragmentu ke světlu. Jedinou změnou oproti dosavadnímu výpočtu je převrácení jeho směru.

```
float globalInt =
    max(dot(norm, normalize(-light.globalDir)), 0.0);
vec3 global = globalInt * light.global *
    vec3(texture(texture_diffuse1, TexCoord));
```



Obrázek 3.2: Graf intenzity světla v závisloti na vzdálenosti

3.3.2 Bodové světlo

Intenzitu světla je možné snižovat lineárně s vzdáleností, kterou urazí. Toto řešení zajistí nižší nasvícení vzdálených objektů, a však vypadá poněkud uměle. Světla v reálném světě jsou z pravidla velmi jasná, pokud se nacházíme v jejich blízkosti. Jejich jas velice rychle klesá s narůstají vzdáleností. V určitém bodě se klesání zpomalí a přibližuje se k nule.

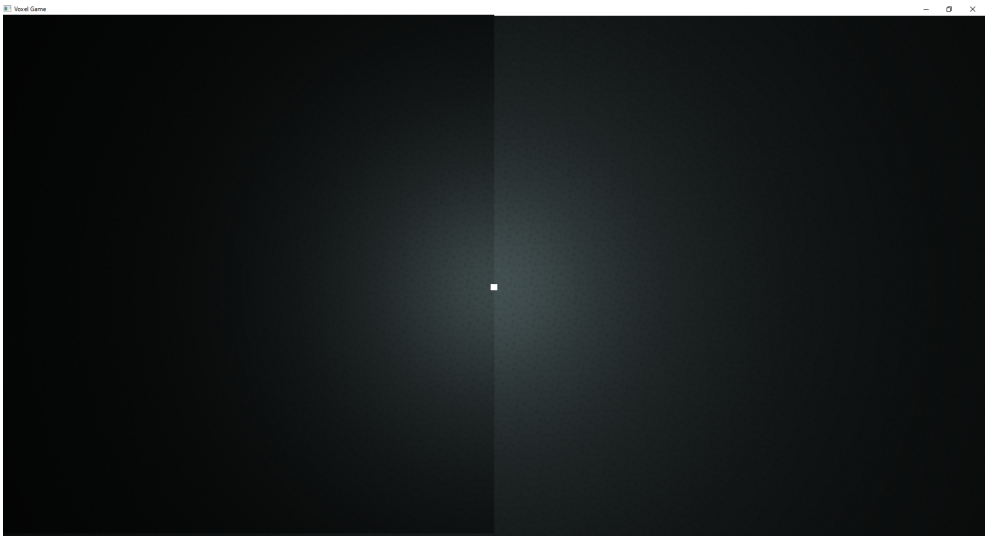
K výpočtu útlumu světla lze použít následující rovnici [28]:

$$F_{att} = \frac{1.0}{K_c + K_l \cdot d + K_q \cdot d^2}$$

Proměnná d reprezentuje vzdálenost fragmentu od zdroje světla. Dále jsou nastaveny tři konstanty:

- K_c — Konstantní složka. Obvykle je ponechána na hodnotě 1. Zajišťuje, aby hodnota jmenovatele neklesla pod 1 a nedošlo k navýšení intenzity světla.
- K_l — Lineární složka.
- K_q — Kvadratická složka.

Graf 3.2 zobrazuje porovnání hodnot intenzity světla danou rovnicí útlumu (oranžová křivka) a lineární závislostí na vzdálenosti (šedá přímka).



Obrázek 3.3: Intenzita světla v závisloti na vzdálenosti

Hodnoty parametrů jsou určeny požadovaným dosvitem světla. Parametr $K_c = 1$, $K_l = 0,045$ a $K_q = 0,0075$ [29]. Lineární rovnice má tvar:

$$F_{att} = \frac{100 - d}{100}$$

Obrázek 3.3 ve své levé části zobrazuje scénu vykreslenou pomocí rovnice útlumu. Pravá část snižuje intenzitu světla lineárně.

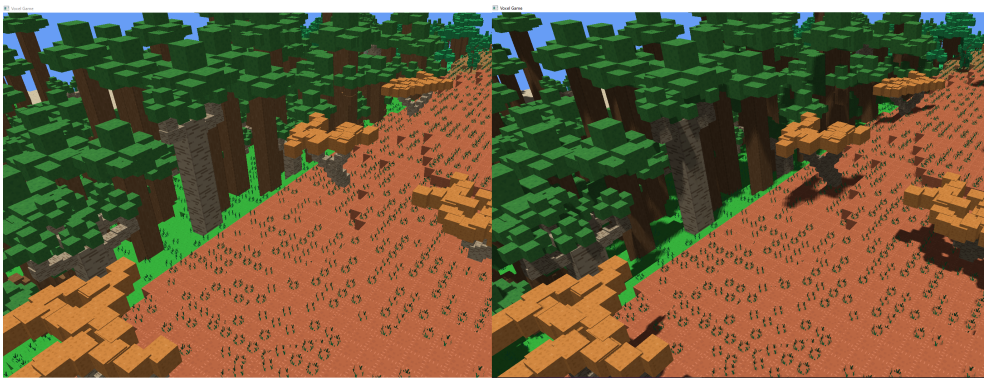
Kapitola 4

Stínování scény

V levé části obrázku 4.1 lze vidět scénu nasvícenou globálním světlem. Každý blok scény je nasvícen individuálně a ignoruje ostatní bloky, které by ho mohly zastiňovat. Tento efekt působí velice nereálně zvláště v podrostu džungle a pod širokými korunami akácií, které jsou nasvíceny plným světlem. Přidáním stínů lze docílit věrohodnější reprezentace reálného světa a hráč získá lepší představu o pozici bloků vůči sobě.

Trasování tisíců paprsků světla v reálném čase je výpočetně neúnosné. Proto je použita technika mapování stínů, využívající hloubkového bufferu grafické karty. Scéna je nejprve vykreslena z pohledu světla a hodnoty hloubky fragmentů uloženy do textury nazývané mapa stínů.

Pro druhé vykreslovací kolo, je mapa stínů předána fragment shaderu. Ten provede transformaci pozice fragmentu z prostoru světa (world space) na pozici v prostoru světla a porovná jeho hloubku s hloubkou uloženou v mapě stínů [30].



Obrázek 4.1: Porovnání scnény bez stínu a se stíny

4.1 Vykreslení mimo obrazovku

OpenGL umožňuje změnit objekt, do kterého se uloží výsledek vykreslovacího řetězce. V základním nastavení to je obrazovka hráče. Toto nastavení lze změnit vytvořením nového framebufferu a textury, do které se bude vykreslovat.

Textura může mít jiné rozlišení, než má okno, na které je vykreslována scéna. Zvětšením rozlišení se zlepší kvalita stínů, na úkor rychlosti vykreslování. Mapa stínů nemusí pokrývat veškeré objekty, které hráč vidí. Proto je textuře nastaven okraj, s maximální hloubkou. Při čtení pixelů mimo texturu se díky parametru `GL_CLAMP_TO_BORDER` vrátí maximální hloubka a vykreslovaný objekt nebude mít stín.

```
Texture_.Generate(Width, Height, nullptr);
glBindTexture(GL_TEXTURE_2D, Texture_.Id);
// set no shadow outside of the shadow map
constexpr float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(
    GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
glBindTexture(GL_TEXTURE_2D, 0);
```

Pro vykreslení je vygenerován nový framebuffer, jemuž je textura předána jako hloubková složka. Při výpočtu stínu není potřeba buffer pro barvu. Framebuffer objekt by bez něj nebyl kompletní, a proto se musí explicitně nastavit `glDrawBuffer` a `glReadBuffer` na `GL_NONE`.

```
// attach to framebuffer
glGenFramebuffers(1, &FBO_);
glBindFramebuffer(GL_FRAMEBUFFER, FBO_);
glFramebufferTexture2D(
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D, Texture_.Id, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

Před vykreslením se musí nastavit velikost zobrazovacího zařízení, na které se má scéna vykreslit. V tomto případě velikost textury mapy stínů.

```
glViewport(0, 0, Width, Height);
```

Pro vykreslení do mapy stínů je použit shader, který transformuje vertex do souřadnic světla. A předá pozici textury k dalšímu zpracování.

```
TexCoord = vec2(aTexCoord.x + shift.x, aTexCoord.y + shift.y);
gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
```

Fragment shader vyřadí všechny průhledné fragmenty a zapíše jejich hloubku.

```
float alpha = vec4(texture(texture_diffuse1, TexCoord)).a;
if (alpha < 0.05)
    discard;

gl_FragDepth = gl_FragCoord.z;
```

4.2 Vykreslení stínu

K vykreslení stínu je použit shader z kapitoly o světle. Globální složka světla je vynásobena hodnotou *(1 – shadow)*, kde *shadow = 1* znamená maximální stín a *shadow = 0* žádný stín.

```
float shadow = ShadowCalculation(fs_in.FragPos, dotLightNormal);
vec3 global =
    (1 - shadow) * globalInt * light.global *
    vec3(texture(texture_diffuse1, TexCoord));
```

Funkce `ShadowCalculation` vrátí hodnotu 1, pokud je hloubka fragmentu větší než hodnota v mapě stínů, jinak vrátí 0.

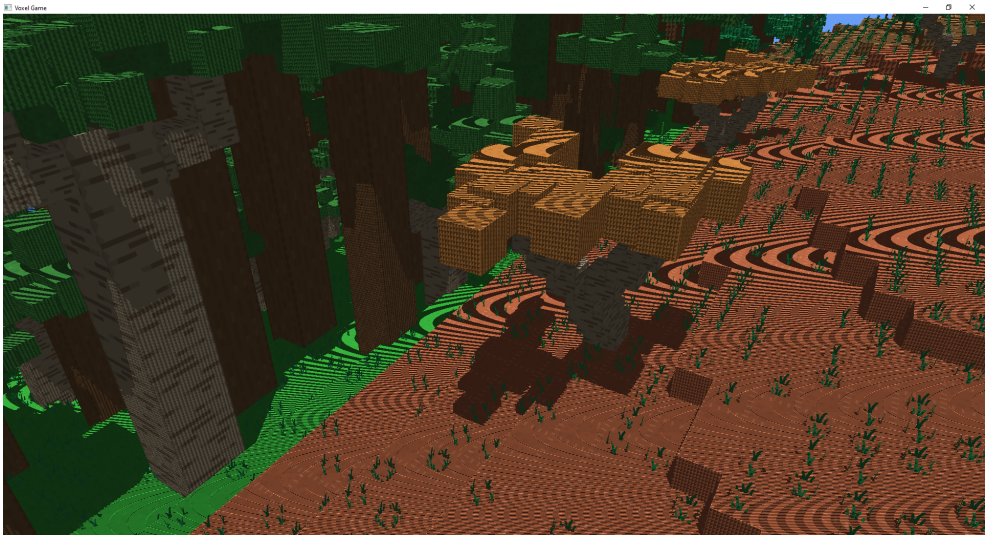
```
// perform perspective divide
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// transform to [0,1] range
projCoords = projCoords * 0.5 + 0.5;
float closestDepth = texture(shadowMap, projCoords.xy).r;
float currentDepth = projCoords.z;
return currentDepth > closestDepth  ? 1.0 : 0.0;
```

Scéna 4.2 je vykreslena s použitím toho shaderu. Stíny stromů jsou správně vykresleny, oko diváka však přitáhnou artefakty vzniklé při výpočtu stínu, nazývající se stínové akné.

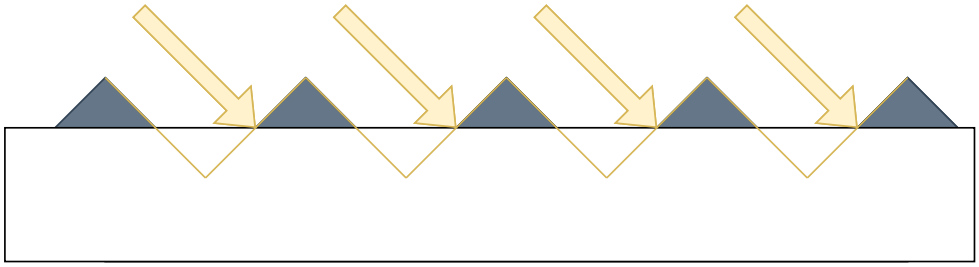
4.3 Stínové akné

Viditelné artefakty vznikají na površích, které by neměly mít stín. Problém ilustruje diagram 4.3, kde je mapa stínů promítnuta na vodorovný povrch. Každá šipka představuje jeden paprsek světla – texel mapy stínů – dopadající na povrch. Omezené rozlišení mapy stínů má za důsledek, že několik fragmentů může číst hodnotu hloubky ze stejného texelu. Fragmenty nacházející se nad žlutou křivkou nemají stín, fragmenty nacházející se pod ní stín mají.

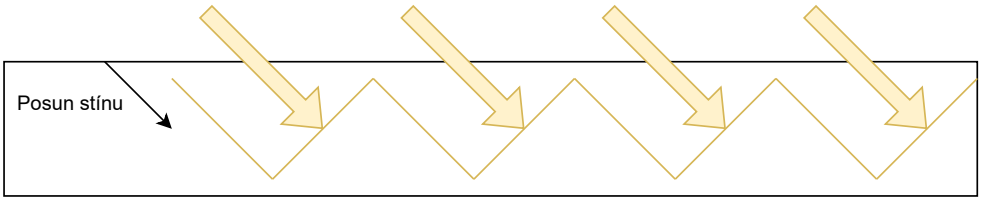
K tomuto problému dochází v případě, kdy světlo dopadá na povrch pod úhlem. V jednoduché scéně, kdy je slunce nad hlavou hráče, by k tomuto



Obrázek 4.2: Stínové akné



Obrázek 4.3: Důvod vzniku stínového akné



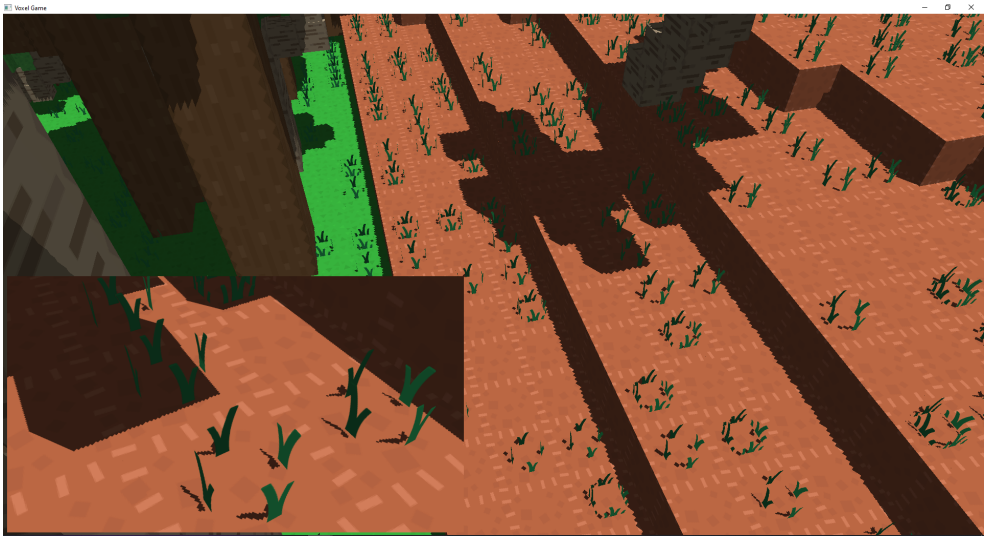
Obrázek 4.4: Zamezení vzniku stínového akné

problému nedocházelo. Takové scény nejsou vizuálně zajímavé a značně by omezovaly možnosti enginu.

Problém lze vyřešit posunutím mapy stínů (nebo povrchu objektu) tak, aby se texely mapy stínů nenacházely nad povrchem objektu, viz diagram 4.4.

Pro posun mapy lze experimentálním způsobem zvolit konstantu.

```
float bias = 0.0085;
```



Obrázek 4.5: Ostrá hrana stínu

```
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

Toto řešení funguje pro světlo dopadající na povrch pod stejným úhlem. Pokud se změní zdroj světla nebo orientace objektu, nemusí být konstanta dostatečně velká a problém se bude opakovat. Robustnějším řešením je vypočítat odchylku – `bias` – podle úhlu dopadu světla, který bude největší pro světlo dopadající na povrch pod ostrým úhlem a nejmenší pro světlo dopadající kolmo.

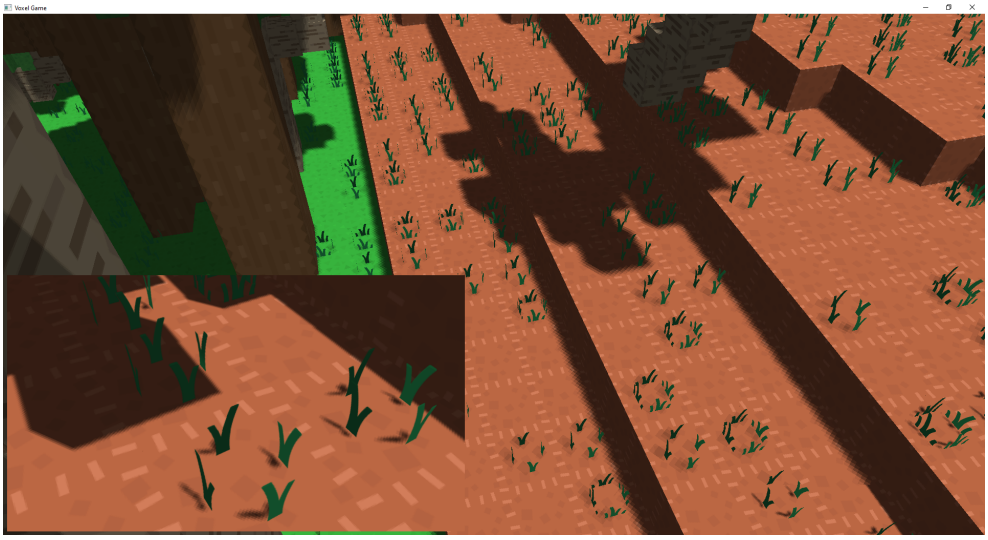
```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
```

4.4 Ostré stíny

Při pohledu na scénu 4.5 lze odhalit nedostatečné rozlišení mapy stínů. Několik vykreslovaných fragmentů se namapuje na jeden texel z mapy stínů. Výsledkem jsou zubaté hrany stínu s velmi ostrým přechodem. Řešení, které nevyžaduje zvětšení rozlišení, je procentuálně bližší filtrování (percentage–closer filtering – PCF).

Technika PCF produkuje jemnější stíny. Ty se jeví méně hranaté a zubatý efekt není tak výrazný. PCF zahrnuje více způsobu filtrování, jehož základem je vícenásobné čtení vzorků z textury stínu a jejich zprůměrování. Jednoduchá technika je čtení hodnot ve čtverci kolem zpracovávaného texelu.

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(texture_shadow, 0);
for(int x = -1; x <= 1; ++x) {
    for(int y = -1; y <= 1; ++y)  {
```



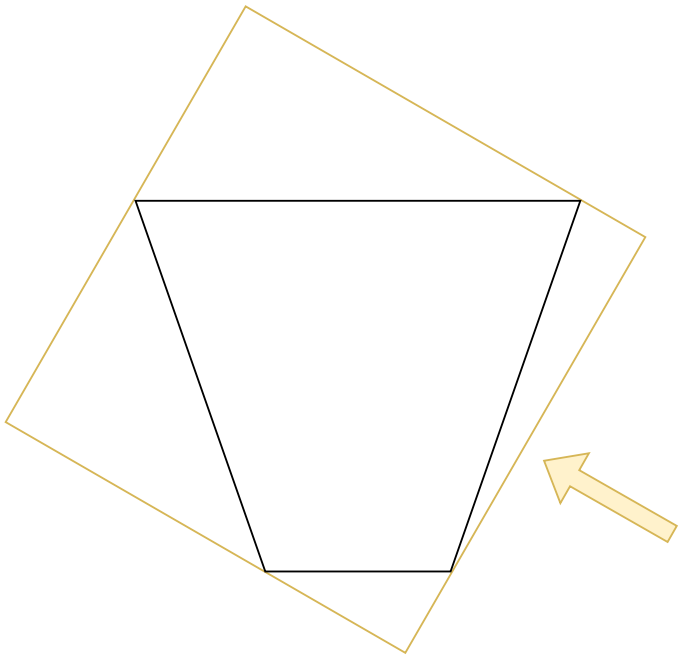
Obrázek 4.6: Měkká hrana stínu

```
            float pcfDepth = texture(
                texture_shadow,
                projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;
```

Scéna 4.6 je vykreslena při použití 9 vzorků. Při pohledu z dálky lze pozorovat lepší výsledky, zuby stínů nejsou tolik patrné, jako při použití jednoho vzorku. Při bližším pohledu na stébla trávy, lze pozorovat rozšíření stínu o jeden texel. Štíhlé textury již nevrhají přesné stíny, výměnou za zlepšení pohledu z dálky, kdy byl vrhaný stín nepřesný (velikost texelu neodpovídala velikosti textury) a byla vykreslena pouze jeho část s velice ostrým přechodem.

4.5 Pozice mapy stínů na scéně

Mapa stínů musí být posouvána podle pozice kamery v herním světě. Objekty jsou generovány kolem postavy hráče a pokud se hráč posune od počátku světa, objekty budou smazány a nemohou vrhat stíny. Mapa stínů má však omezené rozlišení a stínováním objektů, které jsou vygenerované, ale hráč je nemá šanci vidět (jsou například za ním), se snižuje kvalita stínů. Pro maximální využití dostupného rozlišení stačí pokrýt pouze objekty, které může hráč v daném snímku vidět. Viditelná část světa je určena maticí pohledu (view) a maticí projekce (projection). Tyto matice tvoří komolý jehlan, ve



Obrázek 4.7: Frustum světla okolo frusta kamery

kterém budou objekty viditelné. Tento komolý jehlan je potřeba obsáhnout frustem (frustum může tvořit komolý jehlan nebo kvádr, podle typu projekce) světla s co nejmenšími přesahy. Tato skutečnost ve 2D světě, je ilustrována na obrázku 4.7.

4.5.1 Souřadnice frusta kamery

Matice pohledu a projekce transformují souřadnice světa na normalizované souřadnice zařízení (NDC – normalized device coordinates). Každá souřadnice na jedné ze tří os NDC nabývá hodnot $[-1, 1]$. Do souřadnic světa je lze převést vynásobením inverzí matic pohledu a projekce [31].

$$v_{NDC} = M_{proj} \cdot M_{view} \cdot v_{world}$$
$$v_{world} = (M_{proj} \cdot M_{view})^{-1} \cdot v_{NDC}$$

Pro vytvoření frusta světla, je nutné zjistit souřadnice rohů frusta kamery. Díky nim je možné přesně zjistit jakou část světa má frustum světla pokrývat. Souřadnice jsou vypočítané následujícím způsobem:

```
std::vector<glm::vec4> Helpers::Math::FrustumCornersWordSpace(
  const glm::mat4& proj, const glm::mat4& view) {
    const auto inv = glm::inverse(proj * view);
```

```
    std::vector<glm::vec4> frustumCorners;
    for (unsigned int x = 0; x < 2; ++x) {
        for (unsigned int y = 0; y < 2; ++y) {
            for (unsigned int z = 0; z < 2; ++z) {
                const glm::vec4 pt =
                    inv * glm::vec4(
                        2.0f * x - 1.0f, 2.0f * y - 1.0f,
                        2.0f * z - 1.0f, 1.0f);
                frustumCorners.emplace_back(pt / pt.w);
    }}}

    return frustumCorners;
}
```

4.5.2 Matice pohledu a projekce světla

Pro výpočet matice pohledu světla je potřeba zjistit bod, který je ve středu stínované oblasti. Tento bod se nachází ve středu frusta kamery. A je ho možné získat zprůměrováním pozic rohů frusta.

```
glm::vec3 Helpers::Math::FrustumCenter(
  const std::vector<glm::vec4>& corners) {
    auto center = glm::vec3(0.0f);
    for (const auto& v : corners) {
        center += glm::vec3(v);
    }
    center /= corners.size();

    return center;
}
```

Matice pohledu je získána pomocí funkce `glm::lookAt` a směru světla.

```
const auto lightView = glm::lookAt(
    center,
    center + lightDir,
    glm::vec3(0.0f, 1.0f, 0.0f));
```

Pro stínování scény je použito globální světlo. K výpočtu matice projekce bude tedy použita funkce glm::ortho zaručující ortografickou projekci. Pro výpočet je nutné zjistit parametry `left`, `right`, `bottom`, `top`, `zNear` a `zFar`.

Při pohledu od zdroje světla bude jeho frustum osově zarovnaný kvádr, těsně objímající frustum kamery. Rohy kamery lze pomocí matice pohledu světla transformovat do souřadnicového systému pohledu světla. Z těchto

transformovaných rohů je vybráno maximum a minimum v každé ose, definující frustum světla.

```
for (const auto& v : corners) {
    const auto trf = lightView * v;
    minX = std::min(minX, trf.x);
    maxX = std::max(maxX, trf.x);
    minY = std::min(minY, trf.y);
    maxY = std::max(maxY, trf.y);
    minZ = std::min(minZ, trf.z);
    maxZ = std::max(maxZ, trf.z);
}
```

Před vytvořením matice projekce je potřeba upravit proměnnou `minZ` a `maxZ`, představující blízkou a vzdálenou plochu frusta. S aktuální hodnotou by stíny vrhaly pouze objekty viditelné hráčem. Například stín koruny stromu by se objevil pouze, pokud by ji hráč viděl.

Posunuté hranice frustra by měly pokrývat všechny objekty na scéně. Příliš velkorysým posunutím však dojde ke ztrátě přesnosti mapy stínů. Hranice by tedy měly být posunuté podle aktuální scény, například aby byl vykreslen stín pod nejvyšším stromem, u jehož paty hráč stojí.

```
if (minZ < 0)
    minZ *= zMult;
else
    minZ /= zMult;
if (maxZ < 0)
    maxZ /= zMult;
else
    maxZ *= zMult;

const glm::mat4 lightProjection =
    glm::ortho(minX, maxX, minY, maxY, minZ, maxZ);

return lightProjection * lightView;
```

4.6 Kaskádové mapování stínů

Jednoduché mapování stínů má podstatnou nevýhodu. Pokud chceme rozšířit oblast pokrytou stíny, musíme zvýšit rozlišení mapy stínů. V opačném případě by stíny blízko hráče byly rozkostičkované. Tímto zvyšováním kvality stínů je masivně zatěžováno GPU. Stíny, které jsou vzdálenější od hráče, budou mít stejnou kvalitu, jako ty mu blízké. Toto je zbytečné, vzdálenější stíny mohou mít horší kvalitu – hráč bude rozlišovat pouze jejich tvar a existenci, drobné

detaily jsou zbytečné. Tento problém řeší kaskádové mapování stínů, skládající se z následujících kroků [31].

1. Rozděl frustum kamery na n subfrust, kde vzdálená plocha frusta i je blízká plocha frusta $i + 1$.

2. Pro každé frustum spočítej matici prostoru světla.

3. Vykresli mapu stínů pro každé frustum.

4. Předej všechny mapy stínu fragment shaderu.

5. Vykresli scénu, kde podle vzdálenosti fragmentu vybereš patřičnou mapu stínů.

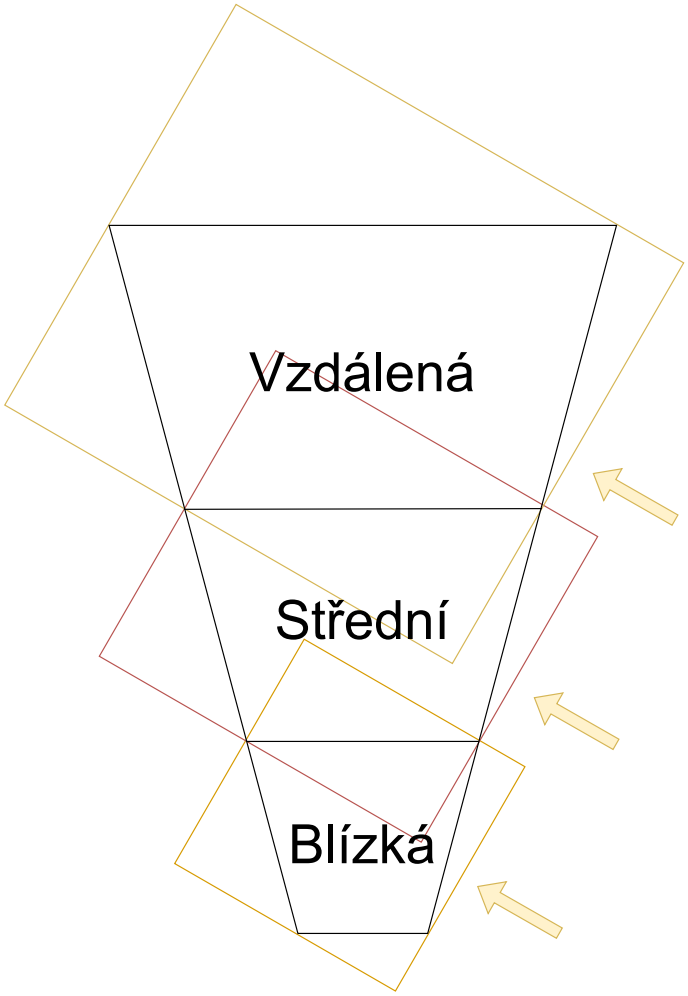
Na obrázku 4.8 je zobrazeno rozdělení frusta kamery na tři části. Velikost částí se zvětšuje s narůstající vzdáleností od blízké plochy kamery. Rozlišení však zůstává stejné, a proto jeden texel mapy stínů bude pokrývat větší plochu světa. Kvalita stínů se proto bude zhoršovat s rostoucí vzdáleností od hráče. Frustum lze rozdělit na libovolný počet částí. S větším počtem dělení, se snižuje viditelnost změny kaskády.

4.6.1 Mapy stínů

Pro práci s více texturami, mající stejný rozměr, OpenGL poskytuje pole 2D textur [32]. Práce s polem textur usnadní práci (navázání pouze jedné textury) a umožní dynamické měnění počtu textur. Vytvoření 3D textury [33] probíhá obdobně jako vytvoření 2D textury. Jedním z rozdílů je zadání počtu textur, pro alokaci paměti grafické karty.

K vykreslení do pole textur se používá technika vrstveného vykreslování. V ní je za pomoci geometry shaderu vytvořena nová geometrie pro každou vrstvu map stínů. Pro každou matici světla z kaskády je vytvořen nový trojúhelník, který je pomocí ní transformován, a nastaveno číslo vrstvy.

```
void main() {
    for (int i = 0; i < 3; ++i) {
        gl_Position = lightSpaceMatrices[gl_InvocationID] *
                      gl_in[i].gl_Position;
        gl_Layer = gl_InvocationID;
        TexCoord = gs_in[i].TexCoord;
        EmitVertex();
    }
    EndPrimitive();
}
```



Obrázek 4.8: Rozdělení frusta kamery na tři části

Pro vykreslení stínů je použit shader vykreslující mapu stínů s úpravami umožňujícími práci s mapami stínů. Pro zvolení správné matice světla je vypočítána hodnota hloubky fragmentu (v prostoru pohledu kamery), která je následně porovnána se vzdáleností nejbližší změny kaskády.

```
vec4 fragPosViewSpace = view * vec4(fragPosWorldSpace, 1.0);
float depthValue = abs(fragPosViewSpace.z);

int layer = CASCADE_COUNT - 1;
for (int i = 0; i < CASCADE_COUNT - 1; ++i) {
    if (depthValue < cascadePlaneDistances[i]) {
        layer = i;
        break;
```

Obrázek 4.9: Velikost stínového akné v závislosti na kaskádě

```
    }
}
vec4 fragPosLightSpace =
    lightSpaceMatrices[layer] * vec4(fragPosWorldSpace, 1.0);
```

Na obrázku 4.9 lze vidět tři kaskády. V první nedochází k stínovému akné, v druhé je již při bližším pozorování patrné a ve třetí je velice výrazné. K tomuto efektu dochází, protože texel v každé vrstvě pokrývá různě velkou plochu vykreslované geometrie. Proto je potřeba volit odchylku na základě vrstvy. Škálování odchylky inverzní hodnoty vzdálené plochy frusta, produkuje výsledky bez stínového akné.

```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
if (layer == cascadeCount) {
    bias /= farPlane * 0.008;
}
else {
    bias /= cascadePlaneDistances[layer] * 0.02;
}
```

Tato technika zavádí větvení, které v shaderech není chtěné. Zároveň neumožnuje dokonalé vylazení odchylky pro jednotlivé kaskády. Odchylka je pevně svázaná s počtem a vzdáleností kaskád. Její výpočet proto může být přesunut na CPU, k definici kaskád. Odchylky jsou předány shaderu pomocí pole a načteny následujícím způsobem.

```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
bias *= cascadeBiases[layer];
```

4.6.2 Změna počtu kaskád

Engine umožňuje měnit počet kaskád při běhu programu. Shadery proto musí mít alokované dostatečně velké místo, pro uložení všech matic. Jedinou výjimkou je geometry shader, ve kterém je specifikován počet vyvolání.

```
#define CASCADE_COUNT 3
layout(triangles, invocations = CASCADE_COUNT) in;
layout(triangle_strip, max_vertices = 3) out;
```

Pokud je `CASCADE_COUNT` menší, než počet kaskád n. $n-1$ nejvzdálenějších kaskád nebude vykresleno. Pokud je větší, bude plýtván výpočetní čas GPU. Počet kaskád shaderu nemůže být předán parametrem – vyžaduje běh shaderu. Engine proto umožňuje znovu zkompilovat shadery za běhu a upravit hodnotu maker. Zdrojový kód je po načtení z disku projit a všechna makra nahrazena specifikovanou hodnotou, kód je následně přeložen. Opakovaný překlad je proveden při změně počtu kaskád.

```
ShaderDepth_ = ResourceManager::SetShaderMacros(
  "shadow_csm", {{ "CASCADE_COUNT", std::to_string(levels) }});
```

Makra jsou použita i ve zbylých shaderech, umožňující alokaci přesného počtu prvků v poli matic světla, kaskád, atd.

4.7 Optimalizace vykreslování

Po zapnutí kaskádového stínování došlo k viditelnému poklesu FPS. K měření výkonu a následné optimalizaci byla vytvořena třída `GpuTimer`, umožňující měřit délku operací probíhajících na GPU. Požadavky pro vykreslení na grafické kartě jsou asynchronní. Synchronizace CPU a GPU ale, probíhá jen v určitých bodech programu, např. výměna bufferů. Pro měření jednotlivých vykreslovacích volání, proto musí být do CPU kódu umístěna synchronizační bariéra – aktivní čekání. Od verze OpenGL 3.3 je možné využít dotazy na dobu běhu [34]. Vygenerování dotazů probíhá zavoláním funkce:

```
glGenQueries(2, QueryId_);
```

Aktuální hodnota časovače je zjištěna dotazem:

```
glQueryCounter(QueryId_[0], GL_TIMESTAMP);
```

Čas je zaznamenán po doběhnutí všech předchozích volání [35]. Před vyčtením hodnoty časovače musí CPU aktivně čekat, dokud není výsledek dotazu k dispozici. A následně výsledky dotazu načíst.

```
glQueryCounter(QueryId_[1], GL_TIMESTAMP);

int available = 0;
while (!available) {
    glGetQueryObjectiv(
        QueryId_[1], GL_QUERY_RESULT_AVAILABLE, &available);
}
long long start, stop;

glGetQueryObjecti64v(QueryId_[0], GL_QUERY_RESULT, &start);
glGetQueryObjecti64v(QueryId_[1], GL_QUERY_RESULT, &stop);

return stop - start;
```

Všechna následující měření probíhala po načtení totožné scény, s kamerou směřující stejným směrem, zprůměrováním prvních 1000 hodnot časovače.

4.7.1 Průhledné textury

Pro správné vykreslení stínů průhledných textur musí fragment shader pracovat s texturou objektu. Načíst zní hodnotu průhlednosti. A zapsat do hloubkového bufferu pouze, pokud je fragment neprůhledný.

```
float alpha = vec4(texture(texture_diffuse1, TexCoord)).a;
if (alpha < 0.05)
    discard;

gl_FragDepth = gl_FragCoord.z;
```

Čtením z textury u fragmentu, který nemá průhlednost je ztrácen výkon. Další optimalizace, které nemůže být použita, je brzký test hloubky (early depth test). Grafická karta může provést test hloubky před spuštěním fragment shaderu a vyřadit fragmenty, které nebudou viditelné. Použitím klíčového slova `discard` a zapsáním do hloubkového bufferu je tento test vypnut [36].

Pro změření výkonu byl vytvořen prázdný fragment shader s explicitně zapnutým brzkým testem hloubky.

```
layout(early_fragment_tests) in;

void main() {}
```

Vykreslení mapy stínů s kontrolou průhlednosti trvalo v průměru o 33,67 % déle, než když byl na všechny objekty použit prázdný shader – optimalizace přináší očekávané výsledky. Při použití obou shaderů, pro průhledné a neprůhledné objekty, došlo k zhoršení času o 4,54 %. Tento přístup však produkuje korektní stíny a díky nízkému procentu průhledných objektů na scéně (celkový počet objektů byl 159 021 z toho 22 613 průhledných), přináší významné zrychlení.

Porovnání výkonu bylo prováděno na jedné scéně, kde průhledné objekty tvořily 14,22 % z celkového počtu objektů. V dalším testu byl proto měřen průměrný poměr průhledných a neprůhledných objektů na scéně. Vykreslovací vzdálenost byla nastavena na 15 chunků a kamerou bylo pohybováno severním směrem, tak aby docházelo ke střídání teplotních pásů a biotopů (viz sekce 6.3.3). Celkem bylo naměřeno 2500 záznamů. Průhledné objekty tvořily průměrně 14,67 % z celkového počtu objektů, s výběrovou směrodatnou odchylkou 2,17 %. Z tohoto měření lze předpokládat, že optimalizace bude přinášet obdobné výsledky i na jiných scénách.

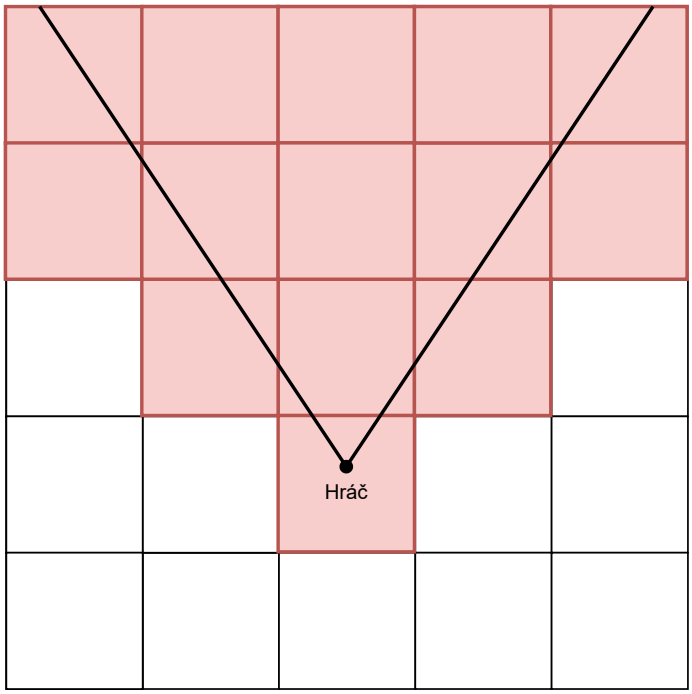
4.7.2 Ořezání scény

Doposud byly grafické kartě předávány všechny objekty na scéně. Ztráta výkonu byla omezena díky ořezávacím testům, probíhajícím před spuštěním výpočetně náročného fragment shaderu. Testy z vykreslovacího řetězce vyřadí všechny fragmenty, které se nacházejí mimo prostor obrazovky.

Při vykreslování map stínů je použit geometry shader, generující nová primitiva pro každou kaskádu. Tyto primitiva nejsou automaticky ořezána grafickou kartou, jako v případě fragment shaderu. Dochází tedy ke generování primitiv, které hráč nemůže vidět a ani nebudou vrhat stín na viditelnou plochu.

Engine proto musí provést ořezání objektů před jejich předáním grafické kartě. Testování je prováděno na úrovni chunků. Pokud je část chunku viditelná hráčem, pak je uložen ukazatel na jeho obsah, který je následně předán grafické kartě. Situaci ilustruje obrázek 4.10, kde jsou viditelné chunky zvýrazněny.

Při testu viditelnosti je každý roh chunku transformován do prostoru kamery. Pokud je bod viditelný, jeho souřadnice na ose x musí být z intervalu $[-1, 1]$. Jeho souřadnice na ose z musí být menší než 1 (nacházet se před blízkou plochou kamery). Souřadnice na ose y nejsou kontrolovány, protože roh chunku se může nacházet pod spodní rovinou vymezující frustum, ale jeho objekty budou přesto viditelné – například stromy. Kontrola zda je $y < 1$ by způsobovala mizení chunků při sklonění kamery.



Obrázek 4.10: Ořezání neviditelných chunků

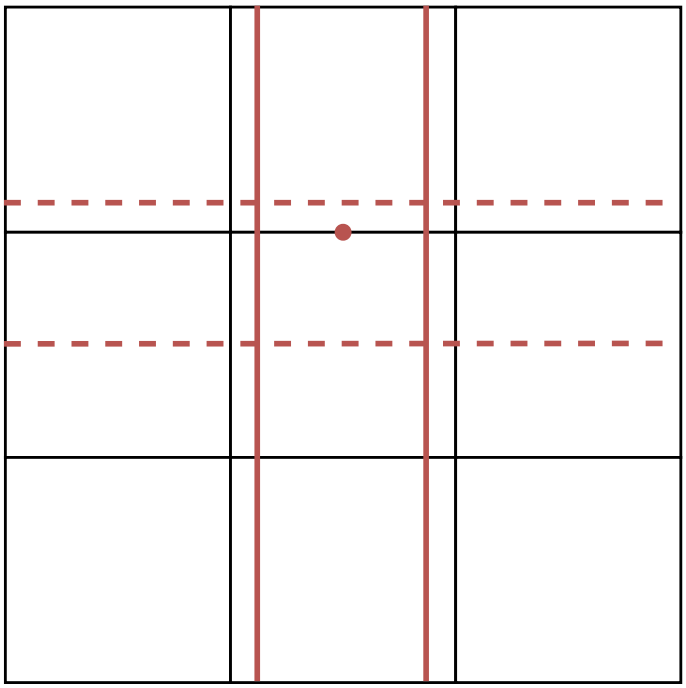
```
bool Scene::IsPointInView(
  const glm::vec3& position, const glm::mat4& projView) {
    auto pt = projView * glm::vec4(position, 1.0f);
    pt /= pt.w;

    return pt.x >= -1.0f && pt.x <= 1.0f && pt.z <= 1.0f;
}
```

Při pouhé kontrole rohů by mohlo dojít k ořezání chunků, jež se nachází před hráčem, ale žádný jejich roh není viditelný. K tomu může dojít, pokud má hráč úzké zorné pole, nebo se dívá směrem k zemi pod dostatečným úhlem. Situace, kdy se hráč dívá kolmo k zemi je ilustrována na obrázku 4.11. Viditelná část herní plochy je vymezena červenými přímkami. Ke kontrole je předán bod, nacházející se na hranici chunků – vždy je vybrána hranice bližší k hráči – a pozici x rovné pozici hráče na ose x, respektive ose z, pokud je hranice rovnoběžná s osou z.

Pozice rohu na ose y je nejnižší bod v chunku. Pokud se hráč podívá vzhůru, všechny rohy v chunku budou oříznuty blízkou plochou kamery. Z tohoto důvodu je omezen maximální náklon kamery, který je použit při výpočtu matice pohledu. Náklon je oříznut na interval $[-90°, 0°]$ vůči vodorovné ploše.

Ořezáním chunků scény byl omezen počet objektů, předávaných grafické



Obrázek 4.11: Ořezání chunků při pohledu kolmo dolů

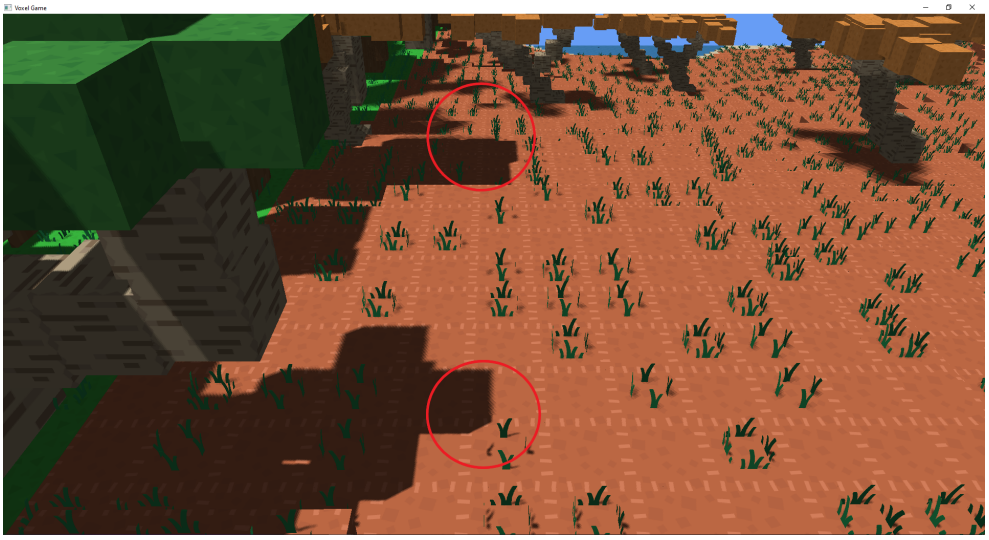
kartě ze 159 021 na 43 605. Díky tomu se 3,4x snížil čas potřebný k vykreslení mapy stínů.

4.8 Výsledky

Scéna na obrázku 4.12 je vykreslena pomocí kaskádového mapování stínů. Lze si všimnout dvou změn kaskád a zhoršení kvality stínů[7]. Rozlišení kaskád je nastaveno na 1024 x 1024 pixelů. S tímto rozlišením je možné pokrýt celou herní scénu v dostatečné kvalitě. Pro pokrytí celé scény v uspokojivé kvalitě, bylo potřeba zvýšit rozlišení jednoduché mapy stínů alespoň na 4096 x 4096 pixelů. Ztrátu kvality však bylo stále možné pozorovat při přiblížení se k objektu.

Měření ukázala, že použitím CSM, nedochází k vysoké ztrátě výkonu. Vykreslení jednoho snímku trvalo o 7,39 % déle než při použití jednoduché mapy. Kaskádové mapování lze zároveň lépe škálovat. Přidáním dvou kaskád se výrazně zvýšila kvalita stínů v blízkosti hráče a vykreslení snímku trvalo o 10,18 % déle. Zvýšením rozlišení jednoduché mapy na 8192 x 8192 pixelů, nedošlo k dostatečnému zlepšení stínů v blízkosti hráče. Vykreslení snímku

[7]Kaskády byly pro viditelnější efekt posunuty k blízké ploše frusta kamery.



Obrázek 4.12: Výsledek použití kaskádového mapování stínů

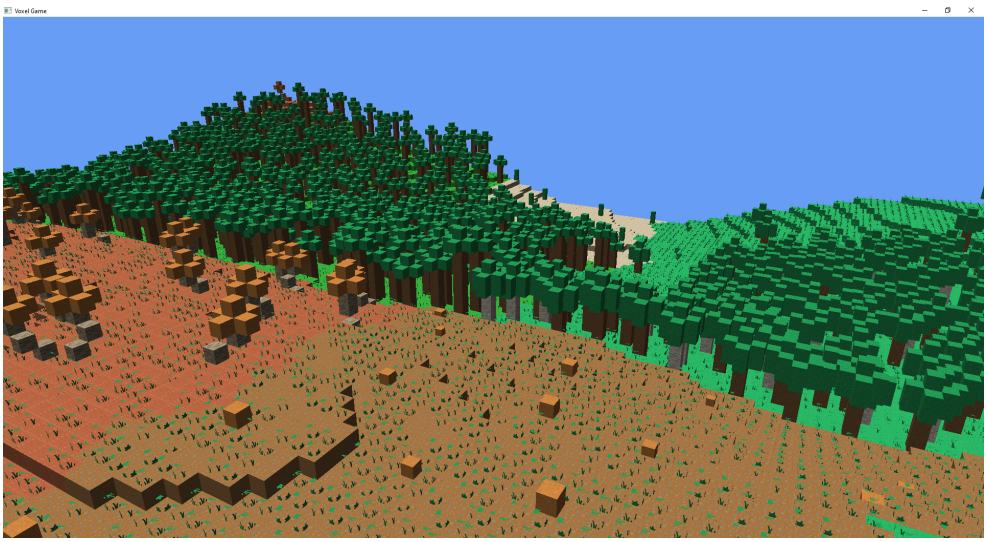
trvalo o 63,88 % déle než při použití pěti kaskád. Použití CSM se tedy jeví jako lepší řešení.

Kapitola 5

Modelování rostlin s využitím L-systémů

Herní svět obsahuje velké množství vegetace a ač jsou si všechny stromy, keře typově podobné, hráč si velice rychle všimne, že jsou identické. Na obrázku 5.1 vidíme tři druhy stromů, které jsou zkopírované po scéně. Stromy v levé zadní části mají různou výšku, přesto působí umělým dojmem.

Stromy v reálném světě jsou si podobné – rozeznáváme jednotlivé druhy stromů, přesto neexistují dva stejné stromy. Pokud druh stromu zapíšeme formální gramatikou nazývanou L-systém, docílíme podobné struktury stromů, které se budou lišit v detailech.



Obrázek 5.1: Identické stromy

5.1 L-systém

L-systém nebo také Lindenmayerův systém je paralelní přepisovací systém vyvinutý maďarským teoretickým biologem a botanistou Aristidem Lindenmayerem v roce 1968. L-systém je typ formální gramatiky skládající se z abecedy, přepisovacích pravidel a počátečního axiomu. Pomocí postupného derivování počátečního axiomu je možné simulovat vývoj rostliny v čase [37].

5.2 Interpretace řetězců pomocí želvy

Řetězce lze graficky reprezentovat pomocí želvy [38], konzumující symboly abecedy. Každý symbol určuje akci, kterou má želva vykonat. Želva se může pohybovat ve 2D nebo 3D prostoru. Ve 2D si můžeme interpretaci představit jako želvu, držící tužku, pohybující se po papíře.

Želvu lze reprezentovat jako trojici (x, y, α), kde (x, y) představuje kartézské souřadnice reprezentující polohu v prostoru a α úhel kam želva směřuje. Zadáním délky kroku d a změny úhlu δ lze želvu ovládat pomocí následujících symbolů.

- F — Posun dopředu o délku d. Stav želvy se změní na (x', y', α), kde $x = x + d\cos\alpha$ a $y = y + d\sin\alpha$. Mezi body (x, y) a (x', y') je nakreslena čára.
- $+$ — Rotace doleva o úhel δ. Nový stav želvy $(x, y, \alpha + \delta)$.
- - — Rotace doprava o úhel δ. Nový stav želvy $(x, y, \alpha - \delta)$.

Nechť je definován následující L-systém. Buď ω počáteční axiom, p přepisovací pravidlo, $\delta = 90°$ a d zmenšené čtyřnásobně pro každý obrázek [39].

$\omega : F - F - F - F$

$p : F \to F - F + F + FF - F - F + F$

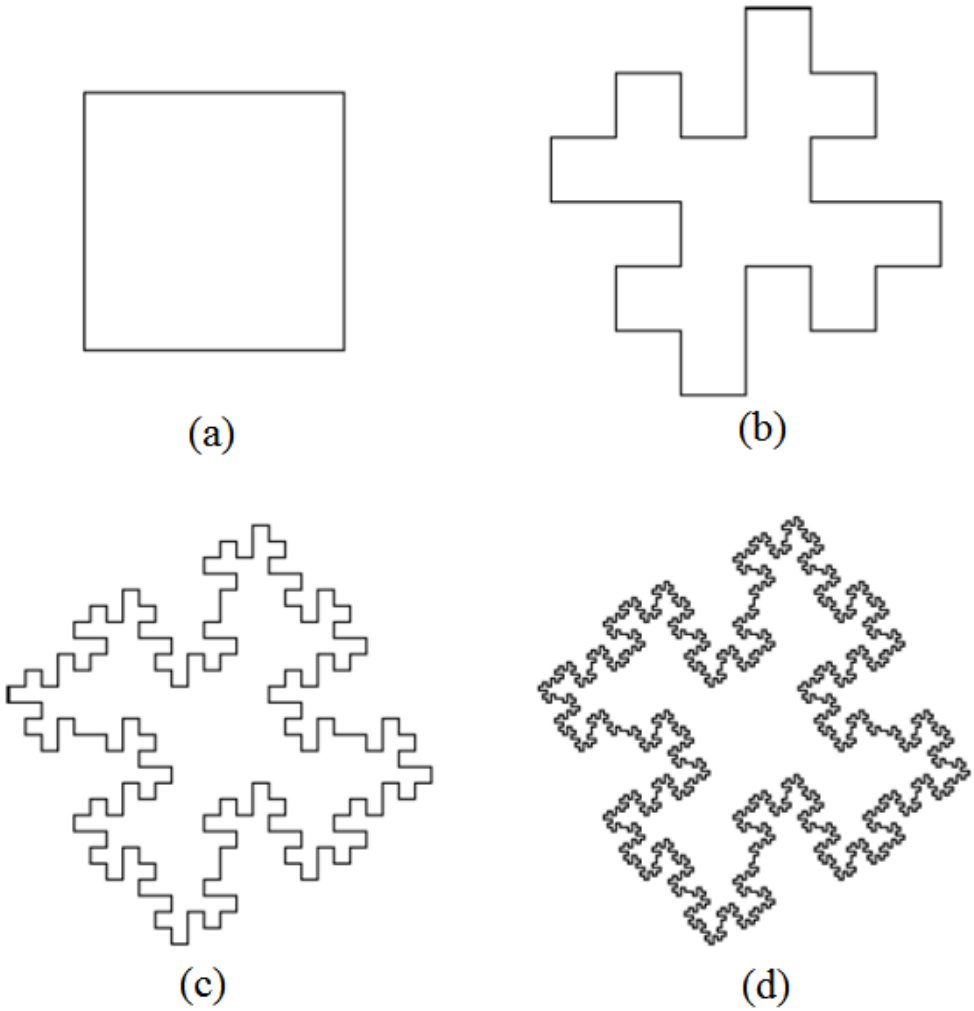
Želva interpretující daný L-systém generuje kvadratické Kochovy ostrovy 5.2. Obrázky jsou vygenerovány derivacemi o délce 0 až 3.

5.3 Větvení v L-systémech

S danými přepisovacími pravidly není možné generovat větvící se struktury. Želva vždy pokračuje od své poslední pozice. Říše rostlin je dominovaná větvícími se strukturami, potřebujeme proto matematické vyjádření této skutečnosti.

Větvení v řetězci můžeme reprezentovat pomocí dvou symbolů [a], kde [značí začátek větve a] konec větve [40].

Symboly jsou interpretovány želvou následovně:



Obrázek 5.2: Kvadratické Kochovy ostrovy

- [– Ulož atributy želvy do zásobníku.
-] – Načti atributy želvy ze zásobníku a smaž je z vrcholu zásobníku (operace pop). Při této operaci není nakreslená žádná čára.

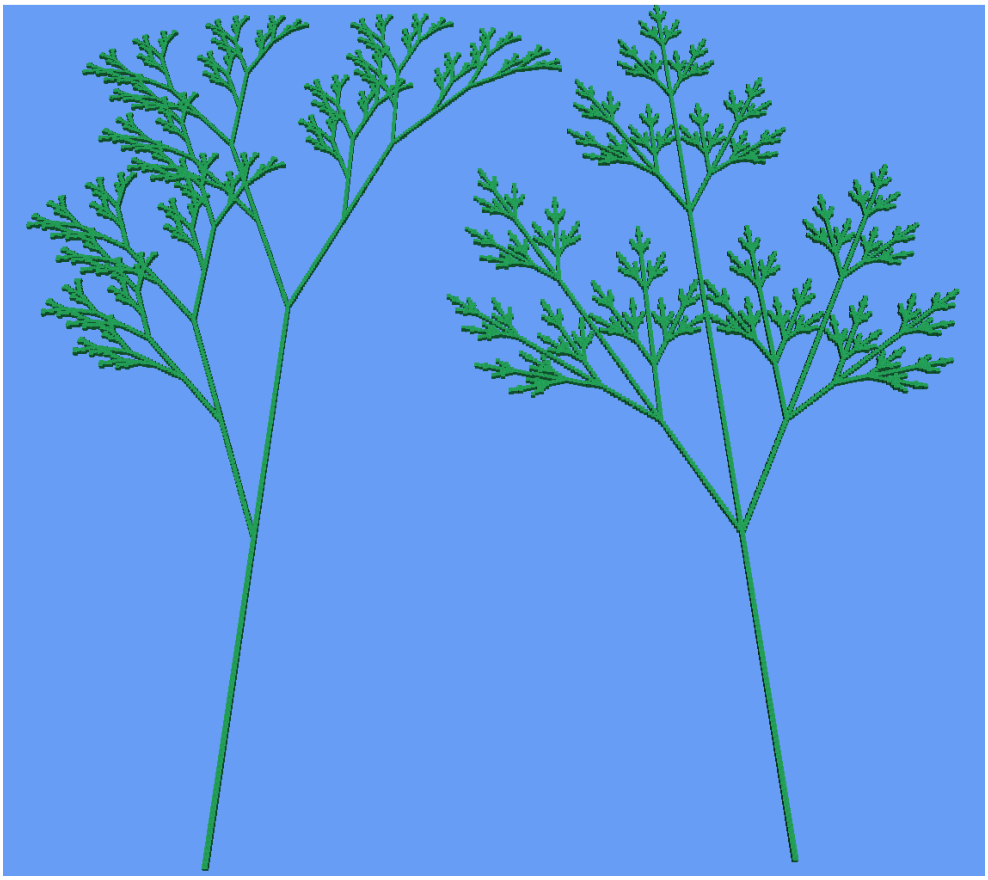
Díky nově přidaným symbolům lze generovat struktury připomínající rostliny. Struktury na obrázku 5.3 jsou generované následujícími L-systémy:

1. $\delta = 20°$

   $\omega : E$

   $p1 : F \rightarrow FF$

   $p2 : E \rightarrow F[+E]F[-E] + E$



Obrázek 5.3: Struktury připomínající rostliny generované pomocí závorkovaného systému

2. $\delta = 25,7°$

   $\omega : E$

   $p1 : F \rightarrow FF$

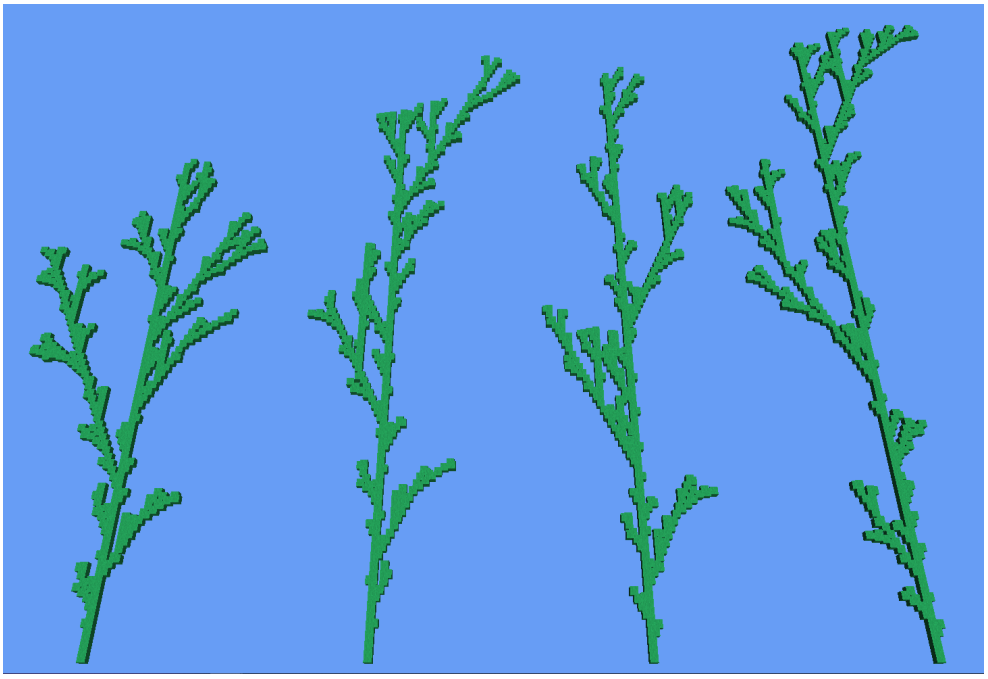
   $p2 : E \rightarrow F[+E][-E]FE$

L-systém 1 generuje rostlinu vlevo, L-systém 2 generuje rostlinu vpravo.

5.4 Stochastické L-systémy

Rostliny generované deterministickým L-systémem jsou všechny stejné. Jejich použití ve scéně by vytvářelo stejný efekt, který je popsán na začátku kapitoly.

K předejití tohoto efektu je nutné zavést variace v rámci druhu. Náhodná interpretace řetězce má limitované využití. Změna úhlu větvení, šířky a výšky



Obrázek 5.4: Využití stochastického L-systému

segmentů rostliny zachovávají topologii struktury, ze které je generovaná. Stochastické L-systémy mohou měnit topologii struktury [41].

L-systém, který byl do teď používán, nemohl mít více přepisovacích pravidel pro stejný symbol abecedy. Pokud má stochastický L-systém více přepisovacích pravidel, je z nich vybráno jedno s pravděpodobností $1/n$, kde n je počet přepisovacích pravidel pro daný symbol abecedy[8].

Scéna 5.4 byla vygenerována za pomocí stochastického L-systému, kde:

$\delta = 25,7°$

$\omega : F$

$p1 : F \rightarrow F[+F]F[-F]F$

$p2 : F \rightarrow F[+F]F$

$p3 : F \rightarrow F[-F]F$

[8]Tato definice se liší, od definice uvedené ve [41]. Tento způsob náhodného výběru je použit v implementaci, kde pravděpodobnostní distribuci zastupuje několikanásobné zopakování přepisovacího pravidla.

5.5 Implementace

5.5.1 Formát L-systému

L-systém může být načten ze souboru pomocí třídy `LSystemParser`. L-systém musí mít následující formát:

```
yaw_angle pitch_angle shring_ratio
axiom
letter > production
.
.
.
letter > production
```

Soubor může obsahovat za sebou jdoucí L-systémy. `LSystemParser` je vrátí jako pole. Soubor může obsahovat komentáře na nových řádcích, začínající symbolem #.

Třída `LSystem` obsahuje gramatiku a tři atributy specifikující úhel náklonu podle osy y (yaw), podle osy x (pitch) a změnu velikosti bloku. Poslední atribut je využit při rozvětvení rostliny. Potomci mateřské větve by se měly řídit postulátem Leonarda da Vinci: „Všechny větve stromu, v každé úrovni jeho růstu, jsou v součtu jejich tloušťky rovné tloušťce kmene pod nimi.“ V případě dvojitého rozvětvení, tloušťky mateřské větve w_1, tloušťky potomků w_2 dostaneme rovnici [42]:

$$w_1^2 = 2w_2^2$$

$$\frac{w_2}{w_1} = \frac{1}{\sqrt{2}} \approx 0,707$$

Hodnotu 0,7 je možné nalézt v L-systémech modelující keře.

5.5.2 Želva

Třída `Turtle` rozšiřuje pohyb želvy – popsané v kapitole Interpretace řetězců pomocí želvy – do 3D prostoru. Vnitřní stav želvy určují následující atributy:

- Pozice v prostoru.
- Velikost bloku vytvořeného želvou.
- Barva použitá pro kreslení (výstupní pole).
- Yaw — rotace podle osy y.
- Pitch — rotace podle osy x.

Želva si udržuje tři navzájem kolmé směrové vektory (nahoru, dopředu, doprava) jednotkové délky, které využívá pro pohyb po scéně. Vektory jsou aktualizované po každé rotaci. Pro výpočet je nutné znát vektor směřující kolmo vzhůru vůči scéně (`WORLD_UP`). Generovaný svět je plochý, proto lze tento vektor nahradit konstantním vektorem $(0, 1, 0)$.

```
glm::vec3 front;
front.x = cos(glm::radians(Yaw_)) * cos(glm::radians(Pitch_));
front.y = sin(glm::radians(Pitch_));
front.z = sin(glm::radians(Yaw_)) * cos(glm::radians(Pitch_));

Front_ = glm::normalize(front);
Right_ = glm::normalize(glm::cross(Front_, WORLD_UP));
Up_ = glm::normalize(glm::cross(Right_, Front_));
```

Délku x v rovině určené osami x a z lze spočítat jako délku přilehlé odvěsny. $\cos(Yaw) = x/h$, kde h je délka přepony. Víme, že vektor má jednotkovou délku, proto $h = 1$. Stejný postup aplikujeme pro rovinu určenou osami x a y.

Tímto způsobem dopočítáme délky y a z vektoru směřujícího dopředu a normalizujeme ho. Jelikož jsou na sebe vektory kolmé, využijeme vektorového součinu, jehož výsledkem je vektor kolmý k oběma původním vektorům. Všechny vektory je nutné normalizovat, aby se předešlo jejich zkracování s tím, jak se Pitch blíží $\pm 90°$.

K zamezení převrácení os jsou z definičního oboru Pitch vyjmuty násobky $90°$.

```
if (Helpers::Math::Equal(cos(glm::radians(Pitch_)), 0.0f))
    Pitch_ -= 0.01f;
```

Výsledná nepřesnost je menší než maximální rozdíl dvou čísle typu float ϵ, která jsou považována za stejná. Funkce `Equal` porovnává desetinná čísla s přesností ϵ.

Želva vystavuje metody pro pohyb ve všech třech osách využívající vektorů `Up_, Right_, Forward_`. K pozici želvy je přičten patřičný vektor naškálovaný délkou pohybu. Např.:

```
void LSystems::Detail::Turtle::MoveForward(float dz) {
    Position_ += Front_ * dz;
}
```

5.5.3 Rozšířená abeceda

Následující symboly abecedy mají speciální význam pro jejich interpretaci.

- U/u — Posuň želvu nahoru.

- F/f — Posuň želvu dopředu.
- x — Zmenši blok produkovaný želvou.
- X — Zvětši blok produkovaný želvou.
- S — Nastav původní velikost bloku produkovaného želvou.
- + — Rotuj želvu doleva podle osy y.
- - — Rotuj želvu doprava podle osy y.
- ^ — Rotuj želvu nahoru podle osy x.
- & — Rotuj želvu dolů podle osy x.
- [— Ulož kopii želvy na vrchol zásobníku.
-] — Vyjmi želvu z vrcholu zásobníku.
- $0 - 9$ — Přepni výstupní pole.

L-systém může obsahovat jakýkoliv jiný ASCII symbol – mimo bílých znaků a # – určený pro expanzi přepisovacích pravidel. Není želvou interpretován.

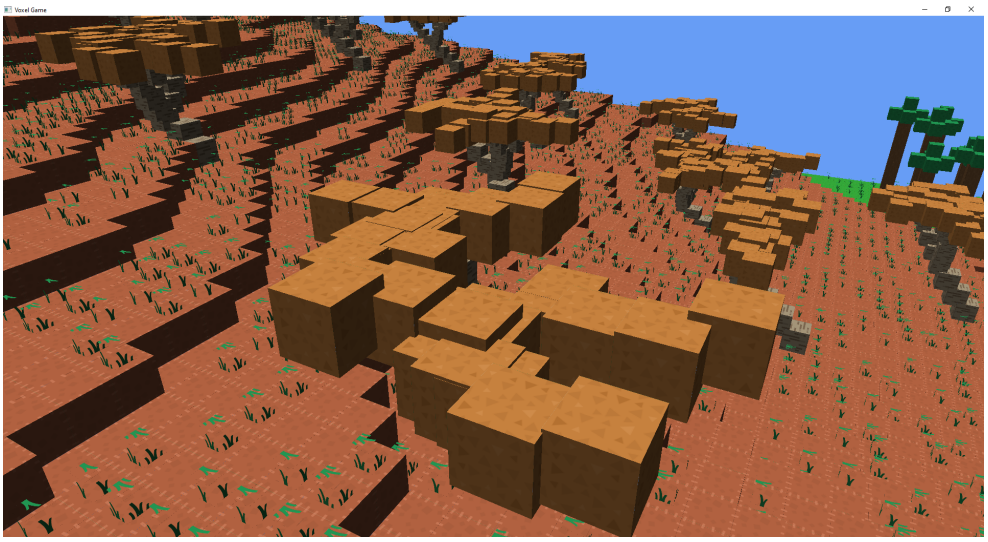
5.5.4 Ovládání želvy

Implementace želvy se nachází ve jmenném prostoru `LSystems::Detail`, uživatel by ji neměl využívat přímo, ale je pro něj připravena třída `LSystemExecutor` zajišťující generování herních objektů z poskytnutého L-systému.

`LSystemExecutor` umožňuje generovat herní objekty na základě stochastického L-systému – topologie struktury výsledného modelu se může měnit mezi jednotlivými voláními generátoru na základě parametru `salt`. L-systém lze náhodně interpretovat na základě těchto parametrů:

- Rozsah počtu provedených derivací.
- Variace v rotaci želvy. K úhlu, o který se má želva otočit, se přičte x-krát původní úhel, kde x je z [`-angleVar`, `angleVar`]. Defaultní hodnota `angleVar` je 0,2.
- Výchozí velikost generovaných objektů. Lze určit rozsahem korespondujícím s počtem derivací.

Přidání náhodného úhlu má výrazný efekt na organický vzhled rostliny. Na obrázku 5.5 lze vidět akáciové stromy vyznačující se plochou korunou. V definici L-systému jsou všechny listy ve stejné výšce, výsledný rozdíl ve výškách je způsoben opakovaným rotováním želvy. V zápisu lze vidět, že se



Obrázek 5.5: Pohled ze shora na korunu akácie

želva otočí o 45°nahoru (^), pokládá větve (u, U), skloní se o 45°(&) a pokládá listy (F). Listy by tak měly být ve stejné rovině, ale nejsou.

L-systém generující akácie:

```
45.0 45.0 0.8
# make sure the plant has splits
# random lenght stem - then split
^uA1S&F+F+F+F
U > uU
A > +uuE
A > -uE
A > +uE
A > -E
A > uuE
A > uE
# top of the plant
E > x[++++UE1S&F+F+F+F]+UE
E > x[++++++UE1S&F+F+F+F]++UE
E > x[++UE1S&F+F+F+F]-UE
```

Výstupem generátoru je 2D pole obsahující herní objekty rozdělené podle čísla výstupního bufferu, který měla želva při generování. Část enginu je tak odstíněna od textur, které jsou definované v části procedurálního generátoru. Díky tomuto rozdělení modelu je možné snadno měnit textury pro jednotlivá pole. Procedurální generátor tohoto využívá a používá stejný model – jiný běh generátoru – pro vytváření bříz a dubů, lišících se texturou kmene a listů.

Obrázek 5.6: Akáciový strom rostoucí v přírodě [43]

5.6 Modelování rostlin

Při modelování vegetace bylo třeba velkého množství pokusů a ladění, kdy rostlina nevypadala přirozeně, ale nebylo jasné, v jaké časti gramatiky je problém. Nejvíce se osvědčila technika nalezení reálné rostliny obrázek 5.6 a následné pokusy o její napodobení obrázek 5.7.

Zvláště užitečným pravidlem se ukázalo být:

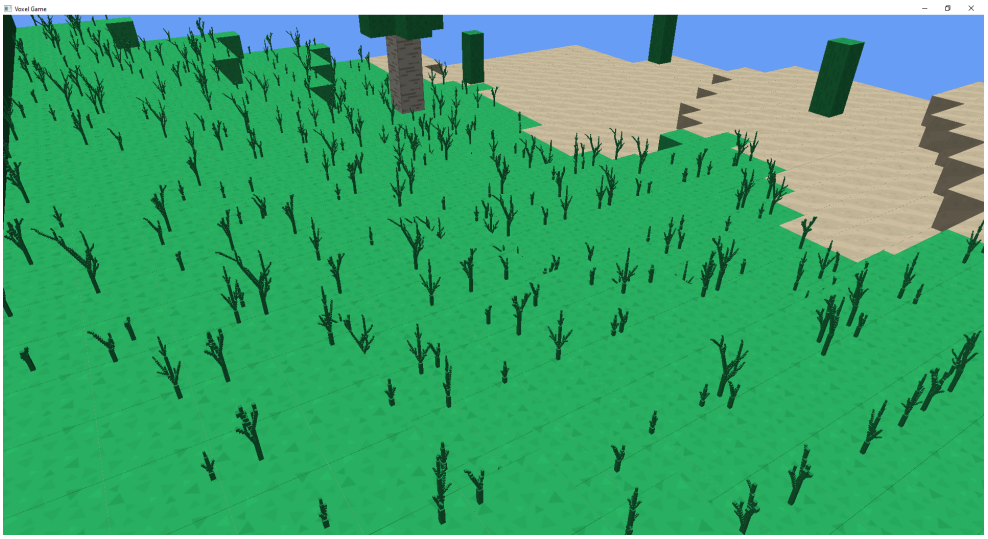
```
U > uU
```

Díky němu jsou větve blíže k zemi delší než větve navazující na korunu stromu. Pokud má rostlina význačné části je vhodné je modelovat samostatně (kmen, větve, koruna) viz L-systém generující akácie.

Tato technika ne vždy přinášela ovoce. Modelování trávy rostoucí na planinách se projevilo jako problém. Tráva neměla dostatečnou hustotu a nezapadala do kresleného vzhledu – obrázek 5.8. S navyšujícím se počtem herních objektů dramaticky rostla spotřeba paměti. Jeden herní objekt s texturou trávy byl nahrazen desítkami herních objektů, z kterých se skládal model trávy. Tento problém by mohl být řešen přesunem vytváření modelu na grafickou kartu, přidáním vertexů v geometry shaderu.

Výsledné modely svou topologií připomínaly strukturu keřů. Byly proto upraveny – přidáním listí, změnou větvení – a využity v generátoru keřů.

Obrázek 5.7: Model akáciového stromu



Obrázek 5.8: Tráva na planinách

Obrázek 5.9: Fáze růstu keře

5.6.1 Simulace růstu

Křovinatý biotop je porostlý dvěma druhy keřů, rozdělených do třech fází růstu. Ty jsou simulovány opakovaným derivováním počátečního axiomu. Keře nejmenšího vzrůstu jsou derivovány 2x, největší keře jsou derivovány 4x. Tloušťka kmene koreluje lineárně s počtem provedených derivací. Větší stromy mají širší kmen a dorůstají vyšší výšky.

Každý druh si zachovává své typické vlastnosti. Na obrázku 5.9[9] lze pozorovat stejné zakončení větví – rozdělení do dvou větví rostoucích na opačné strany – a podobný úhel v jakém se větve oddělují od kmene. Topologie rostliny se díky stochastickému L-systému mění mezi jednotlivými jedinci.

Díky těmto krokům biotop obsahuje rostliny navzájem podobného vzhledu, lišících se v drobných detailech.

5.7 Výsledky použití L-systémů

Definice L-systémů jsou krátké (≈10 řádků na jeden druh rostliny) a produkují velké množství rozdílných jedinců. Předchozí manuální definování rostlin v kódu bylo náročnější, a i při přidání více jedinců pro každý druh, by bylo snadné najít stejné. Odebráním definic rostlin z kódu se zvýšila jeho čitelnost – definice L-systémů jsou zdroje dat, který engine konzumuje. Nutnost kompilace při změně modelu byla odstraněna a zvýšila se rychlost iterace, s kterou je možné upravovat model.

[9]Pro větší názornost byly odstraněny části modelu představující listy.

Obrázek 5.10: Vegetace vygenerovaná na základě L-systémů

Vytvořením vlastního formátu pro zápis modelu rostliny se oddělila závislost na programovacím jazyce. Modely tak může vytvářet jiný člen týmu bez znalosti programování a překladu kódu.

Generováním rostlin za běhu programu se snížila rychlost jeho běhu. Tento problém lze mitigovat cachováním rostlin obsahujících velké množství herních objektů. Toto bylo provedeno pro keře. Před spuštěním generace terénu je naplněn buffer obsahující keře vygenerované na základě seedu. Buffer musí být dostatečně velký na to, aby nedošlo ke snížení diverzity rostlin. Při vytváření keře na scéně je vybrán náhodný index do bufferu, závislý na pozici keře. Vybraný model je zkopírován a přesunut na dané místo.

Při porovnání scény 5.1 ze začátku kapitoly si lze všimnout přirozenějšího vzhledu krajiny. Koruny stromů se mohou překrývat, scéna díky tomu působí více organicky – stromy v přírodě nemají přesně stanové hranice, kde končí jeden a začíná druhý. Výsledná scenérie 5.10 působí méně jednolitě díky rozdílným vývojovým stádiím rostlin.

Kapitola 6

Procedurální generování terénu

Generování částí nebo celých herních světů může výrazně zvýšit rychlost vývoje a přinést různorodost terénu. Světy mohou být generovány před spuštěním hry nebo až v průběhu hraní. Tyto dva přístupy kladou odlišné nároky na procedurální generátor.

Při generování předem je možné využít sofistikovanějších technik, které zapříčiňují delší běh generátoru, nebo vyžadují znalost celé mapy. Pokud se například hra odehrává na ostrově, je možné zvolit body na pobřeží, kde budou vyúsťovat řeky. Následně z nich lze vytvořit říční síť na základě sklonu terénu. Generátor běžící v průběhu hry si toto dovolit nemůže a musí každý kus terénu generovat nezávisle na jeho okolí.

K využití maximálního potenciálu procedurálního generování a vytvoření unikátního světa pro každého hráče, jsou využívány náhodné generátory čísel. Ty však nemohou být zcela náhodné (*true random number generator*). Pokud by hráč objevil horu a rozhodl se jí obejít z druhé strany, musí se objevit na její protější straně. Z tohoto důvodu je nutné, aby generátory produkovaly pokaždé stejnou sekvenci čísel – PRNG (Pseudo Random Number Generator) [44].

Nevýhodou PRNG je vnitřní stav, který se mění s každým vyprodukovaným číslem. Pro zaručení totožných výsledků musí být sekvence čísel použita pokaždé pro stejné operace, nebo musí být PRNG re-seedován.

Místo PRNG lze využít šumy. Šum v jistých aspektech připomíná hashovací funkce – ze vstupu produkují zdánlivě náhodný výstup. Stejně jako může být výstup hashovací funkce ovlivněn inicializačním vektorem, do výstupu šumové funkce může být zakomponován seed. Vstup může být například pozice na mapě, kde má být vygenerovaný strom. Pokud se však změní seed hry, rozmístění stromů bude zcela odlišné.

6.1 Náhodný šum

Pro generování náhodného šumu mohou být použity hashovací funkce jako MD5 nebo SHA1. Jejich požadavky byly kladeny především na zdánlivou ireverzibilitu, kvůli kryptografickému použití. Kvalita funkcí pro šum nemusí být takto dobrá, důraz je kladen především na rychlost. Engine používá implementaci Squirrela Eiserloha [45].

Funkce transformuje vstupní číslo pomocí přičítání a násobení velkým prvočíslem, bitovým posuny a XORy. Po prvním vynásobení je k číslu přičten seed, výstup se tak na něm stává závislým. Pro vytvoření šumu ze souřadnic o více dimenzích jsou použita prvočísla 198491317 a 6542989, kterými jsou souřadnice na ose y a z vynásobeny a následně sečteny.

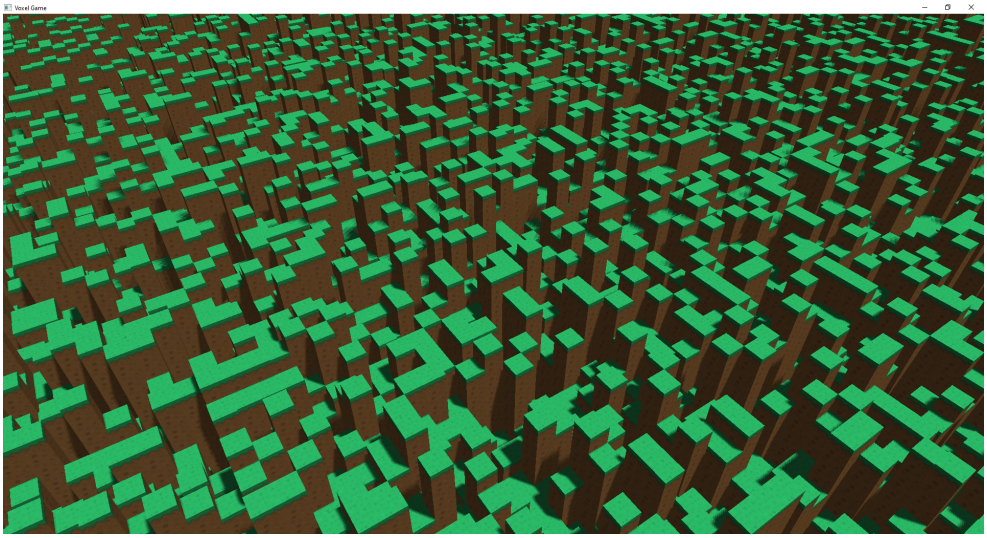
```
unsigned Random::Get1dNoise(int x, unsigned seed){
    auto mangledBits = static_cast<unsigned int>(x);
    mangledBits *= 0x68E31DA4;
    mangledBits += seed;
    mangledBits ^= (mangledBits >> 8);
    mangledBits += 0xB5297A4D;
    mangledBits ^= (mangledBits << 8);
    mangledBits *= 0x1B56C4E9;
    mangledBits ^= (mangledBits >> 8);
    return mangledBits;
}
unsigned Random::Get2dNoise(int x, int y, unsigned seed){
    return Get1dNoise(x + PRIME1 * y, seed);
}
unsigned Random::Get3dNoise(int x, int y, int z, unsigned seed){
    return Get1dNoise(x + PRIME1 * y + PRIME2 * z, seed);
}
```

6.2 Spojitý šum

Pokud má terén vypadat přirozeně, musí na sebe plynule navazovat. Použitím náhodného šumu vznikne roztříštěný, na sebe nenavazující terén, zobrazený na obrázku 6.1.

Jedním ze šumů, jehož výstup se plynule mění, je Perlinův šum, vyvinutým Kenem Perlinem v roce 1983 pro generování přirozeně vypadajících textur. Tuto texturu lze použít například jako výškovou mapu terénu. Ken Perlin v roce 2002 upravil původní algoritmus, aby byl rychlejší a neobsahoval viditelné artefakty [46]. Tento algoritmus se nazývá Vylepšený Perlinův šum a je používán dodnes.

V roce 2001 Perlin vyvinul následovníka Perlinova šum, Simplex šum. Tento šum jím byl patentován, nicméně i přes své výhody se na trhu ne-



Obrázek 6.1: Terén vygenerovaný pomocí náhodného šumu

prosadil. Tento patent vypršel 8.1.2022 [47] a nyní může být volně používán. Jeho výhody jsou [48]:

- Nižší výpočetní složitost, vyžadující méně násobení.
- Lepší škálování do vyšších dimenzí. Složitost je $O(n^2)$ oproti $O(n^{2^n})$.
- Žádné viditelné směrové artefakty.

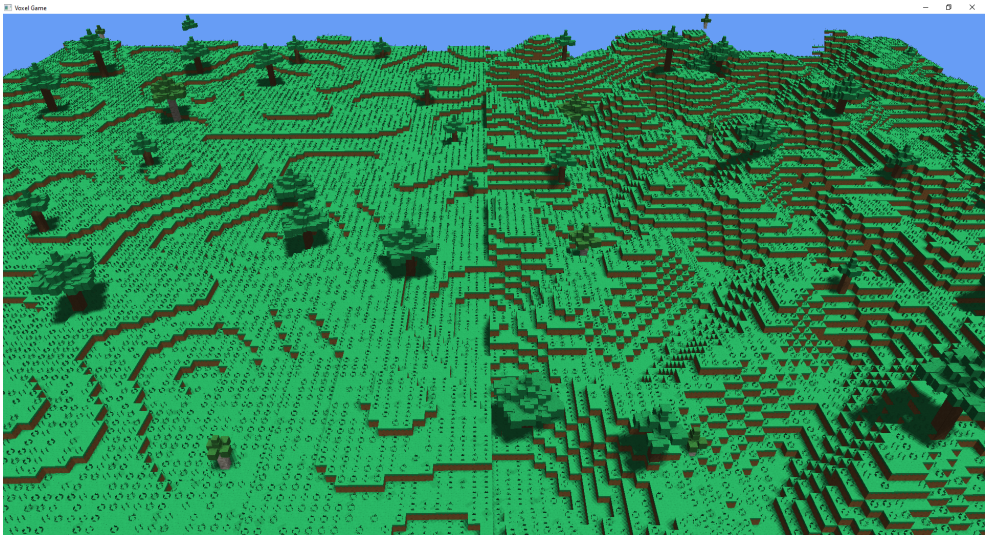
Na obrázku 6.2 je zobrazen terén vygenerovaný Perlinovým (vlevo) a Simplex šumem (vpravo). Lze pozorovat, že Simplex šum má vyšší amplitudu a frekvenci opakování vrcholů. Šumy proto nelze jednoduše zaměňovat, ale je potřeba upravit další parametry generátoru. Toto je jeden z důvodů, proč je Perlinův šum stále používán.

Pro zjednodušení jsou dále v práci používány pojmy 1D, 2D a 3D šum. Pokud není uvedeno jinak, může být použit buď Perlinův nebo Simplex šum, vždy se však jedná o spojitý šum.

6.3 Biotopy

Předchozí obrázek 6.2 zobrazuje přírodně vypadající terén obsahující pouze jeden biotop. Tato krajina není zajímavá a po čase působí opakujícím se dojmem. Země obsahuje řadu biotopů, lišících se svou specifickou faunou i flórou.

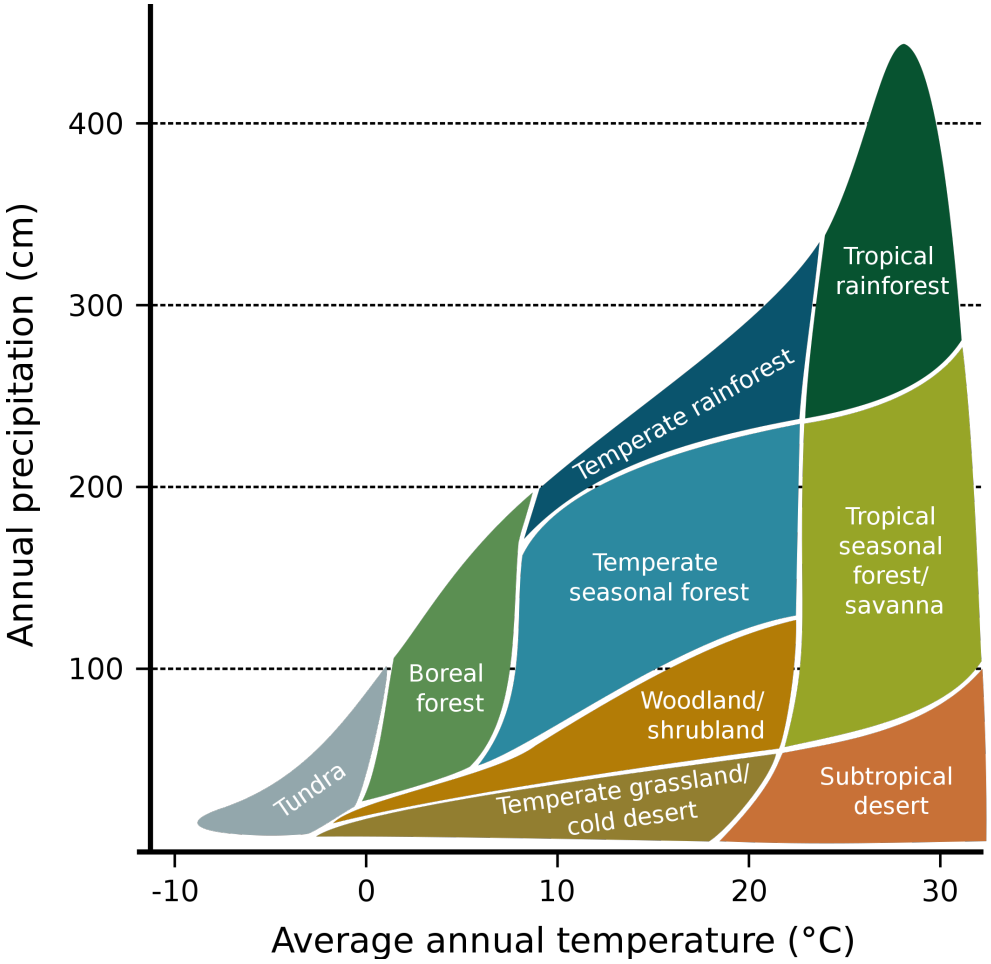
K simulaci rozložení biotopů byl využit Whittakerův systém. Robert Whittaker založil klasifikaci na průměrné teplotě a množství spadlých srážek [49]. Tato závislost je zobrazena na obrázku 6.3. Systém definuje následující biotopy:



Obrázek 6.2: Terén vygenerovaný pomocí Perlinova (vlevo) a Simplex šumu (vpravo)

- Tundra
- Tajga
- Lesy mírného pásma
- Louky mírného pásma
- Studená poušť
- Křoviny
- Zalesněné oblasti
- Subtropická poušť
- Prales mírného pásma
- Savana
- Tropický sezónní les
- Tropický prales

Whittakerův systém nedefinuje trvale zaledněné oblasti, které generátor přidává pro teploty nižší než 10 °C a jakékoliv množství srážek.

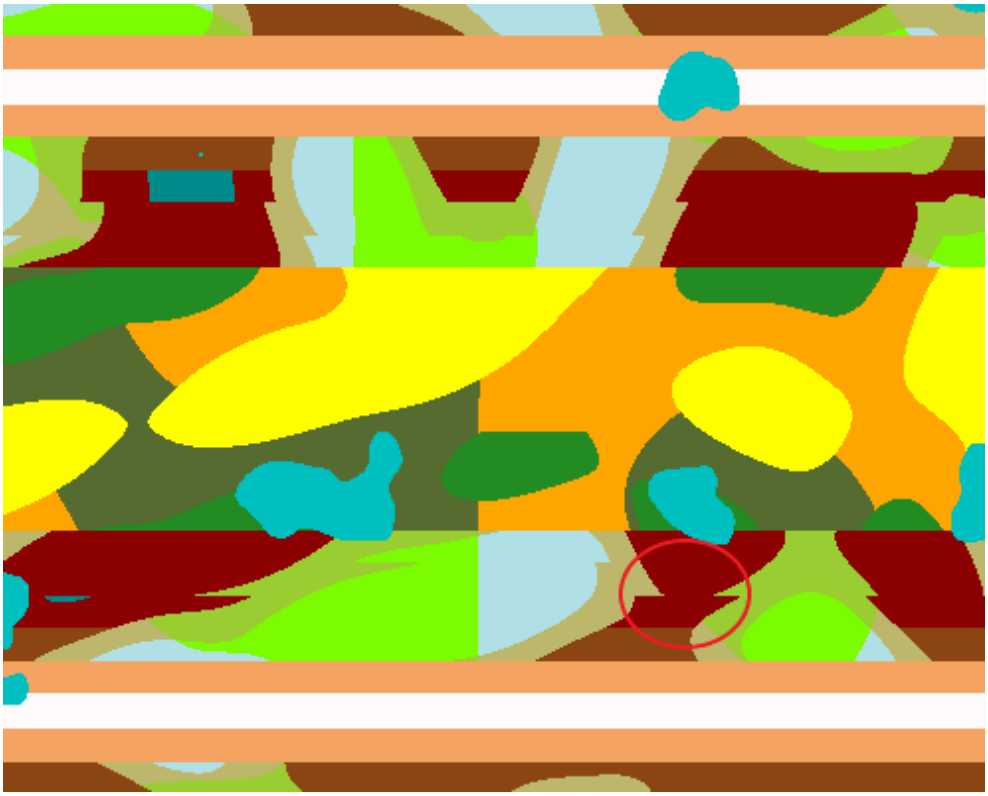


Obrázek 6.3: Whittakerův systém biotopů [50]

6.3.1 Výběr biotopu

Whittakerův systém neobsahuje všechny kombinace teplot a srážek. Generátor terénu tento systém zjednodušuje a rozšiřuje trojúhelníkový tvar na čtverec. Teplota a srážky jsou rozděleny na 10 úrovní a biotopy jsou poměrově rozděleny přes všechny hodnoty srážek. Díky tomu dokáže generátor zpracovat každou kombinací vstupů pouhým vyhledáním v tabulce.

Pokud se v políčku tabulky nachází dva biotopy, generátor z nich vybere jeden pomocí 2D šumu a pozice na mapě. Biotopy by měly pokrývat velké oblasti terénu. Proto je nutné zvolit vhodné měřítko šumu, aby nedocházelo k opakovaným změnám biotopů v jedné oblasti.



Obrázek 6.4: Rovné předěly mezi biotopy

6.3.2 Mapa srážek

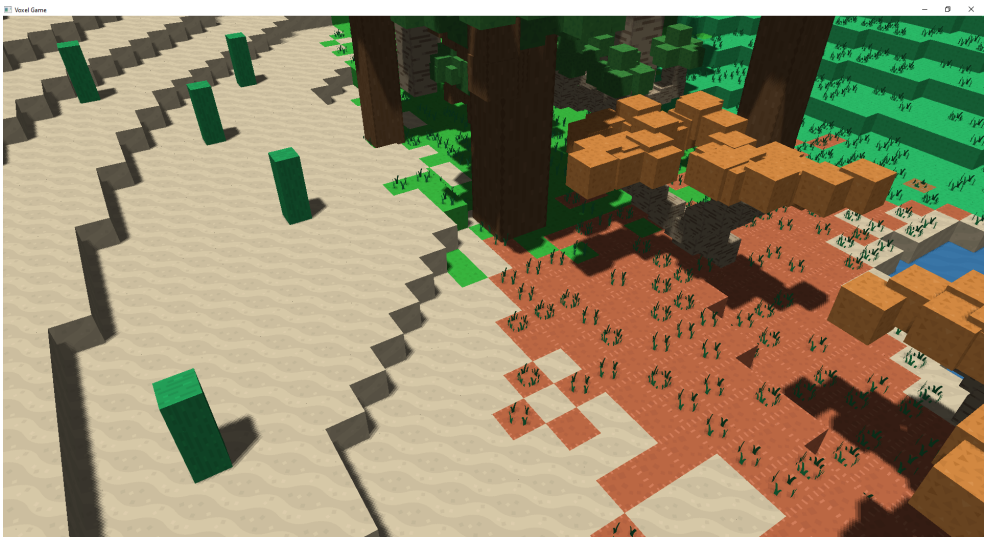
Komplexní srážkové modely mohou brát v potaz globální směr větru, pohoří, blízkost k vodím plochám způsobujících odpar... Generátor používá nejjednodušší verzi modelu, a to mapu srážek vygenerovanou pomocí 2D šumu.

6.3.3 Teplotní pásy

Země je rozdělena do teplotních pásů, čím více se člověk nachází na severu/jihu, tím nižší je průměrná teplota. Generátor tuto skutečnost simuluje a rozšiřuje jí na nekonečný terén cyklickým střídáním pásů. Pokud se hráč pohybuje na sever, teplota nejdříve klesá až na svoje minimum a poté opět roste.

Rozdělení do pásů nepřináší dostatečně dobré výsledky. Pásy jsou přesně definované a hráč si snadno všimne rovných předělů mezi biotopy. Obrázek 6.4 zobrazuje mapu biotopů, kde lze vidět dlouhé rovné předěly vzniklé změnou teploty, ale i ostré hrany biotopů vzniklé změnou kombinace teploty a srážek.

Problém s rovnými předěly lze vyřešit přičtením 1D šumu k poloze na ose z. Šum lze generovat z pozice na ose x, tím však vznikne další viditelný artefakt – všechny pásy budou mít stejný přechod. Tento artefakt působí ob-



Obrázek 6.5: Prolínání biotopů

zvláště rušivě, pokud hráč pozoruje rovnou planinu, která se opět mění v identických pásech. Engine tento artefakt mitiguje přičtením pozice na ose z. Stejné přechody tak nejsou na stejné pozici na ose x, ale jsou posunuty.

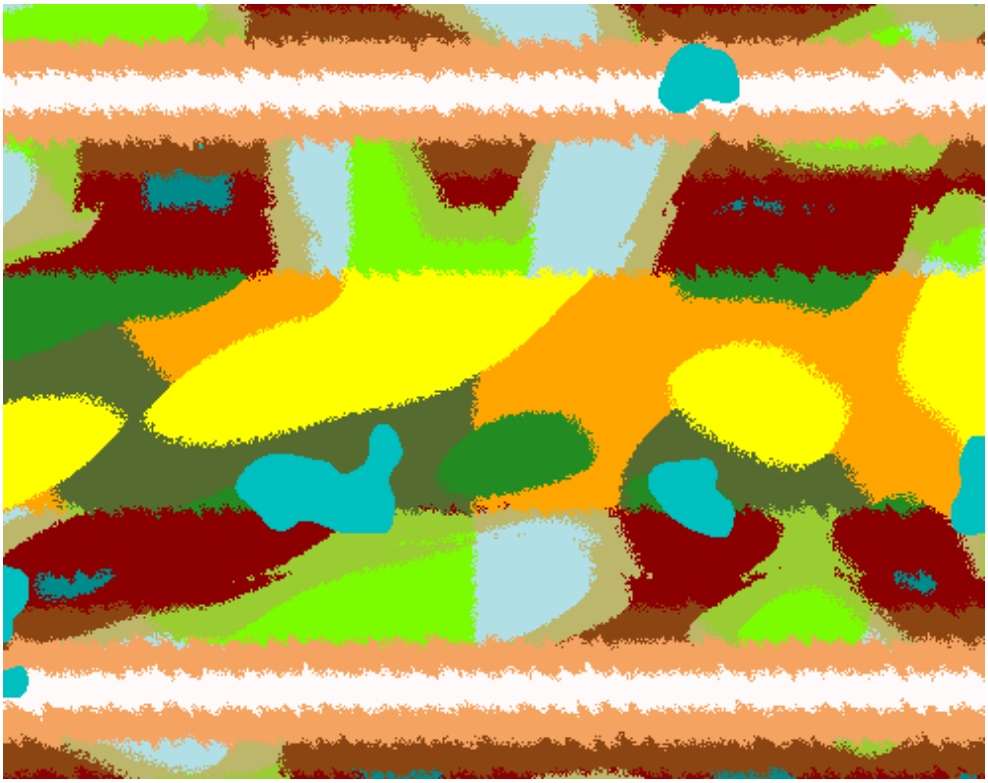
```
auto fluctuation = Engine::Random::Simplex.fractal0_1(
    2, (pos.x + pos.z * 0.5f) * 0.1f) * bandFluctuation;
```

Biotopy jsou od sebe nyní odděleny křivkou, ale přechod se stále jeví jako moc ostrý. Biotopy, jejichž hranice se prolínají, působí přirozenějším dojmem. Na obrázku 6.5 lze vidět, že na hraně mezi biotopy se prolínají oba druhy bloků a změna je postupná. Toto lze implementovat přidáním přechodových biotopů, které mohou mít další vlastnosti. Generátor používá jednodušší řešení, které dostatečně zvyšuje kvalitu přechodů – přičtení náhodného šumu k pozici, ze které je vypočtena teplota a množství srážek.

```
fluctuation += static_cast<float>(
    Engine::Random::GetNoiseLimited(pos, Weather::NOISE));
```

Na mapě 6.6 je zobrazené nové rozložení biotopů. Rovné přechody mezi biotopy byly eliminovány. V místech, kde byly kratší rovné přechody, lze nadále vidět změnu biotopů. Ostrost přechodu však byla odstraněna a biotopy se v těchto místech prolínají. Přičtení náhodného šumu také omezuje vizuální dopad, který vzniká posunem hranic teplotních pásů, pomocí stejné křivky – 1D šumu.

Dalším faktorem ovlivňujícím teplotu je výška terénu. S rostoucí výškou klesá teplota. Předchozí ukázky terénu neobsahují žádný přechod biotopů



Obrázek 6.6: Odstranění rovných přechodů

způsobený výškou terénu. Krajina je převážně rovinatá a neobsahuje žádné pohoří. Tento nedostatek je popsán v následující sekci.

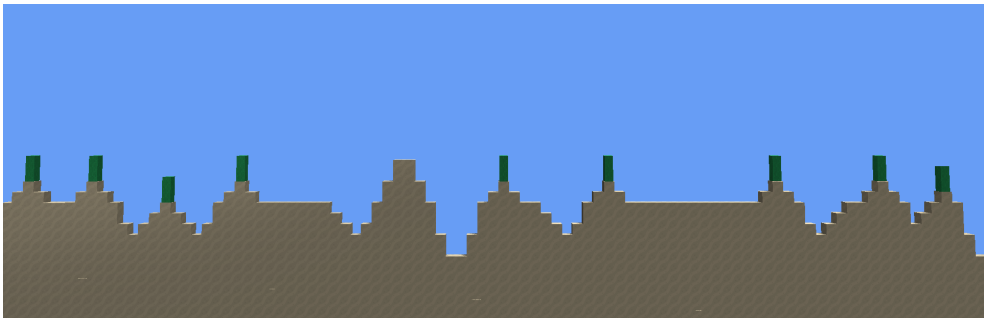
6.4 Rozmístění vegetace

Generátor simuluje přirozeně vznikající biotopy. Vegetaci je proto nutné rozmisťovat tak, aby nebyl patrný žádný vzor. Jakákoliv pravidelnost je viditelná a působí dojmem člověkem vysázeného lesa nebo sadu.

Hustota vegetace je závislá na biotopu, ve kterém se nachází. Pro rozmístění trávy jsou definovány čtyři úrovně hustoty – hustá, normální, řídká, žádná. Každá z nich říká s jakou pravděpodobností bude na dané pozici růst tráva. K jejímu rozmístění je použit náhodný 3D šum.

```
case GrassDensity::Dense:
    return Engine::Random::GetNoise0_1<float>(pos) > 0.2f;
```

Náhodný šum je vhodný pro vegetaci mající velikost maximálně jednoho bloku, která může sousedit s vegetací na dalším bloku – typicky traviny. Pro



Obrázek 6.7: Využití spojitého šumu pro rozmístění vegetace

rozmístění stromů je potřeba větší kontrola nad jejich rozestupy. Při pozorování spojitého šumu si lze všimnout, že klesá a stoupá s nepravidelnou periodou. Tohoto lze využít a stromy umístit do ostrého lokálního maxima toho šumu. Tuto situaci ilustruje obrázek 6.7, využívající spojitý 1D šum, který určuje výšku terénu a rozmístění kaktusů.

Jak lze vidět z obrázku 6.8 vzdálenost mezi stromy (červené čtverce) lze snadno škálovat změnou měřítka (vydělením souřadnic) použité mapy šumu. Vzdálenosti mezi stromy jsou různě velké. V některých částech mohou růst stromy blízko sebe, v jiných je větší prostor beze stromů.

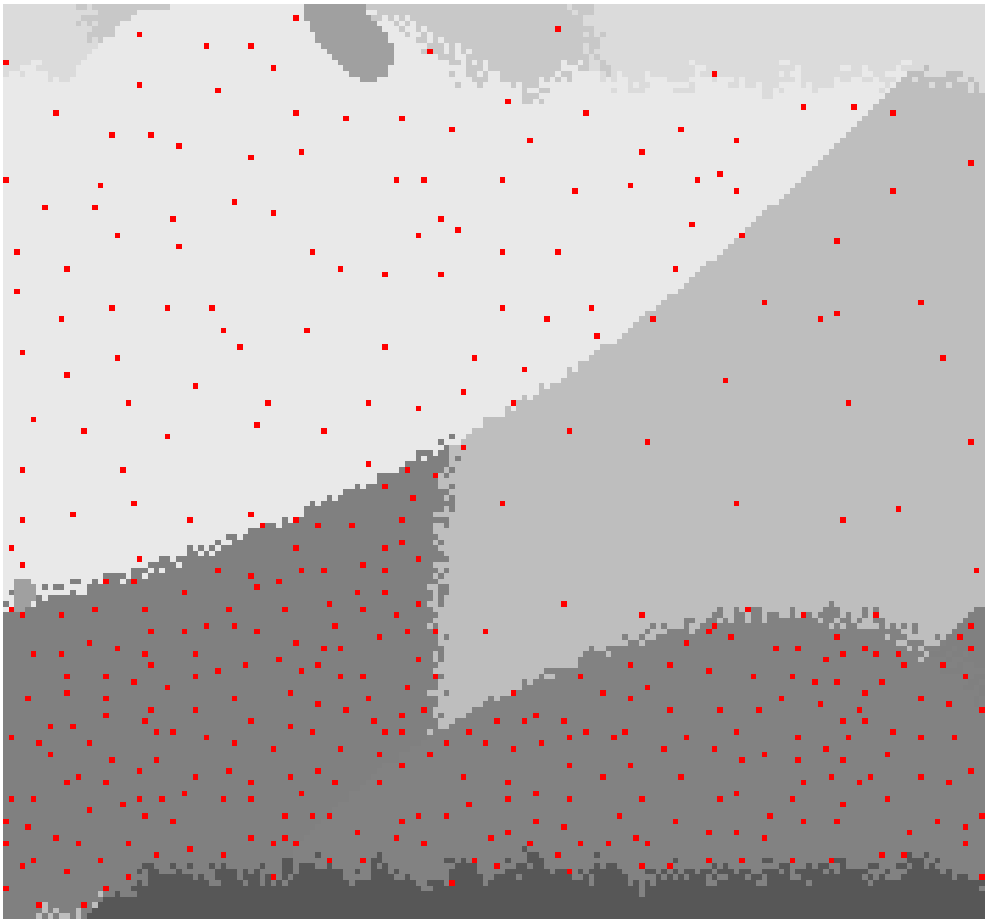
6.5 Výška terénu

Perlinův šum umožňuje generovat terén, který pozvolně mění svou výšku. V přírodě však existují hory, útesy, skály, které svůj výškový profil mění rapidně. Tohoto efektu není možné dosáhnout pouze pomocí jednoho šumu.

Komplexnější terén je možné generovat kombinací několika map šumu do jedné. Například jedna mapa může představovat výšku terénu – pohoří, vysočiny, roviny – a druhá drobné změny v terénu. Avšak prostřednictvím tohoto způsobu nelze docílit prudké změny terénu. Pro vytvoření prudkého stoupání generátor nelineárně transformuje intervaly šumu na výšku terénu. Transformační funkci zobrazuje graf 6.9.

Nicméně s tímto mapováním by měly všechny útesy stejný sklon a pohoří by se nemohla zdvihat pozvolně. Generátor proto kombinuje výšky z následujících šumů:

- Základní výška — Určuje výšku terénu před přidáním pohoří. Níže položený terén častěji obsahuje vodní plochy.
- Hory — Šum, jehož průběh zobrazuje graf 6.9. Určuje, kde se mají nacházet hory.



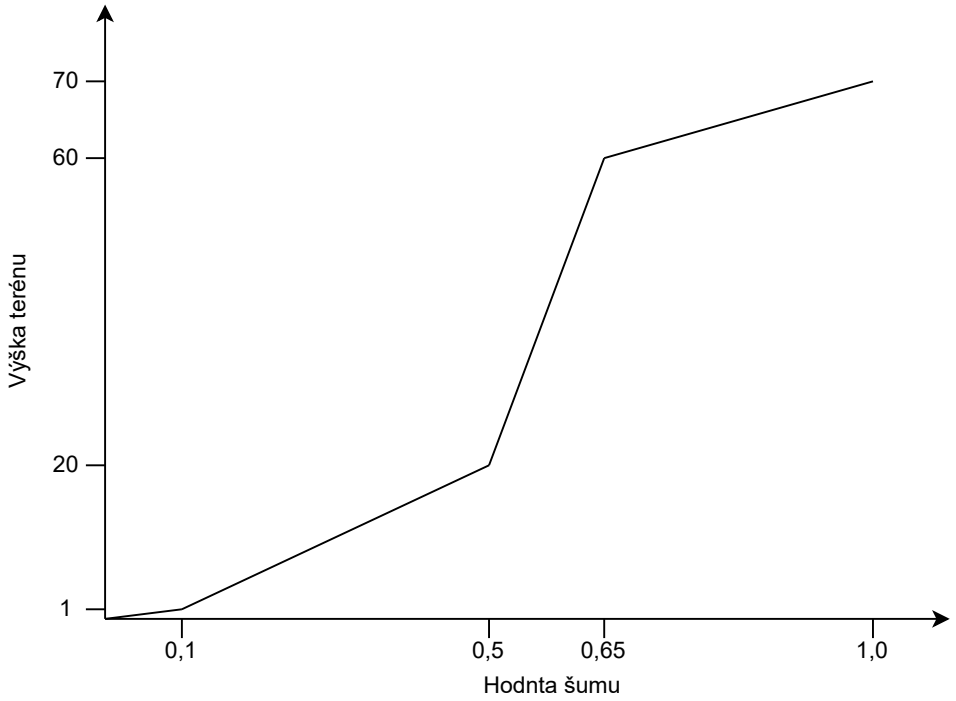
Obrázek 6.8: Rozmístění stromů v závislosti na biotopu

- Pohoří — Používá podobné mapování hodnot jako šum pro hory. Definuje, v jaké části mapy se můžou vyskytovat hory a určuje jejich výšku a úhel stoupání. Má větší měřítko, než šum pro hory.

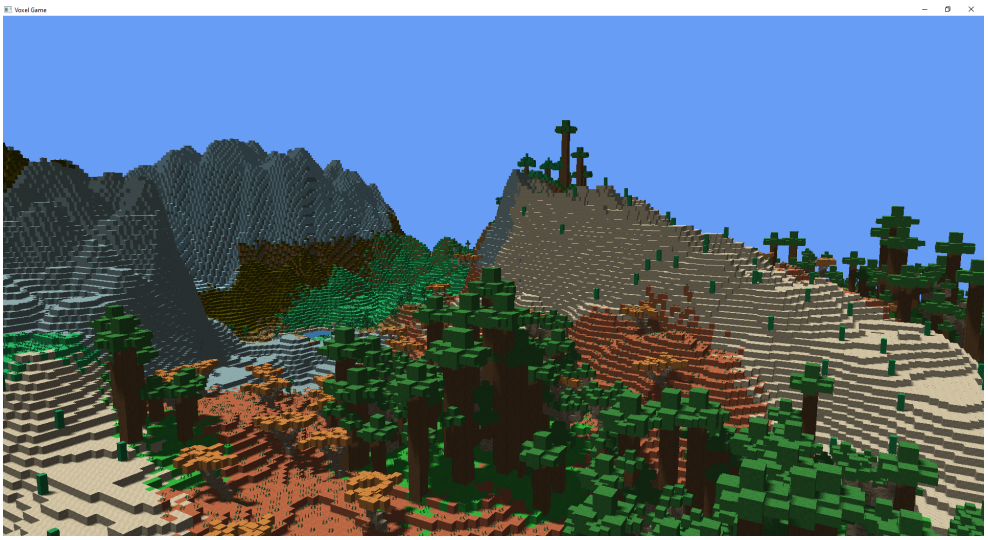
- Údolí a vrcholy — Vytváří ostré špičky hor a údolí, která by byla erodována vodními toky. Hodnota 0,5 značí nejnižší bod údolí. Hodnoty 0 a 1 vrchol hory.

- Detaily — Nejvíce patrný na nízko položených oblastech. Přidává drobná stoupání a klesání.

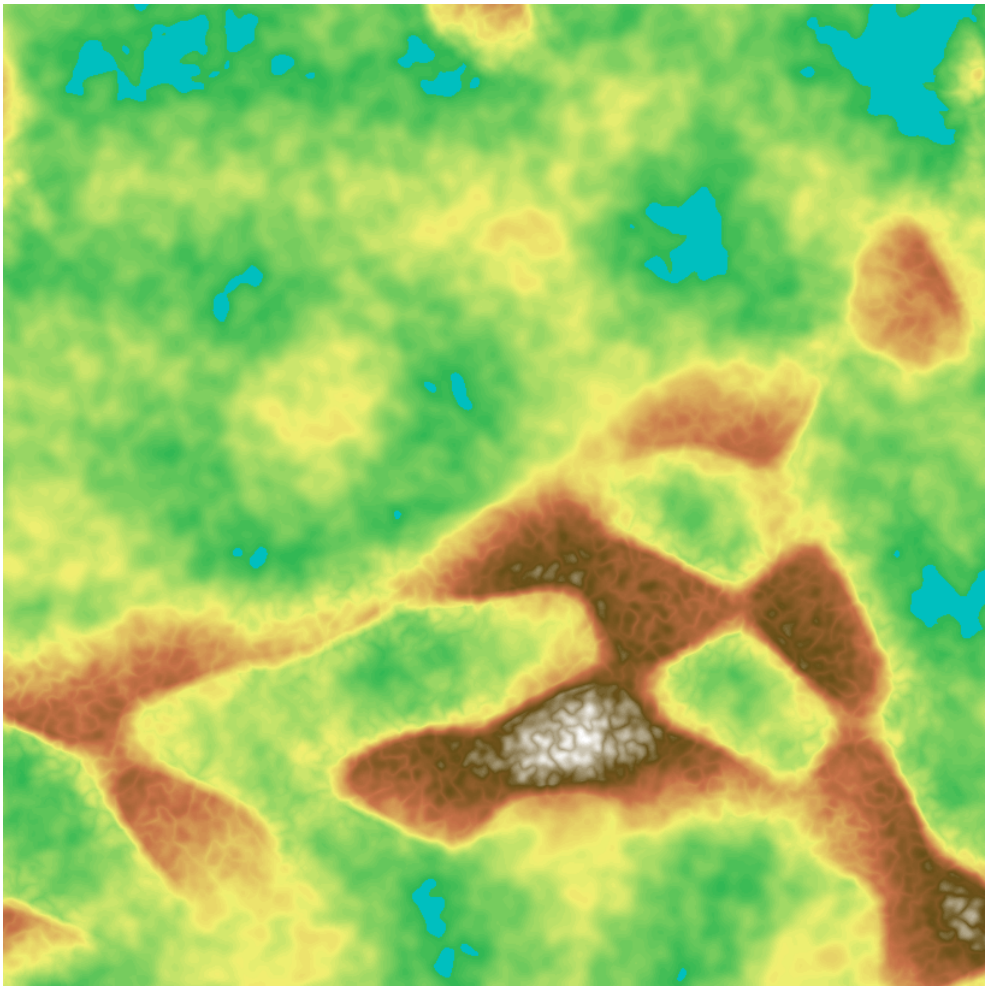
Výsledky zkombinovaných šumů zobrazuje obrázek 6.10. V jeho pravé části pozvolně stoupá pohoří, které je následně protnuté údolím s ostrými úbočími. Kombinace více šumů vede k vizuálně zajímavější krajině, která má méně monotónní vzhled.



Obrázek 6.9: Transformace šumu na výšku terénu



Obrázek 6.10: Generace pohoří a hor



Obrázek 6.11: Mapa výšky terénu

Mapa 6.11 zobrazuje výšku terénu. Pro šum generující pohoří bylo zvoleno větší měřítko, aby došlo k rozdělení světa na část neobsahující pohoří a na část obsahující hory a údolí. Na mapě je taktéž vidět maximální výška hor. Hory v předhůří nedosahují takových výšek, jako hory v centru pohoří.

Kapitola 7

Testování

V této kapitole je shrnuto testování, které probíhalo při vývoji enginu a generátoru terénu. Poslední část se zabývá testy výkonu, které proběhly s finální verzí enginu.

7.1 Automatizované testování

Pro unit testování byl zvolen testovací framework Catch2 [51]. Mezi jeho výhody patří možnost poskytnutí vlastní main funkce. Spuštění testů bylo možné navázat na spuštění enginu (mohou běžet i samostatně). Tato možnost byla zapnuta v defaultním nastavení a pomohla urychlit vývoj, díky včasnému odhalení chyb, které by se projevily vizuálně. Mezi tyto chyby patří hlavně zpracování zdrojů enginu, které nemůže zachytit kompilátor. Například předzpracování shaderů před jejich překladem (vložení konstant do zdrojového kódu shaderu) nebo generace L-systémů.

Automatizované testy jsou ve složce *tests* a jejich struktura odpovídá struktuře zdrojových souborů (složka *src*), které testují. Projekt obsahuje 340 asercí rozdělených do 14 skupin podle souborů, které testují. Pro spuštění testů byla vytvořena nová konfigurace – *Test*, která na základě podmíněného překladu spustí pouze testy.

Testovací framework v základním nastavení zobrazuje pouze testy, které neproběhly úspěšně. Při zapnutí zobrazení všech testů může výsledek vypadat následovně:

```
Resource manager test
  Replaces multiple preprocessor macros
-------------------------------------------------------------
C:\proj\VoxelGame\tests\renderer\ShaderTest.cpp(20)
.............................................................
C:\proj\VoxelGame\tests\renderer\ShaderTest.cpp(27): PASSED:
  REQUIRE( ContainsLine(source, "#define CONS2 2.0") )
```

```
with expansion:
  true

-------------------------------------------------------------
game object tests
  Component can be added
-------------------------------------------------------------
C:\proj\VoxelGame\tests\engine\GameObjectTest.cpp(6)
.............................................................

C:\proj\VoxelGame\tests\engine\GameObjectTest.cpp(10): PASSED:
  REQUIRE( go.HasComponent<Components::Transform>() )
with expansion:
  true

=============================================================
All tests passed (340 assertions in 14 test cases)
```

7.2 Vizuální testování

Generátor terénu je komplexní systém a jeho výstup není přesně definovaný – může se měnit se změnou jeho parametrů. Testování vyžadovalo opakované spouštění hry a manuální kontrolu požadovaných výsledků. Vizuální kontrola je časově náročná a je v ní možné udělat chybu, vzhledem k množství dat, která jsou zobrazena (hory, stromy, přechody biotopů, ...). Někdy také nemožná – vlhkost a teplota nejsou zobrazeny[10], ale interpretovány generátorem.

Z těchto důvodů byl vyvinut nástroj Map Visualizer [52], který umožňuje načíst soubor exportovaný generátorem a vizualizovat ho jako mapu. Některé z těchto map již byly v textu zobrazeny. Nástrojem je možné vizualizovat:

- Výšku terénu
- Distribuci stromů
- Rozmístění biotopů
- Teplotu
- Vlhkost

Vizualizátor dokáže načíst libovolně velkou scénu a každý blok zobrazuje jako jeden pixel. Díky tomu je možné najednou zobrazit vetší herní plochu, než by bylo možné s dostatečným detailem zobrazit ve 3D prostoru enginu.

[10] Tento problém se v praxi často řeší debugovací obrazovkou, která zobrazuje hráči neviditelné údaje.

Tabulka 7.1: Testovací zařízení

Zařízení	**CPU**	**GPU**	**RAM**
Notebook	Intel Core i7-7600U 2.80GHz	Intel HD Graphics 620	16GB DDR4
Stolní PC	Intel Core i7-4790K 4.00GHz	GeForce GTX 960	16GB DDR3

Tabulka 7.2: Průměrná doba vykreslení snímku

Zařízení	**17 chunků**[11] **3 CMS**[12]	**17 chunků 5 CSM**	**40 chunků 3 CSM**
Notebook	67,65	87,01	152,77
Stolní PC	20,61	26,31	62,13

Tabulka 7.3: Směrodatná odchylka vykreslení snímku

Zařízení	**17 chunků 3 CMS**	**17 chunků 5 CSM**	**40 chunků 3 CSM**
Notebook	43,65	45,30	62,17
Stolní PC	6,75	8,61	9,85

7.3 Měření výkonu

K měření výkonu byla použita dvě zařízení, jejichž parametry jsou zobrazeny v tabulce 7.1.

Výkonnostní testy byly zaměřeny na zjištění maximálního množství vykreslitelných objektů a jejich vliv na obnovovací frekvenci. Velikost scény byla upravována změnou vykreslovací vzdálenosti udávané v chuncích. Pokud je vykreslovací vzdálenost 10 chunků, velikost scény bude 336 x 336 bloků ($336 = 2 \cdot 10 \cdot 16 + 16$).

Tabulka 7.2 zobrazuje průměrnou dobu nutnou pro vykreslení jednoho snímku v milisekundách. Lze si všimnou, že PC s výkonnější grafickou kartou dosahuje přibližně trojnásobného zrychlení. Tabulka 7.3 udává výběrovou směrodatnou odchylku vypočtenou z doby nutné pro vykreslení snímku. Výkonnější HW dokázal udržel stabilnější vykreslovací frekvenci při rozhlížení se po scéně, způsobující změnu vykreslených objektů.

Při zjišťování maximálního počtu vykreslitelných objektů byl nalezen limit 32bitové verze aplikace. Množství spotřebované paměti RAM na scéně obsahující přibližně 6 milionů objektů, přesáhl 2 GB a OpenGL nebylo schopno

[11] Počet objektů na scéně byl 1 049 644 a 5 707 621 pro vzdálenost 17, respektive 40 chunků.

[12] Počet kaskád stínů.

alokovat dostatečně velký buffer pro předání dat.

Po přepnutí na 64bitovou verzi byl engine schopen vykreslit scénu obsahující 37 736 428 objektů. Průměrný čas pro vykreslení jednoho snímku byl 86,18 ms se směrodatnou odchylkou 25,11 ms (měřeno na PC). Spotřeba paměti vzrostla na 8,3 GB převyšující velikost paměti GPU (4 GB), ale díky ořezání scény není nutné předávat všechny objekty GPU.

Měření ukázala, že engine je schopen využít potenciálu výkonnějšího hardwaru a zvýšit na něm svou výkonost. Obnovovací frekvence je závislá na složitosti použitých shaderů a velikosti scény. Při pohledu na část scény, obsahující menší počet objektů, dochází ke snížení velikosti předaných dat a obnovovací frekvence se zvýší. Vzniká zde tedy prostor pro budoucí optimalizace datových struktur a zmenšení velikosti dat, reprezentující herní objekt.

Závěr

Hlavním cílem práce bylo navrhnout a vytvořit renderovací engine vykreslující voxelovou grafiku a jeho schopnosti otestovat pomocí vlastního generátoru terénu, který s ním bude komunikovat.

Pro engine byla implementována sada shaderů, využívající možnosti přesunout výpočty na grafickou kartu. Tyto shadery umožňují nasvítit scénu globálním zdrojem světla nebo světlem bodovým. Ke zvýšení vizuální kvality byl vytvořen shader přidávající objektům stíny. Uživateli je umožněno vytvářet vlastní shadery a použít je k vykreslení scény.

Engine je schopen zpracovat stovky tisíc objektů, množství dostatečné pro zobrazení rozsáhlého venkovního prostředí. Velikost scény je možné škálovat a snížit tak nároky na CPU a GPU. Samostatně lze škálovat kvalitu stínů vykresleného snímku, podle možností grafické karty.

Spolu s enginem byl vyvinut procedurální generátor vytvářející terén skládající se z rovin, hor a pohoří. Herní svět pokrytý biotopy, jež se nacházejí na Zemi. Biotopy se mění na základě nadmořské výšky a zeměpisné šířky, jejíž koncept je rozšířen na nekonečný terén. Každý biotop obsahuje vegetaci, která je generována na základě L-systémů.

Herní svět je generován deterministickým algoritmem – nemění se při opakovaném běhu generátoru. Nový svět je možné vytvořit změnou seedu, na nějž jsou navázány všechny funkce generátoru.

K navigaci po světe byl vytvořen prototyp hráče, kterému je umožněno pohybovat se libovolným směrem. Svět je průběžně generován v závislosti na jeho poloze. Oblasti, které opustil jsou uvolněny z paměti.

Práce obsahuje ukázkové scény představující možnosti enginu. Na scénách je zobrazeno například složení bloku z více textur nebo generování rostlin L-systémy. Další ukázky a kód použitý pro jejich generování lze nalézt v příloze B.

Scénu je možné exportovat do souboru ve formátu JSON. Možnosti použití exportované scény byly ukázány na nástroji, který načte scénu a na jejím základu vytvoří mapy terénu.

Třídy enginu jsou otestovány unit testy. Na generátoru terénu byla ukázána možnost, jak testovat jeho výstup pomocí dodatečných vizuálních nástrojů.

Celý projekt je volně dostupný na stránkách: `https://github.com/heppyn/VoxelGame`. MIT licence umožňuje upravovat a dále distribuovat celý projekt bez jakéhokoliv omezení. Engine může být použit jako alternativa k voxel.js, který používá JavaScript a WebGL.

Možnosti rozšíření

Práce se specializuje pouze na úzkou oblast funkcí, které herní enginy poskytují. Existuje proto nespočet možností, jak projekt rozšířit. Mezi ně patří:

- Kolize a hráč — Prototypu hráče je umožněno pohybovat se libovolně po scéně. To znamená i skrze bloky. Pro využití ve hře, by bylo nutné implementovat kolizní systém a doprogramovat mechaniky hráčem ovládané postavy.
- Rozšíření generátoru terénu — Generovaný terén neobsahuje řadu úkazů, které lze najít v přírodě. Mezi ně patří například převislé skalní stěny či jeskyně. Rozšířením generátoru by došlo ke vzniku ještě rozmanitějšího terénu, který by mohl přinášet další herní prvky – monstra schovaná v jeskyních atd.
- Optimalizace datových struktur — Engine dokáže vykreslit jakoukoliv geometrii – stačí definovat její vrcholy (bloky mohou být složeny z jednotlivých stěn krychle). Pomocí UV mapování [53] by bylo možné vyhnout se skládání bloků a snížit počet herních objektů.

  Data herního objektu jsou grafické kartě předána jako matice velikosti 4 x 4. Pokud by byl vytvořen vlastní formát pro uložení dat objektů a matice transformací by byla vytvořena až na GPU, došlo by ke snížení množství přenesených dat mezi CPU a GPU.

Literatura

[1] Gregory, J.: *Game Engine Architecture, Third Edition*. A K Peters/CRC Press, třetí vydání, 2018, 11 s., [cit. 2022-03-28].

[2] Úvod do dokumentace Godot enginu [online]. [cit. 2022-03-28]. Dostupné z: `https://docs.godotengine.org/en/stable/about/introduction.html`

[3] Sedm nejlepších herních enginů roku 2021 [online]. [cit. 2022-03-28]. Dostupné z: `https://www.incredibuild.com/blog/top-7-gaming-engines-you-should-consider-for-2020`

[4] voxel.js [online]. [cit. 2022-03-28]. Dostupné z: `http://www.voxeljs.com/`

[5] Voxel Farm [online]. [cit. 2022-03-28]. Dostupné z: `https://www.voxelfarm.com/gaming.html`

[6] Přehled voxelových enginů [online]. [cit. 2022-03-28]. Dostupné z: `https://www.gamedesigning.org/engines/voxel/`

[7] Gregory, J.: *Game Engine Architecture, Third Edition*. A K Peters/CRC Press, třetí vydání, 2018, 38–59 s., [cit. 2022-03-28].

[8] Entity Component System [online]. [cit. 2022-04-13]. Dostupné z: `https://www.guru99.com/entity-component-system.html`

[9] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-16]. Dostupné z: `https://learnopengl.com/Getting-started/Hello-Triangle`

[10] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`

[11] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: `https://www.khronos.org/opengl/wiki/Tessellation`

[12] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: https://www.khronos.org/opengl/wiki/Fragment

[13] Členové skupiny Khronos [online]. [cit. 2022-04-13]. Dostupné z: https://www.khronos.org/members/list

[14] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-02-28]. Dostupné z: https://learnopengl.com/Getting-started/Creating-a-window

[15] Dokumentace OpenGL [online]. [cit. 2022-03-28]. Dostupné z: https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

[16] Dokumentace OpenGL [online]. [cit. 2022-04-1]. Dostupné z: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

[17] Dokumentace GLM [online]. [cit. 2022-04-1]. Dostupné z: https://glm.g-truc.net/0.9.5/api/index.html

[18] Použití stb image [online]. [cit. 2022-04-1]. Dostupné z: https://github.com/nothings/stb/blob/af1a5bc352164740c1cc1354942b1c6b72eacb8a/stb_image.h#L20

[19] Knihovna implementující Perlinův šum [online]. [cit. 2022-04-1]. Dostupné z: https://github.com/Reputeless/PerlinNoise

[20] Knihovna implementující Simplex šum [online]. [cit. 2022-04-13]. Dostupné z: https://github.com/akriegman/SimplexNoise

[21] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-16]. Dostupné z: https://learnopengl.com/Advanced-OpenGL/Instancing

[22] Návrhový vzor Flyweight [online]. [cit. 2022-04-13]. Dostupné z: http://gameprogrammingpatterns.com/flyweight.html

[23] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-17]. Dostupné z: https://learnopengl.com/Advanced-OpenGL/Face-culling

[24] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-17]. Dostupné z: https://learnopengl.com/Advanced-OpenGL/Blending

[25] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-24]. Dostupné z: https://learnopengl.com/Lighting/Basic-Lighting

[26] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-26]. Dostupné z: https://learnopengl.com/Lighting/Lighting-maps

[27] Čtení z nenastavené textury [online]. [cit. 2022-01-26]. Dostupné z: https://stackoverflow.com/questions/28411686/opengl-reading-from-unbound-texture-unit

[28] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-26]. Dostupné z: https://learnopengl.com/Lighting/Light-casters

[29] Útlum světla v závislosti na vzdálenosti [online]. [cit. 2022-01-26]. Dostupné z: https://wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenuation

[30] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-02-19]. Dostupné z: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

[31] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-02-19]. Dostupné z: https://learnopengl.com/Guest-Articles/2021/CSM

[32] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: https://www.khronos.org/opengl/wiki/Array_Texture

[33] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage3D.xhtml

[34] Časové dotazy OpenGL [online]. [cit. 2022-02-27]. Dostupné z: http://www.lighthouse3d.com/tutorials/opengl-timer-query/

[35] Dokumentace OpenGL [online]. [cit. 2022-02-27]. Dostupné z: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glQueryCounter.xhtml

[36] Dokumentace OpenGL [online]. [cit. 2022-02-27]. Dostupné z: https://www.khronos.org/opengl/wiki/Early_Fragment_Test

[37] Togelius, J.; Shaker, N.; Nelson, M. J.: Grammars and L-systems with applications to vegetation and levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, editace N. Shaker; J. Togelius; M. J. Nelson, Springer, 2016, str. 75, [cit. 2022-01-10].

[38] Turtle Geometry in Computer Graphics and Computer Aided Design [online]. [cit. 2022-04-19]. Dostupné z: https://www.cs.wustl.edu/~taoju/research/TurtlesforCADRevised.pdf

[39] Przemyslaw Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 7, [cit. 2022-01-10].

[40] Przemyslaw Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 24, [cit. 2022-01-10].

[41] Przemyslaw Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 28, [cit. 2022-01-10].

[42] Przemyslaw Prusinkiewicz, A. L.: Modeling of trees. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 57, [cit. 2022-01-10].

[43] Fotografie akácie [online]. [cit. 2022-01-11]. Dostupné z: https://commons.wikimedia.org/wiki/File:Umbrella_thorn_acacia_or_israeli_babool_tree_plant_acacia_tortillis.jpg

[44] Random Number Generators [online]. [cit. 2022-04-15]. Dostupné z: https://www.wolfssl.com/true-random-vs-pseudorandom-number-generation/

[45] Záznam konference o RNG založených na šumu [online]. [cit. 2022-04-10]. Dostupné z: https://youtu.be/LWFzPP8ZbdU?t=2797

[46] Vylepšený Perlinův šum [online]. [cit. 2022-04-10]. Dostupné z: https://mrl.cs.nyu.edu/~perlin/paper445.pdf

[47] Patent Simplex šumu [online]. [cit. 2022-04-10]. Dostupné z: https://patents.google.com/patent/US6867776B2/en

[48] Porovnání Perlinova a Simplex šumu [online]. [cit. 2022-04-10]. Dostupné z: https://www.bit-101.com/blog/2021/07/perlin-vs-simplex/

[49] Whittakerův systém biotopů [online]. [cit. 2022-04-10]. Dostupné z: http://www.bio.miami.edu/dana/330/330F19_9.html

[50] Obrázek Whittakerova systému biotopů [online]. [cit. 2022-04-10]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=61120531

[51] Catch2 [online]. [cit. 2022-04-17]. Dostupné z: https://github.com/catchorg/Catch2

[52] Map Visualizer [online]. [cit. 2022-04-17]. Dostupné z: https://github.com/heppyn/MapVisualizer

[53] Understanding UV Mapping and Textures [online]. [cit. 2022-04-19]. Dostupné z: https://www.spiria.com/en/blog/desktop-software/understanding-uv-mapping-and-textures/

Příloha **A**

Seznam použitých zkratek

GLSL OpenGL Shading Language

GLM OpenGL Mathematics

ECS Entity-Component-System

FPS Frames per second

PCF Percentage-closer Filtering

NDC Normalized Device Coordinates

CSM Cascaded Shadow Mapping

PRNG Pseudo Random Number Generator

Příloha B

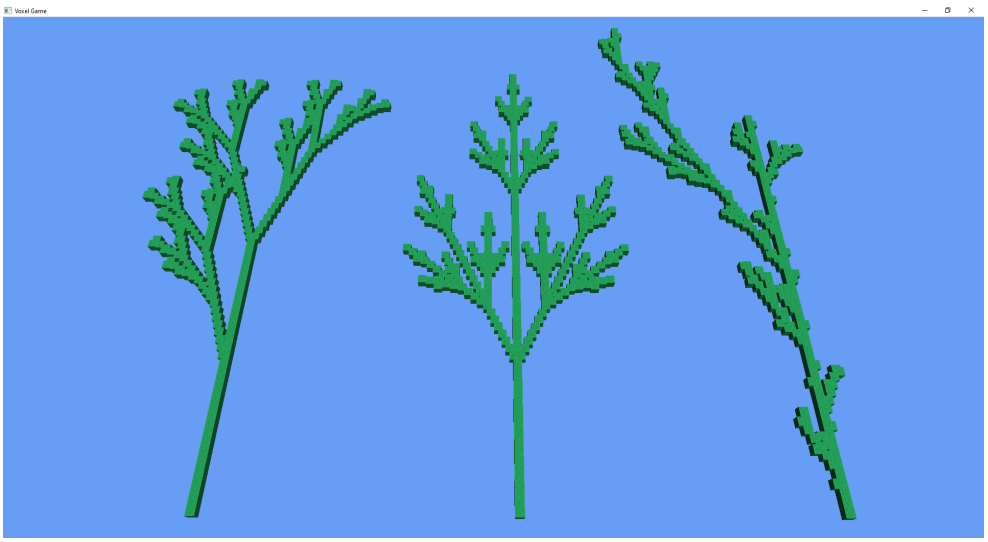
Ukázkové scény

Tato příloha zobrazuje vybrané scény ze složky *examples*, které ukazují možnosti enginu. Pro přepínaní mezi procedurální generátorem a ukázkovými scénami bylo definováno makro `USE_TERRAIN_GEN`.

B.1 Generování trávy

Generace 2D trávy na základě tří za sebou jdoucích L-systémů – obrázek B.1.

```
LSystems::LSystemExecutor ge(0, 0.0f);
Chunk chunk(glm::vec2(0.0f));
// load L-systems from file
const auto lSystems =
```



Obrázek B.1: Generace třech druhů trávy

```
  LSystems::LSystemParser::LoadLSystemFromFile(
    "./res/l-systems/plants/Grass2D.txt");
if (!lSystems.empty()) {
    auto pos = glm::vec3(0.0f);
    for (const auto& lSystem : lSystems) {
        // generate grass based on loaded L-systems
        auto objects = ge.GenerateBasedOn(pos, lSystem,
          0.3f, 5, Engine::Random::GetNoise(pos));
        // add texture to the grass
        // grass only contains one type of object - stem
        for (auto& o : objects[0]) {
            o.AddComponent<Components::SpritesheetTex>(
              glm::vec2(1.0f, 0.0f));
        }
        // move object to chunk data
        chunk.GetObjects()[Chunk::DefaultCube_].insert(
          chunk.GetObjects()[Chunk::DefaultCube_].end(),
          std::make_move_iterator(objects[0].begin()),
          std::make_move_iterator(objects[0].end()));

        pos += glm::vec3(10.0f, 0.0f, 0.0f);
    }
    chunk.FinisChunk();
}
else {
    std::cout << "Failed to load L-system\n";
}

return chunk;
```

B.2 Generování stromů

Ukázka L-systému, který je přeložen z řetězce. Generátor náhodně mění vývojová stádia stromů a upravuje podle nich velikost objektů, ze kterých jsou složeny. Kód zároveň představuje možnost složení stromu z objektů s dvěma různými texturami. Obrázek B.2.

```
Chunk chunk(glm::vec2(0.0f));

// you can generate L-systems from string
// but generation from file is preferred
std::stringstream def;
// yaw pitch shrink ratio
def << "22.5 15.0 0.8\n"
```

Obrázek B.2: Generace vývojových stádií stromu

```
    // axiom
    << "UES1u\n" // use 1 to switch output buffer
    << "U > uU\n"
    // letter > production
    // save and load state of the turtle with [ and ]
    << "E > [B][++++B][----B][++++++++B]+uxUE\n"
    << "# branch expansion\n" // comments start with #
    << "F > f[-xB]+[+xB]xF\n"
    << "F > f[+xB]-[-xB]xF\n"
    << "F > f[+xB][-xB]xF\n"
    << "# branches\n"
    << "B > ^xxfF1Sf\n"
    << "B > ^xxfF1Sxf\n"
    << "B > ^xxfF1SXf\n"
    << "B > ^^xxfF1Sf\n"
    << "B > ^^xxfF1Sxf\n"
    << "B > ^^xxfF1SXf\n";
// load L-system from string stream
const auto lSystems = LSystems::LSystemParser::LoadLSystem(def);
// set random angle to 50 %
LSystems::LSystemExecutor ge(0.5f);
// add random derivations and scale down smaller plants
ge.ScaleDerivations(4, 0.7f, 1.0f);
auto pos = glm::vec3(40.0f, 0.0f, 0.0f);
```

```
for (int i = 0; i < 8; ++i) {
    // generate tree based on definition above
    auto objects = ge.GenerateBasedOn(pos, lSystems[0],
      1.0f, 4, Engine::Random::GetNoise(pos));
    // add bark texture
    for (auto& o : objects[0]) {
        o.AddComponent<Components::SpritesheetTex>(
          glm::vec2(6.0f, 5.0f));
    }
    // add leaves texture
    for (auto& o : objects[1]) {
        o.AddComponent<Components::SpritesheetTex>(
          glm::vec2(4.0f, 3.0f));
    }
    // move object to chunk data
    chunk.GetObjects()[Chunk::DefaultCube_].insert(
      chunk.GetObjects()[Chunk::DefaultCube_].end(),
      std::make_move_iterator(objects[0].begin()),
      std::make_move_iterator(objects[0].end()));

    chunk.GetObjects()[Chunk::DefaultCube_].insert(
      chunk.GetObjects()[Chunk::DefaultCube_].end(),
      std::make_move_iterator(objects[1].begin()),
      std::make_move_iterator(objects[1].end()));

    pos += glm::vec3(12.0f, 0.0f, 0.0f);
}

chunk.FinisChunk();
return chunk;
```

B.3 Generování terénu

Terén generovaný pomocí Perlinova a Simplex šumu. Obrázek 6.2.

```
// get block height from Perlin/Simplex noise
const auto heightPer = static_cast<unsigned>(
  Engine::Random::Perlin.noise2D_0_1(
    i / freq, j / freq) * 8.0f);
const auto heightSim = static_cast<unsigned>(
  Engine::Random::Simplex.noise0_1(
    i / freq, j / freq) * 8.0f);
for (unsigned h = 0; h < heightPer; ++h) {
    chunk.AddObject(GameObjectFactory::CreateObject(
```

```
      { i, h, j },
      { 0.0f, 1.0f }));
}
glm::vec3 pos = { i, heightPer, j };
// generate grassland tree
// biome determines density, type, ...
chunk.AddObjectData(Terrain::Vegetation::TreeFactory::
  GenerateTree(pos, Terrain::BiomeType::Grassland));
// generate grassland grass
auto grass = Terrain::Vegetation::GrassFactory::GenerateGrass(
  pos, Terrain::BiomeType::Grassland);
chunk.AddObjectsTrans(std::move(grass),
  Terrain::Vegetation::GrassFactory::GrassCube());

// generate grass block with different top and sides
chunk.AddObject(GameObjectFactory::CreateObject(pos,
  { 1.0f, 1.0f }), Engine::Cube::PIPE);
chunk.AddObject(GameObjectFactory::CreateObject(pos,
  { 2.0f, 2.0f }),
  Engine::Cube::BlockFaces::CreateBlockFaces(
    Engine::Cube::Faces::TOP));
// do the same for Simplex terrain...
```

B.4 Rozmístění stromů

Vysvětlení, jak generátor rozmisťuje stromy s využitím Perlinova šumu. Stromy jsou umístěny do ostrého lokálního maxima funkce reprezentované výškou terénu. Obrázek 6.7.

```
// visualize distribution of trees using Perlin noise
constexpr auto freq = 5.0f;
const auto height = static_cast<unsigned>(
  Engine::Random::Perlin.noise1D_0_1(i / freq) * 20.0f);
for (unsigned h = 0; h <= height; ++h) {
    chunk.AddObject(GameObjectFactory::CreateObject(
      { i, h, 0.0f },
      { 6.0f, 4.0f }));
}

// place cactus only in sharp local maximum of the function
const auto left = static_cast<unsigned>(Engine::Random::
  Perlin.noise1D_0_1((i - 1) / freq) * 20.0f);
const auto right = static_cast<unsigned>(Engine::Random::
  Perlin.noise1D_0_1((i + 1) / freq) * 20.0f);
```

```
if (height > left && height > right) {
    chunk.AddObjects(Terrain::Vegetation::Tree::SpawnCactus(
      { i, height, 0.0f }));
}
```

B.5 Blok s více texturami

Složení bloku ze tří textur. Obrázek 2.1.

```
chunk.AddObject(GameObjectFactory::CreateObject(
  { 1.0f, 0.0f, 0.0f }, { 1.0f, 1.0f }), Engine::Cube::PIPE);
chunk.AddObject(GameObjectFactory::CreateObject(
  { 1.0f, 0.0f, 0.0f }, { 2.0f, 2.0f }),
  Engine::Cube::BlockFaces::CreateBlockFaces(
    Engine::Cube::Faces::TOP));
chunk.AddObject(GameObjectFactory::CreateObject(
  { 1.0f, 0.0f, 0.0f }, { 0.0f, 1.0f }),
  Engine::Cube::BlockFaces::CreateBlockFaces(
    Engine::Cube::Faces::BOTTOM));

chunk.AddObject(GameObjectFactory::CreateObject(
  { -0.5f, 0.0f, 0.0f }, { 1.0f, 1.0f }), Engine::Cube::PIPE);

chunk.AddObject(GameObjectFactory::CreateObject(
  { -2.0f, 0.0f, 0.0f }, { 0.0f, 1.0f }),
  Engine::Cube::BlockFaces::CreateBlockFaces(
    Engine::Cube::Faces::BOTTOM));

chunk.AddObject(GameObjectFactory::CreateObject(
  { -3.5f, 0.0f, 0.0f }, { 2.0f, 2.0f }),
  Engine::Cube::BlockFaces::CreateBlockFaces(
    Engine::Cube::Faces::TOP));
```

Příloha C

Obsah přiloženého CD

- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF