VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

3D HERNÍ SVĚT V OPENGL

3D GAME WORLD IN OPENGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DAVID BUCHTA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20157/2016/xbucht18

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Buchta David, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **3D herní svět v OpenGL**
3D Game World in OpenGL
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu OpenGL a její nadstavby.
2. Nastudujte metody tvorby různých efektů (částicové efekty, kouř, mlha, odlesky, bloom, shader magic, light scattering...).
3. Navrhněte a implementujte aplikaci.
4. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte video pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato práce se zabývá studiem moderních technik v počítačové grafice; návrhem a vytvořením aplikace, jež může sloužit jako jádro budoucímu hernímu enginu. V práci jsou vyzdviženy techniky tvorby rozsáhlých terénů, pokročilých stínů, generování fyzikálně založené oblohy a vykreslování velkého množství objektů. Závěrem je provedeno výkonnostní testování těchto modulů.

Abstract

Focus of this master's thesis is a study of modern techniques in computer graphics and designing and developing custom application based on which could be developed new game engine. In this thesis are highlighted techniques for creating large terrains, advanced shadows, physically based sky rendering and drawing large set of objects. Finally, performance testing of these modules is performed.

Klíčová slova

OpenGL, 3D herní svět, teselace, nepřímé vykreslování, animovaný skybox, fyzikální simulace oblohy, stínové mapy, kaskádové stínové mapy, dělení prostoru, terén, SSAO

Keywords

OpenGL, 3D game world, tesselation, indirect drawing, animated skybox, physically based sky, shadow mapping, cascaded shadow maps, spatial indexing, terrain, SSAO

Citace

BUCHTA, David. *3D herní svět v OpenGL*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Milet Tomáš.

3D herní svět v OpenGL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Buchta
19. května 2017

Poděkování

Děkuji vedoucímu mé diplomové práce Ing. Tomáši Miletovi za odborné vedení, pomoc, věcné připomínky, vstřícnost při konzultacích a za cenné rady při zpracování této práce.

Obsah

Kapitola 1

Úvod

Herní, potažmo filmový průmysl je jedním z velkých tahounů moderní ekonomiky, jehož přesah lze nalézt i v jiných oblastech než jen v zábavním průmyslu. I proto se dnes firmy předhánějí v tom, která nabídne technologicky lepší výpočetní stroje umožňující reálnější simulaci běžného světa. Ruku v ruce s výkonnějšími prostředky vznikají nové herní tituly, snažící se zvednout hozenou rukavici a představit krajinu nerozeznatelnou od reálného světa.

Každá hra má svůj základ v herním enginu. V dnešní době existuje poměrně slušný výběr v již existujících řešeních. Malé studio, nebo tzv. Indie vývojáři, jistě využijí existující profesionální engine, jehož nejpopulárnějším zástupcem, v době psaní této práce, byl engine jménem Unity. Větší studia a tvůrci špičkových her, též označovaných AAA tituly (tripple A tituly), většinou vyvíjí vlastní proprietární řešení, jež pak mohou s určitým licencováním zpřístupnit k veřejnému použití. Zástupce této skupiny tvoří např. Frostbite engine od studia EA Games, nebo Unreal Engine, který taky nejprve vznikal jako projekt pro vlastní hru a později byl uveden i pro veřejnost. Kromě vývoje enginů s jasným cílem ziskovosti vznikají i nadšenecké nebo univerzitní projekty.

Tématem mé práce je 3D herní svět v OpenGL, tudíž bych se chtěl věnovat vytvoření aplikace, která by mohla sloužit jako základ novému hernímu enginu. Proto navrhované postupy a návrh aplikace budou tvořeny s ohledem na možnost zařazení do větší aplikace s logickým oddělením herní od grafické logiky. Kromě samotných kódů by měly posloužit i zjištěné zkušenosti a nedostatky.

Jednotlivé kapitoly se postupně budou zaobírat vznikající aplikací. Kapitola 2 stručně představí teoretické základy a moderní techniky počítačové grafiky. Kapitola 3 se bude věnovat návrhu aplikace, nezávislém na programovacím jazyce, avšak předpokládajícím OOP, aby následně kapitola 4 uvedla zkušenosti a komplikace, jež mohou nastat při konkrétní implementaci návrhu. V kapitole 5 budou provedeny výkonnostní testy vzniknuvší aplikace. Závěrečná kapitola se bude věnovat shrnutí zjištěných informací a diskutovat možná rozšíření.

Kapitola 2

Moderní techniky v počítačové grafice

První část mé práce budu věnovat teoretickému rozboru postupů používaných v moderním vývoji her. Postupně tedy budou představeny techniky užitečné pro samotné vytvoření herního světa, dále pak vylepšení jeho vizuálních aspektů a v neposlední řadě snížení výpočetní náročnosti.

Herní svět nebo také herní prostředí je místo hráčovy herní interakce s aplikací. Ukázka herního světa je na obrázku 2.1, který představuje fantasy svět podobný lidskému. Avšak hra se může odehrávat i v jiných prostředích, např. ve vesmíru, v lidském těle, na fotbalovém stadioně, případně se jedná o jejich kombinace.

Tato práce se zabývá vytvořením herního světa podobnému uvedené ilustraci. Z toho důvodu bude v této kapitole uvedeno i několik technik, u nichž nemusí být uplatnění u jiných druhů světů.

Obrázek 2.1: Jedna možná podoba herního světa. Tento svět byl zachycen v počítačové hře Skyrim od amerického herního studia Bethesda Softworks, LLC.

2.1 Moderní OpenGL

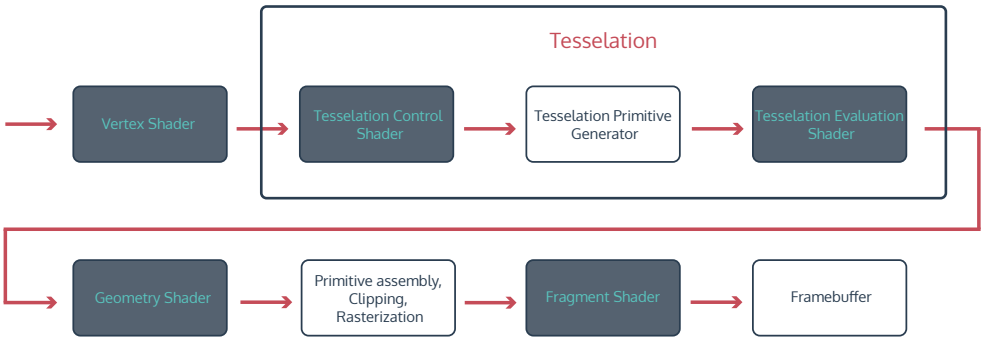
Při vykreslování geometrie v OpenGL probíhá rozsáhlá komunikace mezi CPU a GPU (a případně i mezi RAM a VRAM). U každého kreslícího příkazu je potřeba nastavit celý vykreslovací řetězec a v neposlední řadě je zde i synchronizace komunikace. Toto zpoždění se nazývá driver overhead (zpoždění způsobené ovladačem karty, zkratka DO) a jedná se o závažný problém nynějších aplikací. DO problému si začali všímat i tvůrci grafických programovacích rozhraní a snaha o jeho řešení se označuje AZDO – approaching zero driver overhead. Odpovědi přišly v podobě tzv. post OpenGL api, mezi něž se řadí Vulkán, DirectX 12 nebo Metal, avšak spousta nových konceptů si našla cestu i do OpenGL. Řešením DO se zabývala i GDC přednáška v San Francisku, rok 2014 [7], která posloužila jako zdroj pro tuto sekci.

Grafické adaptéry také obsahují jednotky pro hardwarovou akceleraci. Využití HW akcelerace společně s eliminací DO dokáže zrychlit aplikace o několik řádů. Zdroj k moderním technikám byla OpenGL SuperBbible 6 [21] a OpenGL příručka [19].

2.1.1 Teselace

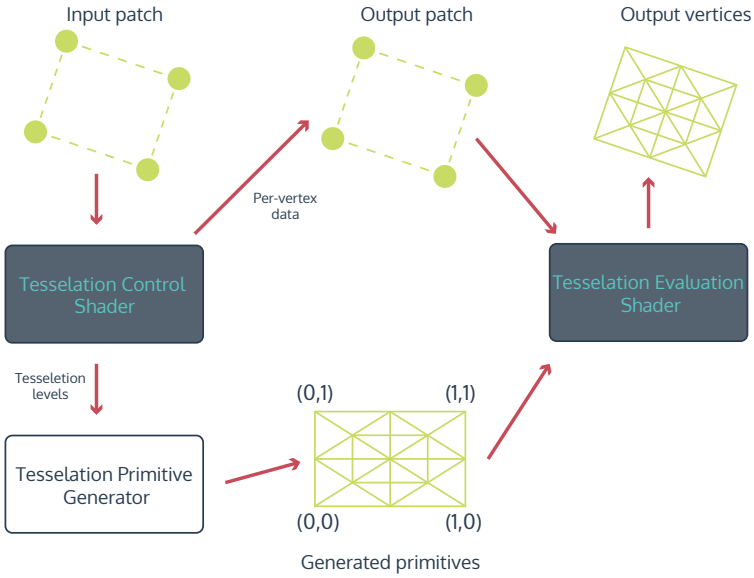
Teselace jako koncept byla zavedena v OpenGL verzi 4 přidáním dvou programovatelných a jedné fixní fáze do vykreslovacího řetězce (ilustrace 2.2). Úlohou teselačního jádra je vygenerovaní dodatečných vrcholů a primitiv v původním objektu. Vzniknuvší části mohou být dále zpracovány na GPU, aniž by byla potřeba další komunikace s procesorem. Celý proces generace je hardwarově akcelerovaný. OpenGL umožnuje teselovat pouze čáry, trojúhelníky a čtyřúhelníky.

Vstupní část teselačního jádra je Control Shader (TCS), vstupní data jsou zde dodána z Vertex Shaderu. TCS je spuštěn jednou pro každý vrchol vstupního primitiva a jeho úlohou je definovat počet teselačních úrovní pro generátor primitiv (TPG). Zjednodušeně lze říci, že TCS určuje, kolik nových primitiv má být vygenerováno.



Obrázek 2.2: Vykreslovací řetězec v OpenGL 4.x se zvýrazněnou teselační části. Vyplněné bloky představují programovatelné části řetězce.

Po vygenerování požadovaných primitiv pomocí TPG, se na řadu dostává Evaluation Shader (TES). TES je spuštěn na každý jednotlivý vrchol vstupního primitiva a zároveň i pro každý vrchol vygenerovaný pomoci TPG. S vrcholy jsou dodány také jejich parametrizované souřadnice a parametrizované souřadnice vzhledem k původnímu primitivu, např. pro čtyřúhelník se jedná o interpolované souřadnice v rozsahu $\langle 0, 1 \rangle$. Celý postup teselace je naznačen na obrázku 2.3.



Obrázek 2.3: Ukázka teselace jednoho patche.

2.1.2 Instanciace

Naivní vykreslení scény vytvoří pro každý objekt jeden, resp., pokud je jeden objekt tvořený více částmi (anglicky se taková část nazývá mesh a v textu budu nadále užívat jen tohoto anglického originálu), více bufferů. Každý jednotlivý mesh je vykreslován pomocí jednoho volání kreslícího příkazu. Se zvyšujícím se počtem modelů vzniká značné zpoždění způsobené nastavováním GPU.

Herní scéna se sice na první pohled může jevit jako sada unikátních objektů, avšak při podrobnějším zkoumání lze nalézt několik skupin, jejichž objekty jsou si podobné. Takovouto skupinu mohou tvořit např. smrky. V lese je sice plno různých smrků, ale navzájem jsou si velice podobné a často se liší jen pozicí. Případy s množstvím stejných objektů lišících se pouze drobnými detaily lze řešit instanciací, jejíž příkaz je ukázán v kódu 2.4. Instanciace pro jeden objekt vykreslí předem zvolený počet kopií.

Při instančním kreslení musí být všechna data přítomná na GPU. U menšího počtu objektů lze data řešit pomocí uniformních proměnných. Avšak uniformní proměnné mají omezenou velikost, proto při větším počtu pomocných dat je potřeba vytvořit nový pomocný buffer podobný bufferu pro vrcholy. Aby se z bufferu četly pro každý vrchol jedné instance stejná data, je nutné nastavit frekvenci čtení pomocí příkazu `glAttribDivisor(n)`, jenž určuje po kolika n instancích dojde k načtení nových dat. Defaultní hodnota je 0, což znamená jedno čtení pro každý vrchol.

```
1        void glDrawArraysInstanced( GLenum mode,
2        GLint first,
3        GLsizei count,
4        GLsizei primcount);
```

Program 2.4: Příkaz pro instanciaci.

2.1.3 Nepřímé vykreslování

Nepřímé vykreslování zvyšuje nezávislost GPU rozšířením instanciace na různé objekty, a tak umožňuje vykreslit více různých meshů bez zásahu CPU. Struktura příkazu vykreslující jeden objekt je uvedena v kódu 2.5 a příkaz pro nepřímé vykreslování ukazuje kód 2.6. Jednotlivé meshe se sdružují do jedné sady bufferu, kde jsou identifikovány indexem počátečního vrcholu, indexem počáteční indicie a celkovým počtem vrcholů. Následné vykreslovaní je řízeno grafickou kartou, jenž zpracovává kreslící příkazy uložené ve zvláštním bufferu příkazů.

```
1    typedef struct {
2        GLuint vertexCount;
3        GLuint instanceCount;
4        GLuint firstIndex;
5        GLint baseVertex;
6        GLuint baseInstance;
7    } DrawElementsIndirectCommand;
```

Program 2.5: Struktura příkazu, který je zpracováván grafickou kartou a vykreslí jeden mesh.

```
1    void glMultiDrawElementsIndirect(   GLenum mode,
2                                        GLenum type,
3                                        const void *indirect,
4                                        GLsizei drawcount,
5                                        GLsizei stride);
```

Program 2.6: Příkaz nepřímého vykreslování.

Kromě pozitivních přínosů v podobě rychlejšího vykreslování, přináší nepřímé vykreslování problémy, které se z určitého úhlu pohledu mohou jevit jako jeho negativní vlastnost. Jelikož vše probíhá zcela autonomně na GPU, nelze měnit stav vykreslovacího řetězce. Změnou stavu se rozumí změna bufferů, textur, průhlednost atd. Změnu bufferů s vrcholy a indiciemi již řeší kreslící příkaz (kód 2.6).

OpenGL požaduje v rámci kreslení navázání každé používané textury na určitou texturovací jednotku a navazování je změna stavu. Navazování není nutné při použití bindless textur, dostupných v době psaní této práce pouze jako ARB (Architecture Review Board) rozšíření, tudíž nemusí být k dispozici na každé GPU. Pokud tedy nejsou k dispozici, je nutno spojit všechny textury do jedné sady objektů a ten následně navázat na texturovací jednotku.

Spojení lze realizovat pomocí uniformních polí textur. Uniformní pole mapují každý svůj záznam na jednu texturovací jednotku, z čehož vyplývají i stejná omezení. Omezením je celkový počet textur, protože OpenGL specifikace umožňuje pouze 16-32 navazovacích bodů pro jednu etapu shaderů. Tento limit lze obejít použitím pole textur.

Pole textur je speciální OpenGL textura, která má strukturu jako klasické pole. Jednotlivé záznamy mohou obsahovat různá data, ale mají stejné rozměry, počet MIP úrovní a další vlastnosti. Pole se vytváří pomocí příkazu `GL_TEXTURE_nD_ARRAY` (n znamená počet

dimenzí, pro klasické textury n = 2). Nová textura je přidaná na nový index. Pole textur se v shaderech navazuje pouze na jeden navazovací bod a konkrétní textura je vybraná pomocí indexu do pole. Pole textur teoreticky umožňuje neomezené množství záznamů a jeho velikost tak může přesáhnout kapacitu grafické paměti.

2.1.4 Buffer objects (BO)

Buffer object, volně přeloženo jako paměťový objekt, je OpenGL objekt, jenž ukládá pole obsahující neformátovanou paměť alokovanou OpenGL kontextem (též označován pouze jako GPU). BO mohou být využity k ukládaní informací o vrcholech, indiciích a řadě dalších věcí.

Uniform buffer object (UBO)

V rámci vývoje dříve či později nastane okamžik, kdy se kopíruje spousta kódu jen proto, aby měl právě vykonávaný program aktuální uniformní data. Tento postup je nejen únavný, ale přináší s sebou velké riziko zavlečení chyb, zhoršenou udržovatelnost programu a v neposlední řadě i celkové zpomalení aplikace. Klasické buffery lze jednoduše sdílet mezi jednotlivými programy, uniformní proměnné pouze pokud jsou v uniformním bufferu.

Uniformní buffery byly představeny společně s OpenGL 3.1. Tyto buffery jsou paměťové oblasti alokované v grafické paměti a z pohledu shaderů se jedná o paměť přístupnou jen ke čtení. V GLSL shaderu se k UBO přistupuje pomocí bloku rozhraní. Kód 2.7 ukazuje blok rozhraní pro jednoduchý UBO. Tento UBO je dále potřeba v aplikaci připojit přes příkaz `glBindBufferBase(GL_UNIFORM_BUFFER, binding_point_index, ubo);`. Pokud se v průběhu aplikace upraví hodnoty v UBO, změna se automaticky projeví ve všech shader programech obsahujících UBO na daném navazovacím bodu. OpenGL limity pro UBO je možno vidět v tabulce 2.1. Kromě OpenGL limitů uvedených v tabulce 2.1 jsou taky jistá omezení u polí. Každé pole musí mít přesně specifikovanou velikost za pomoci výrazu vyhodnotitelného v době překladu.

```
1    layout (binding=5, std140) uniform basicUBO
2    {
3        mat4 Matrices[4];
4        vec4 Wind;
5        float Time;
6    };
```

Program 2.7: Ukázka jednoduchého UBO v shader programu.

Shader storage buffer object (SSBO)

SSBO si lze představit jako UBO, ze kterého lze nejen číst, ale i zapisovat. Také má mnohem větší velikost – u GTX 1060 to jsou 2 GB. Dále není potřeba přesná specifikace velikosti polí, ale lze použít pole, jehož velikost se bude za běhu programu dynamicky měnit. Ovšem také lze nalézt pár negativních vlastností. Předně to je nutnost použití OpenGL 4.3 (na rozdíl od 3.1 pro UBO) a také o něco pomalejší čtení než z UBO, jehož hodnoty jsou nahrávány do paměti konstant.

	Tesla C2075	GTX 1060 6 GB	R9 280	HD6630M
GL_MAX_UNIFORM _BUFFER_BINDINGS	84	84	90	90
GL_MAX_UNIFORM _BLOCK_SIZE	65536 B	65536 B	65536 B	65536 B
GL_MAX_VERTEX _UNIFORM_BLOCKS	14	14	15	15
GL_MAX_FRAGMENT _UNIFORM_BLOCKS	14	14	15	15
GL_MAX_GEOMETRY _UNIFORM_BLOCKS	14	14	15	15

Tabulka 2.1: Tabulka ukazuje maximální hodnoty pro UBO u několika vybraných grafických karet.

SSBO nacházejí využití nejčastěji u nepřímého vykreslování jako uložiště dat pro dynamicky načítané modely. Dále, protože SSBO umožňují zápis v shader programu, nabízí se jako výstup pro data vypočítaná za pomoci GPU.

2.2 Perlinův šum

Základem každého šumu je generátor náhodných čísel. Aby byl využitelný v počítačové grafice, bývá častým požadavkem určitá vzájemná souvislost generovaných čísel, což popírá základní myšlenku generátorů náhodných čísel, jelikož ty se snaží závislosti odstraňovat. Z tohoto důvodu bývají výstupy generátoru náhodných čísel využity jako vstupy jiné funkce, která dává konečnou šumovou hodnotu. Jednou z možností zvýšení korelace je použití goniometrických funkcí, bohužel tyto funkce vytvářejí vlnové vzory.

Nalezením vhodné funkce použitelné pro počítačovou grafiku a filmový průmysl, jež by splňovala všechny podmínky, se zabýval Ken Perlin. Ten v roce 1985 uveřejnil vlastní implementaci šumu pojmenovaném po sobě a za niž obdržel v roce 1997 cenu akademie (Oscar). Funkci zveřejnil v článku: An Image Synthesizer [14]. Perlinův šum je nejpoužívanějším druhem šumu v počítačové grafice a to i přesto, že v roce 2001 přišel Perlin s vylepšeným šumem nazvaným simplex noise publikovaným v článku Improving Noise [15], který snížil složitost z $\Theta(2^N)$ na $\Theta(N^2)$ a zároveň došlo k vylepšení jeho spojitosti.

2.3 Dělení prostoru

Aby byla zachována efektivní práce s velkým počtem objektů, je nutno zvolit lepší přístup než sekvenční vyhledávání v seznamu objektů. Zvolením vhodného způsobu indexování se lze dostat pod lineární složitost. Následující část se bude věnovat jednotlivým indexovacím metodám. Podkladem byl článek Geometric Data Structures for Computer Graphics [22].

2.3.1 Kvandrantový strom

Kvadrantový strom (anglicky quadtree) je kořenový strom, který je buďto prázdný, nebo se sestává z jednoho kořene a čtyř kvadrantových podstromů. Jedná se o rozšíření binárního stromu pro dvousouřadná data.

Hodnoty se obvykle ukládají pouze do listových uzlů. Potom, jestliže se ukládá množina n bodů, má strom $\Theta(\frac{4^{d+1}-1}{3})$ uzlů a hloubka $d = \lceil \log_4(n) \rceil + 1$. Vyhledávací čas $\Theta(\log_4(n))$.

2.3.2 Oktávový strom

Oktávový strom (anglicky octree) je kořenový strom, který je buďto prázdný, nebo se sestává z jednoho kořene a osmi oktávových podstromů. Jedná se o rozšíření kvadrantového stromu pro třisouřadná data.

Hodnoty se obvykle ukládají pouze do listových uzlů. Potom, jestliže se ukládá množina n, bodů má strom $\Theta(\frac{8^{d+1}-1}{7})$ uzlů a hloubka $d = \lceil \log_8(n) \rceil + 1$. Vyhledávací čas $\Theta(\log_8(n))$.

2.3.3 K-d strom

K-d strom nebo také k-dimenzionální strom je datová struktura využívána k uspořádávání množiny bodů v prostoru o k dimenzích. Jedná se o binární vyhledávácí strom. K-d stromy nacházejí uplatnění především u problému vyhledávání v rozsahu a hledání nejbližších sousedů.

Každá úroveň k-d stromu dělí všechny potomky podle specifické dimenze. K dělení se využívají nadroviny kolmé na odpovídající osu. Na úrovni kořene budou data rozdělena na základě první dimenze, tzn. data, jejichž souřadnice v první dimenzi je menší, resp. větší, budou v levém, resp. pravém podstromu. Hodnota, jež slouží jako kořen pro aktuální strom, je medián bodů v příslušném intervalu vzniklého rozdělením předchozího intervalu v dané dimenzi. Každá nižší úroveň se dělí podle další dimenze a po vyčerpání všech zbývajících se začíná zase podle první. Optimálnější variantou může být místo dělení na základě pravidelného střídání os, dělení dle osy, v níž mají data největší rozptyl. Konstrukce probíhá tak dlouho, dokud všechny části neobsahují pouze jediný prvek.

Časová složitost vytvoření k-d stromu je $\Theta(n \cdot \log_2(n))$, což je způsobené především algoritmem nalezení mediánu, který má složitost $\Theta(n)$. Prostorová složitost je $\Theta(n)$ a nalezení prvků trvá $\Theta(\log_2(n))$. Čas nutný k nalezení elementů je horší než u kvadrantového, resp. oktávového stromu, ale při dělení podle osy s největším rozptylem mohou být celkové vlastnosti mnohem lepší.

2.3.4 Bounding volume hierarchy (BVH)

BVH označuje stromovou strukturu nad množinou geometrických objektů. Všechny objekty jsou obaleny ohraničujícími tělesy, jež tvoří listové uzly stromu. Uzly jsou seskupeny do malé množiny a následně opět obaleny tělesem ohraničujícím veškeré menší ohraničují tělesa. Takto rekurzivně lze vytvořit celý strom. Kořenový uzel je obalujícím tělesem pro všechny objekty scény. BVH bývají využívány jako podpůrné struktury pro efektivní operace nad sadou geometrických objektů např. sledování paprsků nebo detekce kolizí, jelikož průnik lze vyvrátit otestováním rodičovských uzlů.

Výběr vhodného ohraničujícího tělesa je kompromisem mezi co možná výpočetně nejjednodušším tělesem, co nejtěsnějším obalením cílového objektu a v neposlední řadě taky paměťovou náročností. V praxi využívanými obalovými tělesy jsou: koule, kapsle, konvexní obálka, ohraničující boxy a mnohé další. Obecně nejpoužívanějším tělesem je AABB.

Axis-aligned bounding box (AABB) je specializace ohraničujícího boxu, kdy jeho hrany jsou rovnoběžné s osami prostoru. AABB je možno reprezentovat jen pomocí dvou bodů, a to minimálního a maximálního vrcholu, z čehož vyplývá i jeho vytváření zjišťováním minimální a maximální hodnoty souřadnic v objektu. Při manipulaci s objektem pomocí

transformací nezachovávajících rotaci je nutno přepočítat AABB. Výpočet postačuje dělat jen ze všech transformovaných rohových bodů zjišťováním nového minima a maxima.

BVH lze sestavit pomocí tří způsobů:

- **Metodou shora dolů** – vstupní množina je dělena na podčásti tak dlouho, dokud nezbydou uzly obsahující pouze jedno primitivum. Metoda shora dolů je jednoduchá na implementaci, konstrukce stromu bývá rychlá, ale výsledný strom nemusí být optimální.
- **Metoda zdola nahoru** – vstupní množinu tvoří uzly obsahující jednotlivá primitiva. Tyto uzly se spojují, dokud nevznikne jeden kořenový uzel. Metoda zdola nahoru je implementačně složitější, avšak dosahuje optimálnějšího stromu.
- **Metoda vkládání** - na rozdíl od předchozích metod, použitelných jen v rámci předzpracování, tato metoda pracuje interaktivně. Počátkem je prázdný strom, do kterého se vkládají nově příchozí primitiva. Lokace pro vložení se vybírá podle cenového ohodnocení všech možných míst pro vložení.

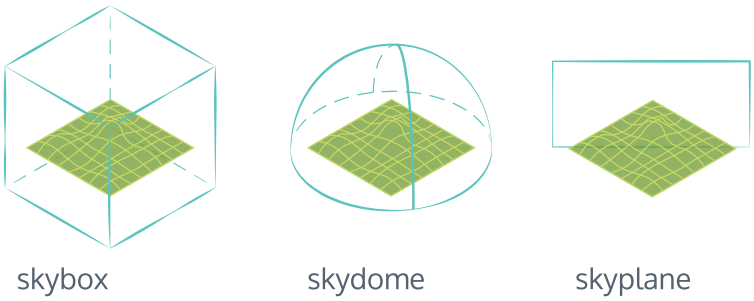
Prostorová složitost koresponduje s použitou primitivní strukturou. Složitost vyhledávání může být ovlivněna i náročností vlastního porovnávání u uložených dat.

2.4 Generování oblohy

Typický snímek herní scény odehrávající se v otevřené krajině obsahuje z velké části i oblohu. Vykreslená obloha musí v hráči budit dojem objektu nalézajícího se daleko nad ním. Dalším častým požadavkem je značná podobnost s reálnou oblohou. Vykreslovaní oblohy je možno rozdělit na dvě části: vykreslení geometrie a její následné obarvení.

2.4.1 Geometrie oblohy

Aby obloha vzbuzovala dojem vzdáleného objektu, tak se často používá geometrie, která obaluje celý herní svět. Pro její vytvoření se obvykle užívá tří různých způsobů, jež jsou ilustrovány obrázkem 2.8. Nejjednodušeji lze oblohu vytvořit za pomocí skyboxu.



Obrázek 2.8: Metody pro vykreslování oblohy řazeny dle složitosti. Vlevo je nejjednodušší.

Skybox obaluje svět do rovnoběžnostěnu, jehož kterékoliv dvě sousední hrany svírají mezi sebou pravý úhel. Vykreslit jej lze za použití šesti čtverců nebo dvanácti trojúhelníků. Kromě jednoduché geometrie přináší skybox triviální vytvoření a aplikaci textury, avšak

vznikají zde artefakty na hranách, kde často bývají viditelné spoje. Artefakty vznikající na hranách řeší skydome.

Skydome znamená vytvoření polokoule, jež se tyčí nad herní plochou. Polokouli lze vytvořit použitím matematického předpisu 2.1, kde r – poloměr, u – zeměpisná délka (longitude), $u \in \langle 0, 2\pi)$, v – zeměpisná šířka (latitude), $v \in \langle 0, \pi \rangle$.

$$F(u, v) = [\cos(u) \cdot \sin(v) \cdot r, \cos(v) \cdot r, \sin(u) \cdot \sin(v) \cdot r] \tag{2.1}$$

Takto vytvořena polokoule bude mít směrem k pólu zvyšující se počet pomocných vrcholů, což může, při potřebě zachování určitého celkového množství vrcholů, vyvolat artefakty nedostatečné kulatosti v nižších částech oblohy, proto je výhodnější použit teselaci čtyřstěnu, jehož povrch v každé části tvoří stejně velké trojúhelníky. Textury pro použití se skydome musí být speciálně vytvořené pro mapovaní na kulaté povrchy a jejich vytvoření je mnohem náročnější. Také je složitější samotné mapovaní textury.

Poslední možností je využití roviny vyplňující obrazovku – skyplane. Tato rovina se kreslí na zadní ořezovou stěnu pohledového jehlanu. Skyplane nalézá uplatnění v situacích, kdy se nepoužívá textura, ale barva vykreslované oblohy je dynamicky generovaná.

2.4.2 Obarvení oblohy

Obarvení oblohy pomocí textur spočívá pouze v namapování textury na zvolenou geometrii. Pro dosažení dynamické oblohy s pravidelným denním cyklem je výhodnější využít procedurálního generování oblohy. Následujících několik sekcí se bude zabývat fyzikální simulací oblohy. Nejprve budou představeny nutné koncepty, které budou v sekci Výpočet barvy využity pro výpočet cílové barvy. Jednotlivé koncepty byly nastudovány z knihy GPU Gems 2 [16, kapitola 16] a článků Precomputed Atmospheric Scattering [2] a A Practical Analytic Model for Daylight [17].

Koeficienty rozptylu - β

Sluneční svit je při průchodu atmosférou ovlivňován částicemi v ní se nacházejícími. Koeficienty rozptylu udávají vliv prostředí na specifické vlnové délky světla. První model pro popis onoho jevu byl objeven roku 1871 britským fyzikem lordem Rayleightem. Bohužel tento model nebyl dokonalý, a tak se snažili další fyzici přijít s univerzálnějším modelem. V roce 1907 německý fyzik Gustav Mie vyřešením Maxwellových rovnic vytvořil vlastní model. V praxi se používají oba, každý však pro jiný typ částic.

Rayleightův rozptyl (rovnice 2.2 z článku [12, strany 2-3]) se uplatňuje pouze u částic, jejichž velikost je minimálně desetkrát menší než vlnová délka počítaného světla. Toto kritérium obvykle splňují jen atmosférické plyny (kyslík, dusík atd.). Jak lze vidět z rovnice 2.2, jedná se o jev velice závislý na vlnové délce, na které je nepřímo úměrný. Vlnové délky světla ve viditelném spektru se pohybují od 380 po 780 nm, RGB barevný model lze převést na vlnové délky – 680nm, 550nm a 440nm. Rayleightův rozptyl nejvíce ovlivňuje modré světlo, a tedy vysvětluje modrou oblohu, již lze vidět za slunného dne. Jednotlivé členy rovnice jsou – N objemová hustota částic na úrovni moře, n index lomu vzduchu, h aktuální výška a H (scale height) označuje nárůst výšky, pro kterou atmosférický tlak klesá exponenciálně s faktorem e. Pro H se obvykle používá hodnota 8000 m. V nulové výšce se výpočet koeficientů redukuje o násobení exponenciální funkci a z toho důvodu je možné rozptylové koeficienty předpočítat pro nulovou výšku a pro požadovanou výšku pouze upravit patřičnou exponenciálou.

$$\beta(h,\lambda) = \frac{8\pi^3 \cdot (n^2-1)^2}{3 \cdot N \cdot \lambda^4} e^{\left(\frac{-h}{H}\right)} \tag{2.2}$$

Mieův rozptyl (rovnice 2.3) je podobný Rayleightovu, ale používá se u částic, které se nalézají v nižších hloubkách atmosféry. Jejich velikost bývá větší než vlnová délka světla. V praxi není potřeba počítat hodnoty pro všechny vlnové délky, postačuje jedna hodnota, což je konstanta změřená na úrovni moře - $\beta_0 = 210 \cdot 10^{-5} m^{-1}$. H faktor se nastavuje na 1200 m.

$$\beta(h) = \beta_0 \cdot e^{\left(\frac{-h}{H}\right)} \tag{2.3}$$

Fázová funkce

Fázová funkce popisuje množství světla, které je rozptýleno směrem k pozorovateli. Pro její výpočet je zapotřebí znát úhel Θ, jenž svírá vektor od pozorovatele k danému bodu a vektor z bodu směrem ke slunci. Konstanta g ovlivňuje symetrii rozptylu mezi dopředným a zpětným směrem (zpět ke světelnému zdroji). Pokud $g = 0$ je rozptyl v obou směrech symetrický. Je-li g kladné, resp. záporné, pak zvětšuje rozptyl v zpětném, resp. dopředném směru. Hodnota g se obvykle volí z intervalu $(0,76;1)$ [16, GPU Gems2 - kapitola 16]. Jak již bylo zmíněno výše, existují dva modely pro dva typy částic. Rayleightův rozptyl se počítá podle rovnice 2.4 a Mieův rovnice 2.5. Obě rovnice jsou převzaté z článku: Precomputed Atmospheric Scattering [2, kapitola 2.1]. Je také možné pro oba rozptyly použít pouze rovnici 2.5, kdy při počítaní Rayleightova rozptylu se nastaví konstanta g na nulu (takto postupovali např. v knize GPU Gems2 [16, kapitola 16]).

$$Phase_{Rayleigh}(\Theta) = \frac{3 \cdot (1+\cos^2(\Theta))}{16\pi} \tag{2.4}$$

$$Phase_{Mie}(\Theta, g) = \frac{3 \cdot (1-g^2) \cdot (1+\cos^2(\Theta))}{8\pi \cdot (2+g^2) \cdot (1+g^2+2g \cdot \cos(\Theta))^{\frac{3}{2}}} \tag{2.5}$$

Průsečík vektoru s koulí

Dynamicky generovaná obloha většinou neobsahuje žádnou geometrii, tudíž nejsou známy jednotlivé souřadnice jejich bodů. Zároveň se předpokládá, že taková obloha je kulovitého tvaru. Tato premisa dovoluje vypočíst body oblohy jako body na povrchu koule, což je průsečík vektoru a povrchu koule.

Analytické řešení problému vychází z rovnice polopřímky v parametrickém tvaru 2.6, kdy P – průsečík, O – počáteční bod polopřímky, $\vec{D}$ – směrový vektor jehož průsečík je počítán a t – parametr určují vzdálenost bodu od počátku O. Dále je zapotřebí znát rovnici koule v počátku souřadnicového systému: $x^2+y^2+z^2 = R^2$. Substitucí jednotlivých souřadnic za bod lze rovnici koule upravit na 2.7. Po dosazení rovnice polopřímky do upravené rovnice koule vzniká tvar 2.8 po menší úpravě 2.9.

$$P = O + t\vec{D} \tag{2.6}$$

$$P^2 - R^2 = 0 \tag{2.7}$$

$$(O + t\vec{D})^2 - R^2 = 0 \tag{2.8}$$

$$O^2 + 2Ot\vec{D} + t^2\vec{D^2} - R^2 = 0 \tag{2.9}$$

Zápis 2.9 není nic jiného než kvadratická rovnice $f(t) = at^2 + bt + c$, kdy $a = \vec{D}^2$, $b = 2O\vec{D}$ a $c = O^2 - R^2$. Po vyjádření diskriminantu 2.10 lze dle rovnice 2.11 jednoduše spočítat parametry t.

$$\Delta = (2O\vec{D})^2 - 4\vec{D^2} \cdot (O^2 - R^2) \tag{2.10}$$

$$t_{1,2} = \frac{-2O\vec{D} \pm \sqrt{\Delta}}{2\vec{D^2}} \tag{2.11}$$

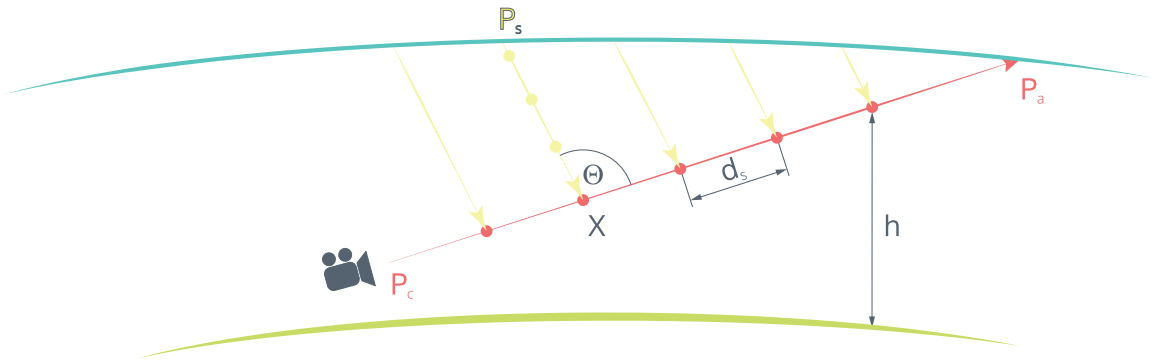
U obecného výpočtu průsečíku s koulí mohou nastat tyto stavy:

- $\Delta > 0$ polopřímka je sečna a má dva průsečíky
- $\Delta = 0$ jedná se o tečnu a jeden průsečík (dotykový bod)
- $\Delta < 0$ polopřímka danou koulí neprochází

U světů, kde je kamera umístěna na povrchu herního světa, bude počátek polopřímky vždy ve vnitřním prostoru polokoule, a tak lze bez újmy na obecnosti očekávat, že diskriminant bude vždy kladný. Dále, jelikož předpoklad byla polopřímka a bod na ní ležící, splňují tuto podmínku pouze kladné hodnoty odmocniny z diskriminantu.

Výpočet barvy

Nyní již je možno přistoupit k výpočtu výsledné barvy jednoho konkrétního pixelu obrazovky reprezentující jeden konkrétní bod oblohy. Celý postup je ukázán na obrázku 2.9.



Obrázek 2.9: Výpočet výsledné barvy jednoho pixelu na obloze za použití numerické integrace paprsku vyslaného k danému pixelu a zároveň i integrací slunečních paprsků působících v jednotlivých bodech.

Neboť se Slunce nachází ve velké vzdálenosti od Země, přešlo již jeho záření z kulové vlny na vlnu rovinnou a jednotlivé dopadající paprsky jsou vzájemně rovnoběžné. Barva bodu oblohy se vypočítá integrací paprsku vyslaného k němu z kamery dle rovnice 2.12. Integrace probíhá od bodu P_C – kamera, po bod P_A - obloha. Jedná se tedy o klasické sledování paprsku.

$$Color(P_c, P_a) = \int_{P_c}^{P_a} T_R(P_c, X) \cdot T_R(X, P_s) \cdot T_M(P_c, X) \cdot T_M(X, P_s) \cdot e^{-\frac{h}{H}} dr \tag{2.12}$$

Pozice kamery je triviálně známá, bod oblohy se získává výpočtem průsečíku vektoru a koule představující atmosféru. Protože celý výpočet probíhá ve screen space, resp. view space, tak se směrový vektor $\vec{D}$ získává z bodu P, jenž se nachází na zadní ořezové stěně pohledového jehlanu, vynásobením inverzní projekční, resp. view, maticí a odečtením pozice kamery. Důležitá je následná normalizace vektoru $\vec{D}$. Písmeno T v rovnici 2.12 označuje funkci transmitance neboli množství světla, které projde z bodu A do bodu B, její výpočet popisuje rovnice 2.13. Neznámá β v této rovnici označuje koeficienty rozptylu upravené pomocí rovnice 2.14 pro aktuální výšku.

$$T(P_a, P_b) = e^{-\sum_{P_a}^{P_b} \beta(h) d_s} \tag{2.13}$$

$$\beta_s(h) = \beta_s(0) \cdot e^{-\frac{h}{H}} \tag{2.14}$$

Dolní indexy transmitancí použitých v rovnici 2.12 představují transmitance pro Rayleightův a Mieův rozptyl. Integraci postačuje vyjádřit numericky jako součet hodnot v bodech ležících na vektoru směrem k obloze a rozložit ji tak na několik podintegrálů pro každý bod. V každém bodě jsou počítány transmitance z počátku k danému bodu a ze Slunce k danému bodu. Součin transmitancí je násoben exponenciální funkcí poměru výšky bodu a poměrné výšky pro daný rozptyl.

Příspěvky jednotlivých rozptylů se dosadí do rovnice 3 pro výpočet finální barvy, kdy I je celková intenzita Slunce a β koeficienty rozptylu. Konečná barva má často vysoký dynamický rozsah, a proto je vhodné provést namapování na nižší dynamický rozsah, který je bezproblémově zobrazitelný běžným monitorem. Přebarvení se provádí pomocí Reinhardovy mapovací funkce s gama korekcí (rovnice 2.15). Hodnota korekce se obvykle volí 2,2.

$$C_{LDR} = \left(\frac{C_{HDR}}{C_{HDR} + 1} \right)^{\frac{1}{\gamma}} \tag{2.15}$$

2.4.3 Mraky

Dalším stupněm k větší realističnosti jsou mraky, jelikož i za velmi slunného dne se na obloze vždy nalézá nějaká jejich forma. Oblak je útvar nepravidelného tvaru nacházejícího se ve vyšších částech atmosféry. Oblačnost je složena především z vody a malých částic ledu. Kromě vody obsahují i prach a zplodiny vznikající průmyslovou výrobou [13, kapitola 2.3, strana 5].

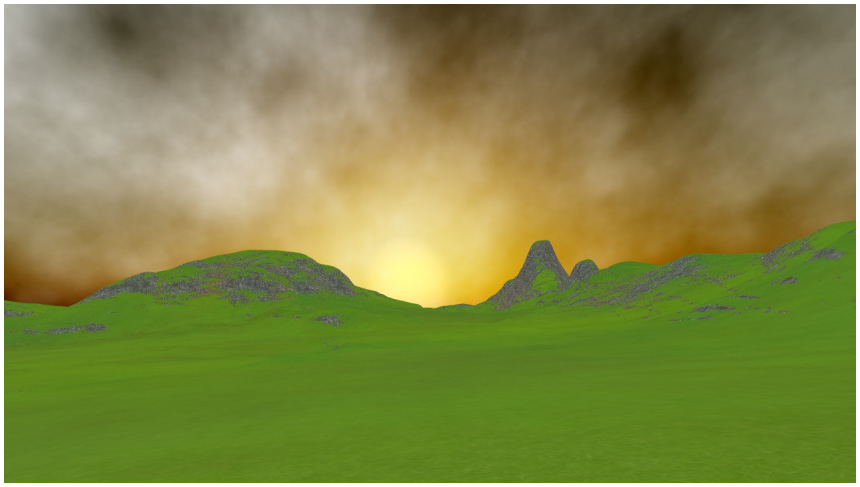
Mraky stojí na pomezí geometrie a pouhého upravení barvy. Při použití textury pro obarvení oblohy bývají mraky již její součástí. Situace se komplikuje v okamžiku, kdy se hráč může ocitnout v jejich výšce, jelikož hráč očekává prostorové mraky, které ho budou obklopovat. Jedním z řešení vzniknuvší situace je použití 3D modelů mraků. Kromě textury, nebo 3D modelů, je možnost postupovat obdobně jako při výpočtu barvy oblohy.

Vykreslení mraků často znamená jen ovlivnění bodu oblohy barvou mraků. Ovlivnění je charakterizováno změnou optické hustoty, a tudíž je možné ho přímo využít v rovnici 2.12 k upravení barvy. Takto vytvořené mraky jsou velmi realistické, avšak jejich praktické

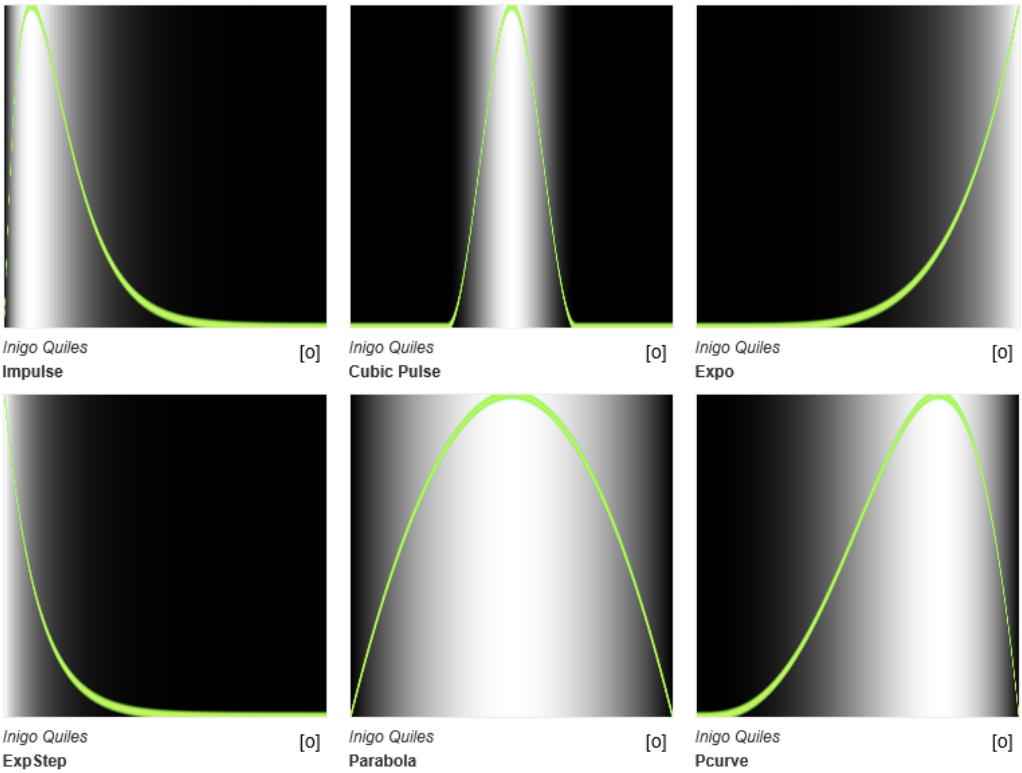
nasazení je sporné, neboť pro interakci s jinými objekty je zapotřebí znova provádět celé složité výpočty. Proto bývá mnohdy výhodnější vytvořit mrakům vlastní samostatnou vrstvu (v případě 3D mraků těleso), na kterou se budou mraky vykreslovat.

Ať už se mraky fyzikálně simulují, nebo jen jednoduše vykreslují, je nutno určit s jakou hustotou se mrak v daném místě nachází (pokud tam není, intenzita je nulová). Za tímto účelem se používají šumové funkce např. Perlinův šum. Tyto funkce existují ve variantách pro různé dimenze. U časově proměnlivých mraků je potřeba volit alespoň 3D verzi při vykreslování plochy, 4D pro objemové mraky. Použití 4D verze u plochy vytváří efekt připomínající více kouř než mlhu, viz ilustrace 2.10. K dosažení větší důvěryhodnosti bývá opět vhodné použiti HDR mapovaní (rovnice 2.15) a také prahování nebo tváření za pomocí jiných funkcí, jejich příklad uvádí obrázek 2.11.

Skutečné mraky vykazují určitou míru fraktality, kterou lze napodobit použitím Brownova frakčního pohybu. Prakticky se jedná o skládání šumu z několika oktáv. Oktáva má zde stejný význam jako v hudební teorii označujíce tón s dvojnásobnou frekvencí. Někdy není frekvence násobena přesně dvěma, ale číslem těsně oscilujícím kolem čísla 2. Kromě frekvence je upravována i amplituda. Obvykle se použije převrácená hodnota násobku základní frekvence vynásobeného základní amplitudou, ale opět lze frekvenci upravovat různě a získávat tak různé výsledky. Základní amplituda je obvykle 1. Pro kombinaci oktáv postačuje vážený součet jejich šumových hodnot, váhou je amplituda.



Obrázek 2.10: Generování jedné vrstvy mraků za pomocí 4D Simplex šumu. Výsledný efekt více připomíná kouř než mraky.



Obrázek 2.11: Příklady možných tvarovacích funkcí, které mohou sloužit k měkkému prahovaní, kdy můžeme měnit rozložení vstupních dat. Převzato z https://thebookofshaders.com/05/ [5.12.2016]

2.5 Terén

Terén patří, vedle oblohy, k základní části každé hry odehrávající se v běžné otevřené krajině. Zároveň ho lze považovat i za jistý kontejner pro ostatní objekty, jelikož se často používají vlastnosti terénu pro úpravu vlastností objektů. Vykreslení terénu se může jevit jako zcela triviální úloha nezasluhující si větší prostor, avšak s rostoucí komplexitou hry roste i velikost terénu. Kupříkladu hra Skyrim (rok 2011) má herní svět o přibližné velikosti 16 km^2, GTA 5 (vydání pro PC 2015) 49 km^2. Tato dvě čísla ilustrují náročnost herního terénu a zároveň umožňují vidět podobnost s geografickými informačními systémy (GIS).

V následujících podčástech budou postupně představeny techniky pro vytváření terénu, snížení datové náročnosti terénu a vlastní vykreslení terénu. Problematika byla nastudována primárně z článku DirectX 11 Terrain Tessellation od firmy Nvidia [3] a dále pak z přednášky od vývojářů herního enginu Frostbite přednášené na herní vývojářské konferenci (GDC) v San Franciscu ročník 2012 [20].

2.5.1 Modelování terénu

Při tvarování krajiny se lze vydat dvěma směry. První je zcela odlišný od způsobu, který používají GIS, a jedná se o procedurální generování krajiny. Velikou výhodou tohoto přístupu je malá, téměř nulová paměťová náročnost na vstupní data. Veškerý terén je popsaný matematickou funkcí, případně sadou funkcí. Vhodně zvolená funkce dokáže jednoduše vy-

tvořit jeskyně a další podobné prvky. Nevýhodou tohoto přístupu je menší kontrola nad výsledným tvarem krajiny a z toho vyplývající složitější návrh herního světa a také mohou nastat situace, kdy nelze vytvořit konkrétní krajinnou oblast. Princip generovaného světa je použit např. ve hře Minecraft nebo No man's sky.

Druhý způsob je již podobný GIS a jedná se o použití výškové mapy. Výšková mapa určuje pro každý bod terénu jeho výšku. Formulace pro každý bod není zcela přesná, jelikož záleží na vzorkování mapy, ale obecně to lze prohlásit. Výšková mapa neumožňuje vytvořit jeskyně a obecně vícevrstvé prvky. U oblasti obsahující jeskyně je zapotřebí použít jejich 3D modely anebo hloubkovou mapu. Hloubková mapa nachází využití také v situacích, jež působí destruktivně na dané prostředí. Ku příkladu takto byly vytvářeny krátery po výbuších ve hře Battlefield 3 [20]. Nevýhoda výškových map tkví v paměťové náročnosti vstupních dat. Naproti tomu obrovská výhoda se skrývá v možnosti modelovat reálné oblasti světa, jejichž data lze získat měřením např. pomocí lidaru.

Výše uvedené dělení ovšem není zcela striktní. V praxi se daleko častěji používá kombinace obou přístupů, kdy pro jemnější detaily krajiny lze použít funkcionálního generování. Funkce může sloužit také pro alteraci a následnou recyklaci výškových map za účelem rozbití vzorů při recyklaci map. Avšak ani jedna uvedená technika neřeší problémy, jež vznikají v hrách a ve své podstatě i v GIS při vykreslování rozsáhlých scén s vysokou úrovní detailů (levels of details, zkracováno jako LOD) bez ztráty efektivity a výkonnosti.

2.5.2 Dynamická úroveň detailů terénu (LOD)

V oblastech vzdálených od kamery je zbytečné pracovat s velkou granularitou, jelikož perspektivní projekce způsobí značný překryv u jednotlivých trojúhelníků. Tento překryv v lepším případě způsobí mrhání GPU zdroji, v tom horším může vytvořit nechtěný aliasing. Snížení granularity vzdálenějších částí lze dosáhnout využitím teselace, která zároveň sníží i počet nutných dat k přenesení na GPU.

Kus terénu blíže kamery potřebuje větší počet vrcholů, a tak je potřeba nastavit mu větší počet teselačních úrovní, a opačně pro vzdálený. Každý kus terénu se skládá z několika vrcholů. Souřadnice těchto vrcholů lze použít pro výpočet jejich vzdálenosti od kamery. Problém, jenž může vznikat, jsou díry na hranicích jednotlivých dlaždic. Tento problém je zapříčiněn nestejnou teselační úrovní sousedních dlaždic, kdy v daném místě vznikne nový bod pouze pro jednu z dvojice. Řešením je zajištění stejné teselační úrovně pro společnou hranu u obou dlaždic.

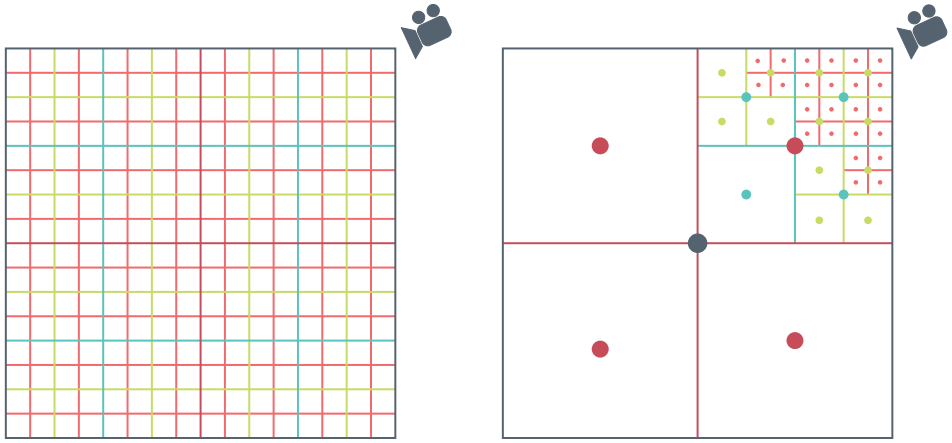
Výpočet vzdálenosti je nutno provádět pro každou hranu vstupní dlaždice a používají se dva způsoby:

- **Promítnutí hran** – koncové body dané hrany se promítnou do prostoru kamery. Jejich rozteč slouží k určení vzdálenosti dlaždice od kamery. Bohužel tento postup funguje dobře jen na hrany, které jsou kolmé na směrový vektor kamery. Čím menší bude úhel, tím ve větší vzdálenosti se hrana bude jevit a hrany rovnoběžné s vektorem kamery budou nekonečně daleko.
- **Promítnutí obalové koule** – kolem hrany je vytvořená opsaná koule a teprve ta se promítá do prostoru kamery. Vzdálenost určuje poloměr promítnuté koule.

2.5.3 Vykreslování terénu

Základní naivní přístup (nevyužívající teselaci ani jiné optimalizace) vykresluje velikou geometrii složenou z velkého množství vrcholů. S tím, jak roste velikost terénů, roste i množství

vrcholů nutných k reprezentaci požadovaných detailů a celkově se snižuje výkon. Pokročilejší přístup na GPU nahraje pouze inicializační sadu a další potřebné vrcholy generuje teselační jednotka.



Obrázek 2.12: Srovnání datové náročnosti terénu při použití uniformních a neuniformních dlažidc. Body ve středu dlaždic znamenají uzly v kvadrantovém stromu.

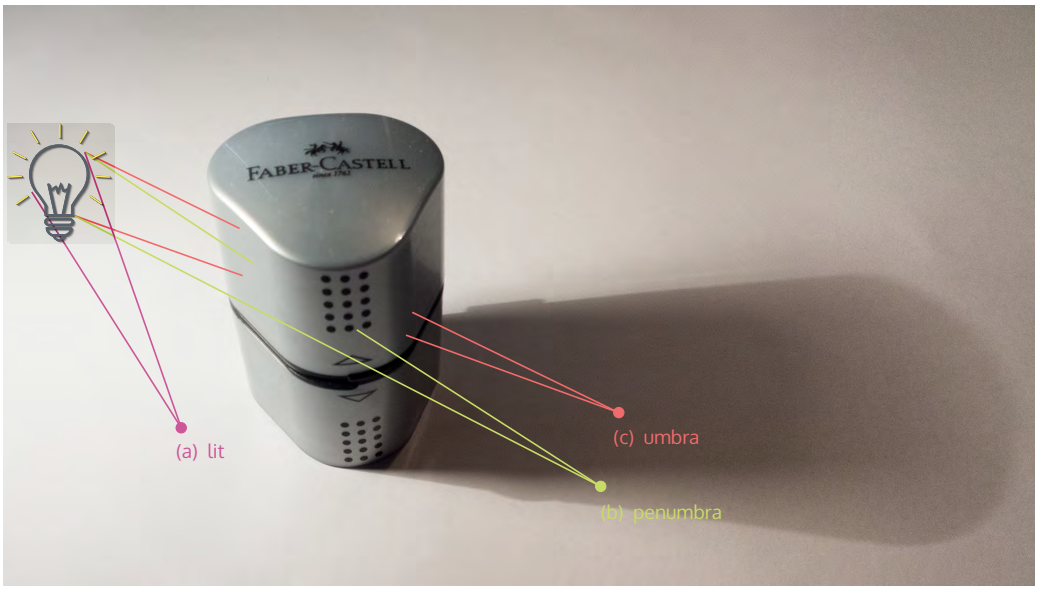
Koncept terénu uniformních dlaždic vytváří inicializační sadu z dlaždic o stejných rozměrech, rozmístěných po celém terénu. Jedná se o zcela přímočarý postup, který ale není škálovatelný. Jednak maximální úroveň teselace v OpenGL je 64, avšak i větší počet by pouze odsunul problém, protože s rostoucí velikostí roste i celkový počet dlaždic, jelikož je nutné zajistit, aby jejich rozměry dovolovaly potřebné detaily v oblastech blízko kamery. Lepší škálovatelnost umožňují dlaždice neuniformních rozměrů. Neuniformní dlaždice blíže kamery mají menší velikost, tak aby maximální teselační úroveň poskytovala požadovanou jemnost. Oba dva koncepty jsou ilustrovány obrázkem 2.12 .

V paměti bývá terén reprezentován pomocí kvadrantového stromu, jelikož se jedná o dvourozměrný objekt. Strom je obzvlášť výhodný při použití neuniformních dlaždic. Uzel ve stromu představuje střed dlaždice. Konstrukce probíhá rekurzivně od středu terénu. U uniformních dlaždic pokračuje dělení ve všech větvích, dokud není dosažena požadována velikost největších dlaždic. Analogicky dochází k vytváření stromu pro neuniformní dlaždice, zde ale probíhá dělení pouze ve větvích blízko kamery. Blízkost se zjišťuje vzdáleností daného kvadrantu od kamery. Vzniklý hierarchický strom umožňuje použití hierarchických výškových map pro lepší detaily terénu. Nevýhodou oproti stromu s uniformními dlaždicemi je nutnost přepočítávat dělení při každém pohybu kamery.

2.6 Stíny

Stín označuje tmavou oblast ve scéně způsobenou absencí světla, jež bylo pohlceno objektem vyskytujícím se mezi pozorovaným místem a zdrojem světla. Průmět stínu na plochu vytváří dvourozměrnou siluetu zastiňujícího tělesa. Přítomnost stínů zvyšuje realističnost scény a zjednodušuje určování prostorových vztahů vykreslovaných objektů.

Geometrie scény $\mathcal{S}$ je množina bodů p, které tvoří trojúhelníky objektů. Světelný zdroj $\mathcal{L}$ je množina bodů l formující rovinu světla. Světelný zdroj emituje fotony, a pro běžné prostředí se dá předpokládat, že dráha světla jsou rovné čáry. Tudíž pro zjištění zastínění



Obrázek 2.13: Fotografie objektu reálného světa použita k ilustraci druhů stínů. Bod je buď zastíněn (b,c), nebo plně osvětlen (a). Dále buď objekt blokuje světlo zcela (c), nebo jen částečně (b).

bodu p je potřeba vytvořit množinu $\mathcal{M}$, která obsahuje relace r, pro které platí, že bod p je v relaci s l pouze a jen tehdy, jestliže paprsek z l na dráze směrem k p neprotíná žádný objekt ve scéně. Velikost množiny $\mathcal{M}$ určuje typ stínů. Jednotlivé typy lze vidět na ilustraci 2.13 a jsou to tyto tři:

- **Plně osvětlený bod** – mezi bodem a zdrojem světla se nenachází žádný objekt, a proto veškeré paprsky vyslané ze světla směrem k pozorovanému bodu k němu doputují.
- **Penumbra** – měkký stín, je přechod mezi plně osvětlenou a zcela zastíněnou plochou. Pro body v této oblasti platí, že některé paprsky byly pohlceny, některé ne.
- **Umbra** – zcela zastíněný bod. Veškeré primární paprsky ze světla jsou blokované, bod je osvětlen pouze světlem odraženým ve scéně.

Podrobnější a především formálnější definici stínů lze nalézt v publikaci Real-Time Shadows [5].

Praktická implementace výpočtů stínů je náročnější, jelikož v současné době, na současných grafických kartách, neexistují algoritmy umožňující provádět jejich přesné výpočty v reálném čase. Pro aproximaci se používají bodová světla a nejpoužívanější metodou ve video hrách je shadow mapping [18] a její vylepšení cascade shadow mapping [4]. K výpočtu objemových stínů u objektů tvořených množstvím menších, např. kouř nebo vlasy, lze použít hloubkové stínové mapy s vrstvami [10] nebo konvoluční stínové mapy [1]. Aproximace bodovým zdrojem vytváří pouze tvrdé stíny, jelikož nemůže vznikat penumbra. Dosažení reálných měkkých stínů je možno jen použitím metod globální iluminace.

2.6.1 Shadow mapping

Shadow mapping je v současnosti jedna z nejrychlejších a nejefektivnějších metod pro vykreslování stínu v malých až středně velkých scénách. Zároveň patří k nejjednodušším metodám umožňujícím vytváření vysoce realistických stínů. Jedná se o dvouprůchodovou metodu, kdy v prvním průchodu jsou vytvořeny stínové mapy, které se v druhém průchodu využijí pro výpočet finálního osvětlení.

Stínová mapa je hloubková textura připojena k aktuálnímu frame buffer objektu a vytváří se vykreslením scény za pomocí kamery z pohledu světla. U bodových světel je kamera umístěna do stejných souřadnic, jež má dané světlo. Osa z kopíruje směr světla a využívá se perspektivní projekce. Směrové světelné zdroje, což je například Slunce, nemají definovanou pozici, proto se kamera pouze zarovná ve směru světla. Jelikož pro tyto zdroje platí, že jejich jednotlivé světelné paprsky jsou rovnoběžné, využívá se zde ortogonální projekce. Následně je scéna vykreslena a jsou zaznamenány hloubky objektů do hloubkové mapy. Barvu objektů není potřeba zaznamenávat.

Aplikace stínů se provádí v druhém průchodu, při kterém je kamera umístěna v klasické pozici např. pohled hráče. Pro každý vykreslovaný bod se provede porovnání jeho aktuální hloubky s hloubkou zaznamenanou při prvním průchodu. Pakliže nová hloubka není větší, objekt je plně osvětlen. Aby bylo možno zjistit hodnotu v hloubkové mapě je nutno vykreslovaný bod promítnout do prostoru světla.

Z principiálních důvodů nebude shadow mapping nikdy přesný. Algoritmus stínových map způsobuje několik artefaktů:

- **Shadow acne** – sebe zastínění, artefakt, jehož důvodem je diskrétní reprezentace hloubkové mapy. Projekce při obou průchodech určí stejnému fragmentu rozdílnou hloubku a fragment může zastínit sám sebe. Řešením je k porovnávané hodnotě přičíst určitou odchylku.

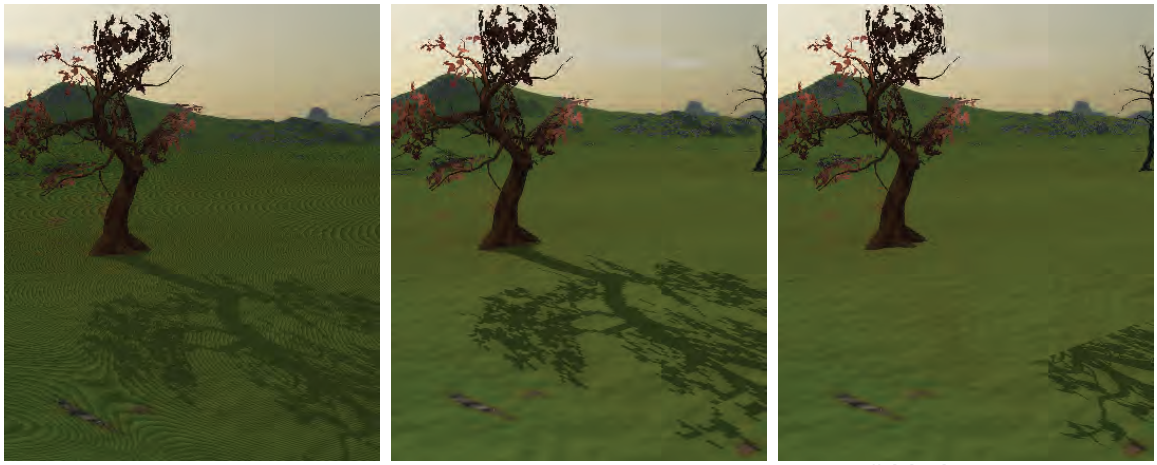
- **Aliasing** – je způsoben nedostatečným rozlišením hloubkové mapy. Projevuje se ostrým, zoubkovaným přechodem mezi osvětlenou oblastí a oblastí ve stínu. Lze řešit zvýšením rozlišení hloubkové mapy. Ostré hrany nevznikají, pokud jsou texely hloubkové mapy a pixely zobrazovacího zařízení zarovnány 1:1.

- **Chybějící stíny** – problém, který nastává u směrových světel, kdy projekční matice ořeže objekty, jež by měly vrhat stíny. Odstranění se provádí zvětšením projekce na celý rozsah pohledového jehlanu.

- **Peter Panning** – oddělený stín objektů. Nejvíce se projevuje na hranách, kde za objektem vzniká viditelný posun stínů. Na velikosti Peter Panningu se i přímo úměrně podílí odchylka používaná k potlačení sebe zastiňování.

Percentage-Closer Filtering (PCF)

PCF technika vznikla za účelem potlačení aliasingu, zároveň umožnuje vytvářet falešné měkké stíny. Při PCF se nevzorkuje pouze jeden pixel, který dává binární hodnotu, ale na základě okolních bodů se vypočte procentuální zastínění bodu. Velikost porovnávaného okolí ovlivňuje šířku oblasti penumbra stínů. Urychlení PCF je provedeno využitím HW akcelerace, umožňující provádět filtraci čtyř sousedních bodů.



Obrázek 2.14: Artefakty vznikající při použití shadow mappingu.

2.6.2 Cascaded Shadow Maps (CSM)

Nejtěžším problémem k překonání u stínových map bývá perspektivní alias. Tento problém se snaží řešit CSM, jež jsou v praxi nejlepším řešením a běžně používaným v moderních hrách. Základní koncept spočívá v rozdělení pohledového jehlanu do podoblastí. Myšlenka vychází z pozorování, že body v rozdílné vzdálenosti od pozorovatele potřebují rozdílnou hustotu vzorků, která s rostoucí vzdáleností klesá. Dělení se provádí podél z-osy. Toto umožní, aby objekty blíže ke kameře měly stíny vykreslené ve větším rozlišení než vzdálenější objekty. Zároveň absolutní rozlišení textury může být pro všechny oblasti stejné, mění se jen velikost jejich projekční plochy.

Každá nová oblast představuje vlastní pohledový jehlan. Jedna z možných technik dělení spočívá ve vypočtení intervalu od nuly do sto procent. Jednotlivé intervaly pak reprezentují blízkou a vzdálenou ořezovou plochu jako procentuální interval z-osy. Přepočítávání dělení pohledového jehlanu pro každý snímek způsobuje v praxi mihotající hrany stínu. Proto bývá výhodnější použít statickou množinu dělících intervalů. Výběr kaskádových intervalů je ovlivněn i geometrií scény – u her, kde se kamera nachází velmi blízko povrchu, např. fotbalová hra, je zapotřebí jiná sada intervalů než u leteckých simulátorů. Vlastní dělení lze realizovat vypočtením stejně velkých částí, logaritmickým dělením z-osy a kombinací obou.

Projekční matice pro každou kaskádu těsně přiléhá příslušné podoblasti. V situaci, kdy osa světla je ortogonální k z-ose pohledového jehlanu, nevzniká téměř žádný překryv kaskád. Jakmile je osa světla rovnoběžná, nastává největší překryv. Kromě toho záleží i na mapování jednotlivých kaskád.

Po zvolení intervalů jsou kaskády namapovány jedním ze dvou způsobů:

- **Uzpůsobení scéně** – všechny kaskády mají stejnou blízkou ořezávací rovinu, zároveň mají obrovský vzájemný přesah, protože každá větší zároveň obsahuje celou předchozí.
- **Uzpůsobení kaskádě** – kaskády jsou vytvořeny tak, aby blízká ořezávací rovina další kaskády byla stejná jako vzdálená ořezávací rovina předchozího úseku.

Vytvoření stínových map probíhá několikanásobným vykreslením scény podle projekční matice každé kaskády. Při následném výpočtu stínů je potřeba určit, ke které kaskádě daný bod patří. Určení lze provést dvěma způsoby:

- **Intervalově založený výběr** – výběr se provádí na základě hloubky vykreslovaného bodu a hranic dělících intervalů.
- **Mapově založený výběr** – zohledňuje kromě hloubky i možný překryv map, kdy hloubka může být již větší než hranice, ale natočení mapy způsobí, že bod je vykreslen do mapy s větším rozlišením.

2.6.3 Určení projekční matice světla

Ať už se používá základní mapovaní nebo kaskádové mapy, vždy je potřeba určit projekční matici tak, aby neořezávala objekty vrhající stín. Jednou z možností je vytvořit matici pro celou scénu, což je ale nepoužitelné u větších scén.

Optimální projekční matici lze určit na základě AABB pohledového jehlanu. Aby bylo AABB co nejtěsnější, bývá vhodné vytvářet ho až z bodů transformovaných do prostoru světla. Minimální, resp. maximální hodnota v z-ose označuje blízkou, resp. vzdálenou ořezovou plochu.

Zmíněný postup neposkytuje ideální výsledky zejména kvůli chybějícím stínům u objektů, které se sice nacházejí mimo pohledový jehlan, ale jejich vržený stín by v něm měl být. Toto je způsobeno ořezovými rovinami, jež příliš těsně přiléhají k pohledovému jehlanu. Řešením je nevytvářet roviny přímo ze z souřadnic, ale udělat průnik scény s prostorem světla. Ořezové roviny pak mohou být vypočítány např. algoritmem pro ořezávání trojúhelníků.

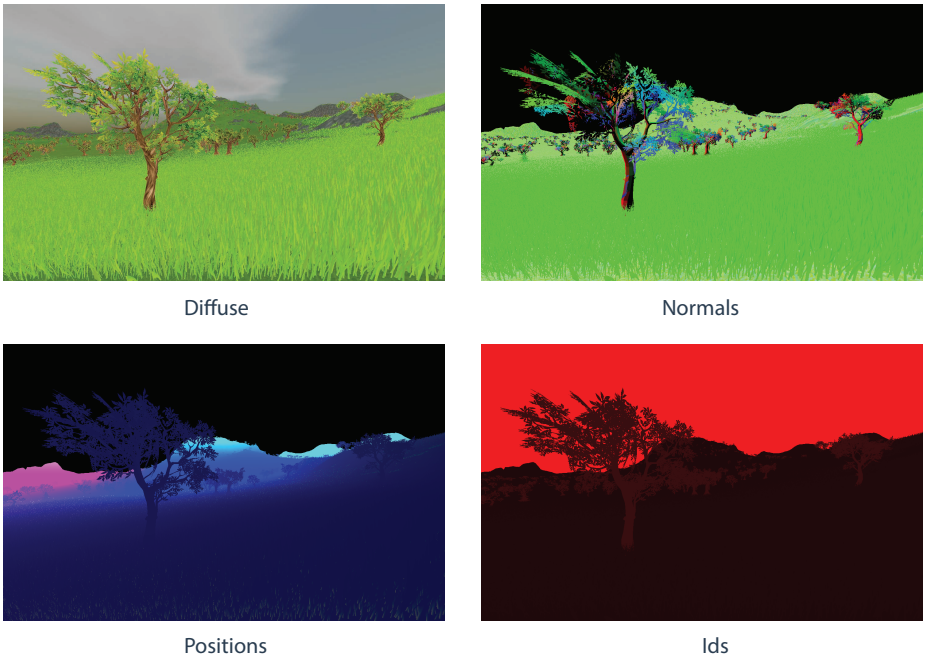
2.7 Frustrum culling

Frustrum culling je technika kladoucí si za cíl snížit výpočetní náročnost vykreslování scény. Po transformaci vykreslovaných objektů se často stává, že některé objekty jsou kompletně mimo pohledové těleso. Aby se zamezilo zbytečným výpočtům s vrcholy, bývá užitečné provést hrubý výpočet s využitím obalových těles pro určení viditelnosti daného objektu. Využívat frustrum culling je zvlášť užitečné v případě reprezentace vykreslovaných objektů pomocí BVH stromu.

Výpočet viditelnosti se provádí otestováním obalové geometrie proti rovinám tvořícím pohledové těleso. V případě AABB to je všech šest vrcholů. Pokud test určí alespoň jednomu bodu příslušnost v pohledovém tělese, obalovaný objekt musí být vykreslen. Extrakci rovin lze provést z projekční matice, jak uvádí článek Fast Extraction of Viewing Frustum Planes from the WorldView-Projection Matrix [8]. První tři sloupce matice určují jednotlivé roviny, a to postupně vždy dvojici levá – pravá, spodní – vrchní a blízká – vzdálená. Pro první, resp. druhý element dvojice je nutno sloupce přičíst, resp. odečíst od posledního sloupce matice. Matice udávající příslušný prostor zároveň určuje, v jakém prostoru musí být výpočet prováděn.

2.8 Výpočet osvětlení pomocí deferred shadingu

Přestože se fragment shader spouští pouze pro fragmenty nacházející se v pohledovém tělese, tak i zde mohou nastávat zbytečné výpočty, jestliže je daný fragment přepsán jiným, jenž je blíže pozorovateli. Aby nebylo nutno vykonávat tyto výpočty, lze použít dvouprůchodovou techniku s názvem deferred shading neboli odložené stínování.



Obrázek 2.15: Uživatelský FBO obsahující textury pro zápis difuzních barev, pozic, normál a identifikátorů objektů

Deffered shading pro svoji činnost potřebuje uživatelsky definovaný framebuffer object (FBO). Uživatelské FBO zapisuje hodnoty do klasických OpenGL textur, které byly v průběhu vytváření FBO uživatelem vytvořeny a připojeny na příslušné výstupy. Pro vícevrstvý FBO je nutné použití texturových polí. Počet a typ textur závisí na počtu a druhu informací nutných k výpočtu finálního osvětlení, případně dalších postprocesových úprav. Typický FBO lze vidět na ilustraci 2.15 a obsahuje textury pro difusní barvy, normály, pozice fragmentů. Požadované hodnoty jsou zapsány do textur ve fragment shaderu při prvním průchodu scénou. Kvalifikátor `layout = n` u výstupních proměnných koresponduje s navazovacím bodem ve FBO a datový typ s datovým typem textury. Jednotlivé textury FBO neomezují způsob práce, z hlediska OpenGL aplikace není žádný rozdíl oproti běžným popisující například materiál objektu. Finální výpočet je proveden ve druhém průchodu, který má k dispozici FBO textury. Vykresluje se čtyřúhelník zabírající celou obrazovku. Pro každý jednotlivý bod jsou z příslušných textur vybrané požadované informace a vypočtena konečná barva bodu.

Konečná barva se modeluje pomocí lokálního osvětlovacího modelu. Jeho vytvoření probíhá použitím rovnice 2.16, což je upravená rovnice z článku The rendering equation [9]. Tato rovnice udává celkovou radianci L, jež se odrazí od povrchu v bodě x ve směru $\vec{\omega}$. Radiance je spočtena integrací všech radiancí dopadajících na povrch v bodě x vynásobenou dvousměrovou odrazovou distribuční funkcí f. Z lokálního osvětlovacího modelu byly odvozeny empirické osvětlovací modely například: Phongův, Blinn-Phongův nebo Lambertův osvětlovací model.

$$L_r\left(x, \vec{\omega}_r\right) = \int_{\Omega} f\left(x, \vec{\omega}_r, \vec{\omega}_i\right) L_i\left(x, \vec{\omega}_i\right) \cos\theta d\omega_i \tag{2.16}$$

2.9 Screen space ambient occlusion (SSAO)

SSAO je postprocesová technika snažící se o vytvoření aproximace ambientního zastínění ve scéně. Technika byly vyvinuta firmou Crytek pro hru Crysis a představena ve článku Finding Next Gen – CryEngine 2 [11].

SSAO vytváří šedotónovou mapu určující poměr zastínění okolními body. Výsledná mapa se nejčastěji aplikuje na ambientní složku osvětlení, ale lze je použít i jiným způsobem. Jak již název napovídá, metoda pracuje pouze v prostoru obrazu, místo s kompletní 3D scénou. K výpočtu se používají hloubky bodů získané při vykreslování scény. Princip spočívá ve vyslání několika náhodných vektorů v kouli okolo pozice aktuálního pixelu. Tyto vzorky se promítnou do roviny obrazu a z rozdílu hloubky oproti původnímu bodu je určen faktor zastínění. Z důvodu vzorkování v kouli bude pro rovnou plochu polovina vzorků zastíněných, a tudíž barva bude šedivá, viz obrázek 2.16. Optimalizací je vysílat vzorky jen v polokouli orientované dle normály bodu. Počet vyslaných vzorků zvyšuje přesnost výpočtů.

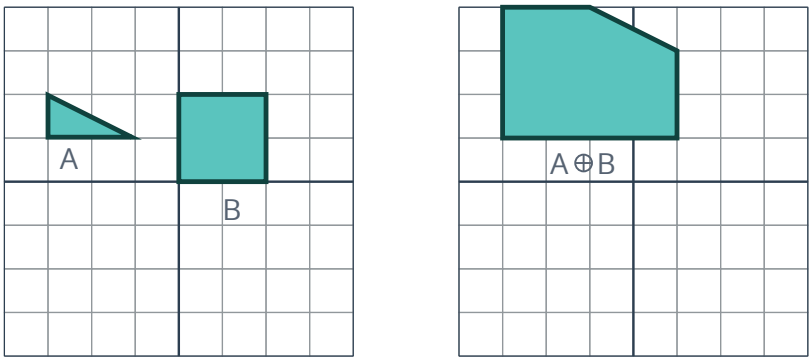
Vytvořená textura má bohužel příliš ostré hrany a přechody. Tento artefakt lze odstranit jejím rozmazáním. Další problém je s malou velikostí kernelu a z toho vyplývající malé náhodnosti vysílaných vektorů. Pro zvětšení entropie se vytváří zvláštní malá textura sloužící pro rotaci kernelu.

Obrázek 2.16: Ukázka SSAO z implementace v Crysis od firmy Crytek.

2.10 Kolize

Scéna se skládá ze statických a dynamických objektů. Statické objekty jsou pevně dané designerem úrovně, a jestliže jsou v kolidujícím stavu, bývá to většinou záměr. Dynamické objekty při pohybu scénou mohou kolidovat, což umožňuje interakce s dalšími objekty. Řešení kolizí se skládá ze dvou kroků: zjištění, zdali došlo ke kolizi a řešení vzniknuvší kolize. Vyhodnocování kolizí je složitá problematika, a tedy vznikají knihovny zabývající se jejich řešením, např. PhysX od Nvidie nebo GJK knihovna.

Pro přesné zjištění kolizního stavu by bylo potřeba otestovat každý trojúhelník, jestli nekoliduje s jiným. Klasická scéna obsahuje velké množství objektů majících tisíce trojúhelníků. Testování každého z nich by zabíralo neúměrnou spoustu času a znemožnilo by



Obrázek 2.17: Ukázka Minkowského součtu dvou objektů.

vykonávání programu v reálném čase. Z toho důvodu se využívají agregační struktury, např. obalová tělesa a vhodné dělení scény. Pro vlastní zjištění kolize je zapotřebí určit průsečík objektů. V rámci článku The Gilbert-Johnson-Keerthi algorithm [6] byla představena detekce kolizí pomocí Minkowského rozdílu.

Minkowského rozdíl vychází z Minkowského sumy, která je definovaná pro množinu bodů A a B pomocí rovnice 2.17 a je ilustrována obrázkem 2.17. Rozdíl popisuje rovnice 2.18. Geometricky lze rozdíl získat přičtením A k B zrcadlenému kolem počátku a tedy $A \ominus B = A \oplus (-B)$. Objekty jsou v kolizi, jestliže nový objekt vzniknuvší spočtením Minkowského rozdílu prochází počátkem. Pro výpočet Minkowského rozdílu u pohybujícího se objektu se používá množina, která vznikne sjednocením bodů na trase z počátečního do koncového bodu pohybového vektoru.

$$A \oplus B = \{a + b : a \in A, b \in B\} \tag{2.17}$$

$$A \ominus B = \{a - b : a \in A, b \in B\} \tag{2.18}$$

Reakce na kolizi může být následující:

- **Odražení** – objekt je odražen ve směru, jenž je zrcadlení pohybového vektoru kolem normály kolizního povrchu. Velikost nového vektoru je vypočtena z pohybového vektoru odečtením velikosti vektoru před kolizí.
- **Posunutí** – kolizní objekt je posunut po zbývající dráze pohybujícího se objektu.
- **Sklouznutí** – objekt po nárazu pokračuje po povrchu kolizního objektu. Délka pohybu je definována jako v prvním bodě.

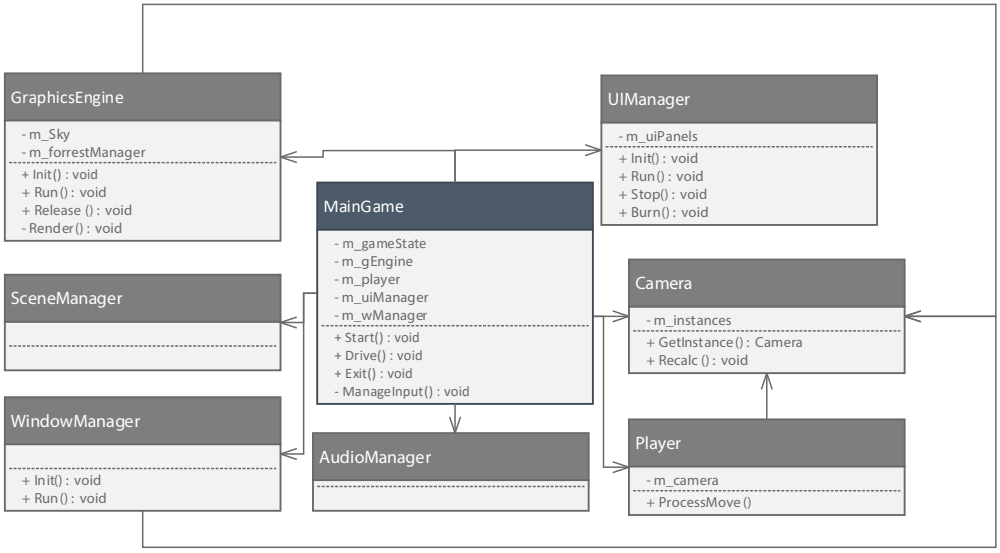
Kapitola 3

Návrh aplikace

V rámci této práce byla vytvořena demonstrativní aplikace využívající většinu technik popsaných v předchozí kapitole. Tato kapitola prezentuje jejich aplikaci a uzpůsobení konkrétní situaci.

3.1 Architektura aplikace

Základní bloky aplikace ilustruje UML diagram 3.1. Celá aplikace běží v nekonečné smyčce v hlavním objektu MainGame. Tento objekt je vstupním bodem aplikace, ověřuje existenci konfiguračních souborů a inicializuje další potřebné bloky. Zároveň se jedná o rozhraní předávající komunikační zprávy mezi různými částmi aplikace. Objekt může v aplikaci existovat pouze jednou, tudíž je zde použit návrhový vzor singleton. MainGame zajišťuje přepínání herního stavu aplikace za jejího běhu. Herní stav ovlivňuje právě volané manažery a může sloužit i k jejich opožděné inicializaci.



Obrázek 3.1: Zjednodušený UML diagram tříd jádra aplikace. Na obrázku jsou zobrazeny pouze relevantní metody a data.

Aplikace se dále skládá z několika na sobě nezávislých manažerů. Tito manažeři mají na starost vlastní logicky ucelené části dané problematiky. Pro svoji činnost využívají další již specifičtější třídy. Výjimku tvoří kamera, která představuje speciálního manažera, jenž se dále nedělí a je využíván mnohými dalšími.

3.1.1 Kamera

Kamera tvoří v aplikaci zvláštní objekt, který je nezávislý na ostatních. Každá kamera je identifikovaná unikátním jménem v rámci kategorie. Vytváření probíhá přes získávání instance, kdy, pokud kamera zadaného jména neexistuje, vytvoří se nová, v opačném případě se vrátí již dříve vytvořená instance.

Metody, jež objekt podporuje, slouží k vytváření pohledových a projekčních matic. Matice jsou vytvářeny z pozice a rotace kamery. Tyto hodnoty jsou aktualizovány přes speciální funkci. Kamera využívající ortogonální projekci umí vytvořit matice z rohů pohledového tělesa. Zároveň pomocí metody ořezávání trojúhelníku vypočte blízkou a vzdálenou ořezovou rovinu.

3.1.2 Graphics engine

GE tvoří grafické jádro aplikace. Jeho úkolem je vytvořit a spravovat specializované manažery zaměřené na konkrétnější části grafické stránky aplikace. Do této sekce nespadá UI. GE ukládá manažery do svých vnitřních struktur, provádí jejich inicializaci, a jelikož jeho životnost přesahuje dobu běhu jedné úrovně, obsahuje i funkce pro uvolnění OpenGL paměti. Kromě správy manažerů, GE spravuje a vykresluje `GBuffer` a aktualizuje hodnoty v UBO.

Vykreslování není zvlášť volané z `MainGame`, ale je obsaženo v rámci `Run();` funkce, která provádí vyčtení aktuálních hodnot matic kamer, úpravu UBO a zavolání privátní vykreslovací funkce. Vykreslovací funkce provádí změnu RBufferu. Dále provede všechny průchody scénou, přičemž volá vykreslovací funkce manažerů, provede post procesové efekty s vykreslenou scénou a zajistí vykreslení scény do defaultního FBO.

3.1.3 RBuffer

`RBuffer`, také někdy označován jako `GBuffer`, ale toto slovo má v mé aplikaci poněkud odlišný význam, je objekt sloužící ke zjednodušení práce s OpenGL FBO. Poskytuje objektové rozhraní pro jednoduchou změnu FBO a připojení FBO pro čtení.

3.1.4 Player

Objekt `Player` definuje ve scéně hráče. Každý hráč může, nebo nemusí mít přiřazenou kameru. Pokud ji má, ovlivňuje svojí pozicí její pozici, která pak odpovídá definovanému bodu na objektu hráče. Prostorové vlastnosti hráče jsou navenek určeny funkcí, jež dokáže s využitím Minkowského součtu rozhodnout, zdali dochází, nebo nedochází ke kolizi. Aby funkce byla schopna kolize vyhodnocovat, využívá obalových těles pro reprezentaci hráče.

Aktualizace pozice hráče provádí `MainGame` objekt pomocí pohybového vektoru. Při obdržení nových dat provede `Player` vyhodnocení kolizí s okolními objekty a terénem, na jejichž základě se provede úprava pohybového vektoru. Při kolizi s objektem dochází pouze k přičtení obrácené hodnoty zbývající části pohybového vektoru. U kolize s terénem se kontroluje několik věcí. První kontrola zjišťuje, zdali objekt stojí na povrchu, pokud ne,

upraví se pohybový vektor o gravitační zrychlení působící směrem dolů. Druhá kontroluje, jestli výška cílového místa není větší než maximální povolená stoupavost. Poslední využívá gradient získaný ze čtyř rohových bodů dlaždice, na které se nachází `Player`, k výpočtu gravitačního vektoru. Tento vektor umožňuje sklouznutí z prudkých svahů a řeší chybu druhé kontroly, jež selhává při pohybu, jehož směr se blíží směru vrstevnice v daném místě.

3.1.5 Správce zvuku

Správce zvuku je objekt aplikace mající na starost abstrakci zvukového rozhraní. Tento objekt opět implementuje návrhový vzor singleton. Objekt řídí přehrávání zvuků dle vnitřní situace scény. Situace je ovlivňování skrz jeho rozhraní. Správce obsahuje podporu pro 2D i 3D zvuky. Konkrétní zvukové soubory jsou získávány z konfiguračního souboru.

3.1.6 UI

Uživatelské rozhraní má v aplikaci na starost UIManager. UI je vytvářeno pomocí pohledů, kdy každý pohled musí implementovat následující rozhraní:

- `void Init();` – metoda volaná při vytváření pohledu, sloužící k nastavení potřebných hodnot a vytvoření struktur pro správnou práci pohledu.
- `void Wake();` – funkce volaná při změně aktuálně vykreslovaného pohledu na tento pohled. Slouží především k vyresetováni stavu pohledu.
- `void Run();` – metoda, jež se volá při každém snímku pro aktuálně vykreslovaný pohled.
- `void RunImgui();` – funkce, jež se volá hned po metodě `Run();` a je pro ni typické, že běží v kontextu `ImGUI` snímku.
- `void Stop();` – metoda, jež určuje, že dochází ke změně aktuálně vykreslovaného pohledu a tento pohled již nebude dále vykreslován. Metoda se volá vždy na začátku nového snímku, těsně před metodou `Wake()`;
- `void Burn();` – inverzní metoda k `Init();` sloužící k uvolnění alokovaných zdrojů. Po této metodě může, nebo nemusí dojít k destrukci samotného objektu. Pokud byla tato metoda zavolána na nějaký objekt, musí dojít před jeho znovupoužitím k opětovné inicializaci.

Kromě hrubého dělení na pohledy, obsahují jednotlivé pohledy jemnější dělení na panely. Správa a měnění vykreslovaných panelů je interní záležitostí pohledů. Ke své činnosti pohledy dále využívají objekt `ConfigStore` ke zjišťování konfiguračních vlastností a objekt `Translator`.

Translator slouží k získání přeloženého řetězce z jazykových souborů. Každý řetězec je identifikovaný unikátním klíčem složeným z klíče a kategorie. Pro získání přeloženého řetězce se volá metoda `Translate(key, category);`. `Translator` využívá návrhový vzor singleton. Jazykový soubor je načten dle aktuální hodnoty zvoleného jazyku v konfiguračním souboru nastavené v době vytváření první instance. Změna jazykového souboru se provádí pomocí funkce `Reload();` jež provede znovunačtení jazykových souborů, kdy opět využívá hodnotu z konfiguračního souboru.

Pohledy se v aplikaci vytváří při inicializaci `UIManageru`, avšak není nutno provádět jejich okamžitou inicializaci. Jednotlivé pohledy jsou uloženy ve vnitřní struktuře `UIManageru` umožňující dynamické přidávání pohledů za běhu aplikace. K pohledům `UIManager` přistupuje přes rozhraní definované výše.

3.2 Správci vykreslování

Správci vykreslování tvoří ucelené objekty pracující přímo s 3D modely. Jejich druhy se odvíjí od druhů vykreslování v OpenGL a tedy existují dva správci. Správci pracují s modely mimo jiné přes datový ukazatel – `DataPtr`.

3.2.1 Datový ukazatel (DataPtr)

`DataPtr` představuje objekt sloužící ke zjednodušení práce s modely. Každý model tvoří jeden až n meshů, které mohou být uložené v různých strukturách pro vykreslování. `DataPtr` abstrahuje tyto struktury, a tak při jeho použití se provede úprava všech relevantních dat.

`DataPtr` neobsahuje žádnou funkcionalitu, jeho interpretace je zcela závislá na objektu, kterému je předán. Přes něj se provádí transformace modelů a jejich mazání. Všechny tyto úpravy ovlivňují pouze grafickou část aplikace.

3.2.2 GBuffer

`GBuffer` je správce vykreslovaných objektů starající se o jejich vykreslování pomocí nepřímého kreslení. Jeho návrh je nezávislý na počtu a druhů vykreslovaných objektů.

Základním požadavkem kladeným na `DataPtr` je schopnost jednoznačně identifikovat jeden konkrétní model. Identifikace se provádí s využitím správce vykreslování daného objektu. V rámci správce musí být `DataPtr` unikátní. O vytvoření `DataPtr` se starají konkrétní správci a grafu scény jej předávají při načítaní modelů.

`GBuffer` se nevytváří konkrétně pro jeden objekt, ale jeho správu má na starosti `GraphicsEngine`. Každý načtený objekt je předán `GBufferu`, jenž si jej zpracuje a uloží do vnitřních struktur. Ukládat se musí vrcholy, indicie a instanční transformační matice. Dále spravuje i požadované textury objektů a zajišťuje, aby byly načítány pouze unikátní textury. Určení unikátnosti textury je provedeno na základě objektu, k němuž patří, typu textury a jejího názvu. Ve vnitřní struktuře jsou jednotlivé objekty definované svým řetězcovým jménem. Objekty se přidávají do `GBufferu` buď úplným záznamem nebo pomocí instančních matic.

Úplný záznam (kód 3.2) slouží k přidávání meshů. Objekt, k němuž patří, je definován textovým jménem. Pokud objekt daného jména neexistuje, vytvoří se nový, pokud ano, tak se k němu pouze přidá další mesh. Takto přidané objekty nemají vlastní instance, tudíž k nim ani neexistují `DataPtr` objekty.

Instance se přidávají pomoci seznamu instančních matic. Model, k němuž patří, je opět definovaný svým jménem a je nutné, aby již existoval. Při jeho neexistenci dojde k výjimce. Při přidávání instancí jsou již vraceny `DataPtr` pro jednotlivé objekty, jedna matice = jeden `DataPtr`. Ve své vnitřní struktuře je `DataPtr` schopný identifikovat jednotlivé meshe.

Pro další práci s `GBufferem` je nutné jej zapéct. Zapečení vytvoří požadované struktury na grafické kartě a vytvoří vykreslovací příkazy. Po zapečení již není možné přidávat nové objekty, lze pouze měnit počet instancí, transformační matice, dodatečná data a mazat.

Mazání pouze odstraní příslušná instanční data, ostatní informace pořád zůstávají na GPU. Vykreslení probíhá voláním příslušné funkce a je zcela v režii `GBufferu`.

```
1     struct GBufferStruct {
2         vector  vertices;
3         vector indices;
4         mat4 transformation;
5         vector textures;
6         GLfloat alphaDiscard;
7         GLuint firstIndex =0;
8         GLint baseVertex =0;
9     };
```

Program 3.2: Vnitřní struktura GBufferu ukládající informace o jednom meshi.

3.2.3 Modelová třída

Modelová třída představuje druhého správce vykreslovaného objektu a zároveň je to třída představující samotný objekt. Vykreslování probíhá využitím přímého kreslení. Modelová třída má v aplikaci dva úkoly: načíst zadaný model a obstarávat jeho vykreslování. První část činnosti se provádí i při použití nepřímého vykreslování.

U načítání modelu správce zpracovává zadaný soubor. Zpracování spočívá v předzpracování načítaného modelu a výběru relevantních dat. Tato data jsou uložena do vnitřní struktury. V průběhu zpracovávání dochází k opožděnému načítání textur umožňující zvýšení datové efektivity. V případě nepřímého vykreslování dojde pouze k předání dat do `GBufferu`. Při přímém vykreslování jsou vytvořeny potřebné OpenGL struktury, které jsou vytvářeny pro každý mesh zvlášť. Přímé vykreslování je podporováno pouze instanciační formou, tudíž musí být opět dodány instančí matice a po jejich dodání jsou navráceny vytvořené `DataPtr` objekty. Transformace a mazání je stejné jako u `GBufferu`. Kromě toho je zde možnost smazáním modelové třídy smazat vykreslovaný objekt úplně a uvolnit tak OpenGL zdroje. Smazání této třídy při použití s `GBufferem` nevyvolá neočekávané chování a je tedy z pohledu `GBufferu` zcela bezpečné. Bezpečné nemusí být z pohledu grafu scény, jež obsahuje ukazatel na model. Tento ukazatel však není v současnosti při nepřímém vykreslování používán.

Samotná třída také musí vědět, jestli je použita pro přímé, nebo nepřímé vykreslování. Tento příznak se nastavuje ihned při vytváření třídy. Pokud se uživatel pokusí použít třídu opačně, dojde k vyvolání výjimky. Vykreslení modelu je opět v režii třídy a provádí se voláním příslušné metody.

3.3 Graf scény

Pro různé výpočty s objekty v herním světě nebo jejich úpravu je nutno udržovat určité povědomí o celé scéně. Toto povědomí je nejčastěji specifikováno za pomoci grafu scény s různou granularitou podřízenou konkrétním požadavkům. Já jsem se rozhodl zvolit reprezentaci objektů ve scéně pomocí BVH stromu. Tento strom je dělen pomocí AABB představující jednotlivé záznamy objektů. Jelikož vytvářená aplikace pracuje ve 3D světě,

byl nejprve zvolen oktávový strom jako primitivní struktura pro BVH. V průběhu experimentování se scénou se ukázalo, že přestože se jedná o 3D svět, tak data mají malý rozptyl v y-ose, jelikož fakticky jsou data umísťovaná pouze ve 2D. Tento fakt vedl k degradaci časové složitosti vyhledávání na kvadrantový strom, a zároveň zde vznikalo spoustu prázdných pomocných uzlů zabírající zbytečnou paměť. Zjištěná skutečnost mě vedla ke konečnému rozhodnutí použít kvadrantový strom.

Klasický BVH definuje pro každý uzel pouze pevně daný maximální počet objektů. Při praktickém řešení je potřeba uvažovat situace překrývajících se objektů a objekty, jež jsou větší než daný uzel. Mnou navržený strom neukládá data pouze v listových uzlech, ale datový slot se nachází v každém uzlu stromu. Uzly ve stromu jsou využity pro data, jejichž rozměry jsou větší než rozměry uzlů. Počet těchto dat není omezen. Listové uzly mají pevně definovaný maximální počet dat. Překrývající se objekty jsou počítány za jeden objekt. Dva objekty se překrývají, jestliže jejich AABB mají průsečík. Kvadrantový strom pro 3D je neoptimální při vybírání objektu pomocí vrhání paprsku. Tento problém jsem řešil nastavením nulové vzdálenosti v y-ose pro vytvářený uzel a její přepočítávání pro každý vložený objekt. Protože v určitých částech scény, např. interiér domu, jsou objekty, které mají velký rozptyl v y-ose je potřeba zachovat obě varianty a oktávový strom používat jako podrozdělení kvadrantového. Jelikož je scéna definovaná pomocí absolutních pozic, musí se při každém pohybu objektu zkontrolovat, zdali ještě spadá do daného uzlu. Pokud ne, objekt se vyjme a vloží na nové místo.

Záznam objektu byl navrhnut tak, aby umožnil efektivní práci a poskytoval všechny důležité údaje. A tedy obsahuje:

- AABB těleso
- Jméno modelu a cestu k němu
- Ukazatel na Modelovou třídu (vysvětleno v 3.2.3)
- Příznak, jestli je kreslen nepřímo, a ukazatel na GBuffer (vysvětleno v 3.2.2)
- Datový ukazatel (vysvětleno v 3.2.1)
- Ukazatel na uzel v grafu scény, který jej obsahuje

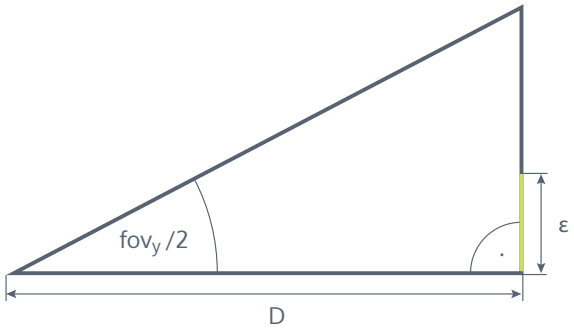
3.4 Generování trávy

Algoritmus, jenž jsem navrhnul pro vykreslování trávy, využívá procedurální generování. Veškerá tráva je vykreslena bez přičinění CPU. Algoritmus byl inspirován koncepty vykreslování trávy pomocí billboardingu, avšak mnou navržená tráva se vykresluje po jednotlivých listech. Vykreslování trávy probíhá ve dvou průchodech. První vytvoří body (seeds), ve kterých se bude tráva generovat, druhý vygeneruje vlastní trávu.

Vytvoření seedů probíhá při vykreslování terénu. Terén obsahuje materiálovou texturu určující zatravněná místa, ve kterých se má generovat tráva. Využití průchodů vykreslování terénu umožňuje hardwarový frustrum culling, jelikož se využijí pouze fragmenty ve viditelné části scény. Nevýhodou je chybějící tráva, která se sice nachází mimo, ale její část by zasahovala do pohledového tělesa. Tento artefakt se zvlášť projevuje v části blízké přední ořezové rovině. Na bocích je viditelný až při větší velikosti generované trávy.

Aby tráva mezi jednotlivými snímky neměnila pozice, což působí velice rušivě, zarovnávají se seedy do pravidelné mřížky. Zarovnání je provedeno zaokrouhlením světových

souřadnic na požadované číslo. Zaokrouhlovány musí být všechny tři souřadnice. Zaokrouhlením vznikne vzájemný překryv fragmentů, kdy více fragmentů vyprodukuje stejný seed. Proto je potřeba odfiltrovat fragmenty, jež nejsou pravoplatnými generátory. K vybírání se používají derivační funkce a konkrétně se zjišťuje velikost fragmentu. Pokud je rozdíl mezi původními a zaokrouhlenými souřadnicemi větší než velikost fragmentu, implikuje to, že musí existovat fragment, který se bude také zaokrouhlovat na stejné hodnoty, a tudíž právě zpracovaný fragment nebude použit jako seed. Experimentálně bylo ověřeno, že s touto kontrolou se sníží počet vygenerovaných seedů na 18 %. Dále lze od určité vzdálenosti ignorovat fragmenty, protože výsledná vykreslená tráva by byla velice malá. Vzdálenost se vypočítá pomocí goniometrických funkcí a to tak, že se hledá vzdálenost, pro niž bude mít tráva známé velikosti zadanou velikost ε viz ilustrace 3.3. Tato hodnota je stejná pro všechny fragmenty a označuje hraniční hloubku pohledového tělesa.

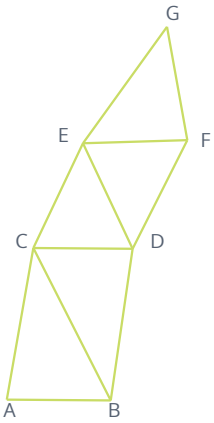


Obrázek 3.3: Výpočet hraniční vzdálenosti, po kterou mají být generovány jednotlivé seedy. Hledá se takové D, aby tráva měla velikost ε.

Pro ukládání hodnot seedů se využívají SSBO. Aby nebylo nutné provádět packing algoritmus nad SSBO, rozhodnul jsem se pro využití atomických čítačů. Po zjištění, zdali daný fragment je generátorem, provede se přičtení požadované konstanty k atomickému čítači. Velikost konstanty závisí na tom, kolik stébel se bude generovat. Prakticky jsem přičítal jedničku. Atomické čítače zároveň vrací jejich hodnotu před operací. Tato hodnota je adresa prvního volného místa v SSBO. Může se stát, že vrácená hodnota společně s počtem seedů je mimo rozsah adres SSBO, proto je ji potřeba kontrolovat.

K odstranění synchronizace mezi CPU a GPU jsem zvolil nepřímé vykreslování a počet seedů určuje počet instancí. Tímto ovšem zmizí možnost kontroly rozsahu adres SSBO, je proto nutno zařadit ještě compute shader, jehož úloha je pouze ořezat převyšující počet instancí. Dále přibyla nutnost použití GPU synchronizačních bariér. Vytvoření jednotlivých stébel má na starosti geometry shader, vytvářející ze vstupního bodu trojúhelníkový pás o pěti trojúhelnících, a tedy generuje sedm vrcholů. Výsledné stéblo lze vidět na obrázku 3.4. Prvotní bod A odpovídá danému seedu. V geometry shaderu lze použitím pseudonáhodného generátoru náhodných čísel rozbít pravidelnost trávy posunutím bodu A a rotací bodu B. Výška jedné části se provede úpravou pevně zadané hodnoty pomocí vygenerovaného náhodného čísla. Upravená výška je pak pro všechny segmenty stejná. Šířka je pevně zadaná, upravuje se pouze pozice bodů. Důležité je, aby dvojice sousedních bodů, např. C a D, byla posunuta ve stejném směru. Dále je vizuálně vhodné posouvat všechny vrcholy stejným směrem. Barva se vybírá pro každý vrchol náhodně z pole barev.

Aby byla zachována statičnost stébel, používají se jako vstupní hodnoty generátoru souřadnice seedů, což ovšem dává pouze jedno náhodné číslo pro celé stéblo. Zvýšení počtu



Obrázek 3.4: Jedno stéblo trávy generované v geometry shaderu pomocí trojúhelníkového pásu.

náhodných čísel lze dosáhnout používáním jednotlivých číslic desetinné části. Nevýhodou je pouze deset variant takového čísla, avšak pokud se pro rotaci bodu B použije celé číslo, není tento problém patrný.

3.5 Editor

V rámci vývoje byl vytvořen i jednoduchý jednoúčelový editor, jehož snímek lze vidět na ilustraci 3.5. Editor je tvořen UI pohledem a podpůrnou třídou, jež vychází z návrhového vzoru singleton. Tento editor umožňuje jednoduché přidávání, mazání a úpravu objektů. Veškeré úpravy ovlivňují celou aplikaci.

Obrázek 3.5: Snímek rozhraní editoru implementovaného v aplikaci.

Výběr objektu probíhá využitím vrhání paprsku nad grafem scény. Průsečík se testuje pouze s AABB objektů. Jelikož objekty v grafu scény mají jiné uspořádaní oproti zarovnání dle hloubky v pohledovém prostoru, kontroluje se kromě průsečíku i hloubka nalezeného objektu. Vybrání objektu probíhá zmáčknutím levého tlačítka myši nad objektem ve vykreslené scéně. Pokud je pod kursorem myši nalezen objekt, je zvýrazněn fialovějším zabarvením. Vybraný objekt lze následně posunout, provést jeho rotaci, změnit mu měřítko nebo ho smazat.

Pozici vkládaného objektu lze zvolit dvojím způsobem. První vloží nový objekt na aktuální pozici kamery. Při druhém způsobu se provádí hledání průsečíku terénu a paprsku vyslaného z pozice myši. Testování probíhá po pevně daných krocích, jejichž vzdálenost je stejná jako nejmenší vzdálenost vrcholů jedné terénní dlaždice. Jakmile je objeven interval, ve kterém dojde k průsečíku s terénem, provede se metodou půlení intervalu dokročení na přesnost definovanou nastaveným epsilon. Data vkládaného objektu se získávají ze šablon, které jsou vytvořeny pro každý objekt při načítání scény. Jméno šablona získává z řetězce uvedeného v rámci jména objektu v konfiguračním souboru.

Kamera je v editoru nezávislá na pohybu myši. Změna natočení kamery se provádí držením pravého tlačítka myši, čímž kamera reaguje na její pohyb. Zdůraznění tohoto stavu indikuje skrytý ukazatel myši.

Všechny změny objektů mají pouze temporální charakter pro aktuální instanci editoru. Instancí se zde rozumí běh UI pohledu editoru. K uložení změn musí uživatel zmáčknout tlačítko Uložit, po jehož zmáčknutí dojde k uložení transformačních matic pro všechny instance modelů aktuálně přítomných v grafu scény. Bližší popis UI editoru je uveden v příloze A.

Kapitola 4

Implentační detaily

Tuto kapitolu bych chtěl věnovat implementačnímu popisu částí zasluhujících podrobnější rozbor. Kromě věcí zmíněných v kapitole Návrh 3 byly v této práci implementovány následující body:

- Teselovaný terén
- Fyzikálně simulovaná obloha se šumově generovanými mraky
- CSM s PCF
- Blin-Phong osvětlovací model
- SSAO
- Mlha
- Vinětace
- Kolize využívající Minkowského součet

Dále osvětlovací shader obsahuje i metodu pro počítání rozptylu světla známou pod názvem God rays nebo také Crepuscular rays.

4.1 Použité technologie

Aplikace byla vyvíjena v jazyce C++. Pro vykreslování 3D grafiky bylo využito OpenGL ve verzi 4.4, ke kterému přistupuji pomocí multiplatformní knihovny GLFW. Načítání modelu je realizováno pomocí knihovny Assimp, obrázky využívají knihovnu FreeImage. Uživatelské rozhraní bylo vytvářené přes knihovnu Dear imgui také známou jako ImGui. O zvukovou stránku aplikace se stará knihovna IrrKlang.

Vývoj a ladění probíhal v systému Windows 10 Education nejprve s grafickou kartou AMD HD 6870, později, z důvodu problému a chyb driveru, jsem přešel na nVidia GeForce GTX 1060 6GB. Vývojové prostředí tvořil program Microsoft Visual Studio Enterprise 2015. Projekt by měl být přeložitelný i na jiných platformách, ale musí se mu dodat potřebné .lib a .dll soubory. Kromě toho je potřeba zkonvertovat projekt pro cílený překladač.

4.2 Obloha

Obloha, tak jak byla popsaná v kapitole 2.4, využívá výpočet v reálných rozměrech, kdy se porovnávají tisíce kilometrů s nanometry, což způsobuje velké odchylky z principu reprezentace čísel v počítači. Pokud se ovšem určí, že průměr koule tvořící atmosféru se rovná jedné, dá se provést značné zjednodušení. Především zjednodušení nastane při výpočtu průsečíku vektoru a koule. Dále pak při poměrovém přepočtení potřebných hodnot už nedochází k porovnávání čísel s velkými řádovými odchylkami.

Jelikož průměrný průměr Země je 6378 km, tak lze zanedbat pozici a výšku hráče a implicitně za výšku považovat průměr. Tato úprava ovšem způsobí, že hráč při pohybu vzhůru nemůže vyletět mimo atmosféru, avšak tento pohyb není v mé aplikaci umožněn.

Při HDR mapování na LDR zmizelo z oblohy Slunce. Chybějící Slunce jsem vyřešil umělým mixováním barvy oblohy s bílou barvou. Mixování se provádělo jen v kružnici, jejíž střed definovala pozice Slunce a poloměr byl empiricky zjištěn.

4.3 CSM

Abych nemusel pro každou mapu volat zvláštní vykreslovací funkci, využil jsem vícevrstvý FBO. V tomto případě fragment shader pouze přeposílá hodnoty do geometry shaderu, který se spouští jednou pro každý vrchol, jenž postupně vynásobí patřičnými maticemi a nastaví odpovídající výstupní vrstvu. Matice získává z UBO.

Problém, který nastává s vrstveným FBO je, že současná verze GLSL neposkytuje funkci pro čtení hloubkové mapy, která by současně podporovala vrstvy a zároveň umožnila nastavit bias hodnotu. Toto omezení jsem se pokusil vyřešit přes `glTextureView` a vytvořil pro každou vrstvu jednu proxy texturu. Textury již šlo použít se čtením hloubkové mapy společně s bias hodnotou, avšak GLSL tuto hodnotu ignoroval, což bylo usouzeno na základě, že jakákoliv nastavená hodnota nemá absolutně žádný vliv na výsledek. Zároveň na AMD grafikách vznikaly artefakty na hranách. Tyto problémy mě vedly zpět ke zrušení proxy textur a místo bias hodnoty pro zrušení sebe zastiňování použít `glPolygonOffset`.

4.4 Strom scény

Strom scény jsem vytvářel s myšlenkou co možná nejvíce univerzální třídy. Důvody, které mě k tomu vedly, byly jednak možné změny ukládaných dat a za druhé možnost využít jeho zdrojové kódy i pro další aplikace. Z těchto důvodů jsem strom scény vytvořil přes klíčové slovo `template` jako šablonovou třídu.

Ve stromě je implementován pouze pre-order průchod, ale jeho největší devizou je lambda funkce v rámci parametru. Této lambda funkci jsou předána data aktuálního uzlu. Využití v mé aplikaci nachází při ukládání scény, kdy lambda funkce provádí serializaci obdržených dat. Podobně jsou řešeny i kolize, kdy strom obsahuje průchod po úrovních s výběrem kolizní cesty. Kolizní cesta je vybírána na základě hodnoty vrácené z lambda funkce předávané v rámci parametru.

Kapitola 5

Výkonnostní měření

V předposlední kapitole mé práce bych se chtěl věnovat testování aplikace. Věc, kterou každý hráč u hry řeší, je počet snímků za sekundu. Kromě nich se využívají FPS k měření výkonnosti a porovnávání grafických karet. A jelikož téma práce zní 3D herní svět v OpenGL, tak jsem si z těchto důvodů zvolil počet FPS za hlavní metriku aplikace.

5.1 Statistické hodnoty scény

Statistické hodnoty všech modelů, jež byly vykreslované ve scéně, jsou ukázány v tabulce 5.2. Modely byly rozdělené do tří sad, v tabulce jsou jednotlivé sady odděleny barevně, přičemž každá sada pokrývala objekty sady předchozí. A tedy sada 1 obsahovala modré objekty, což bylo celkově 1 081 274 trojúhelníků. Sada 2 přidala zelené objekty a celkově měla 3 198 000 trojúhelníků. Sada 3 zvětšila množinu o červené objekty a konečné množství dosáhlo 6 765 201 trojúhelníků.

Modely byly vykreslované metodou nepřímého vykreslování. Z důvodu přípravy na frustrum culling tvořila každá instance jeden kreslící příkaz. Frustrum culling se nepovedlo správně odladit, tudíž nebyl použit a karta musela zpracovávat všechny trojúhelníky.

Rozlišení	Počet seedů	Σ trojúhelníků
1280 x 720	240 472	1 202 360
1680 x 1050	434 461	2 172 305
1920 x 1080	487 688	2 438 440

Tabulka 5.1: Průměrný počet vygenerovaných seedů v závislosti na rozlišení obrazovky aplikace.

Počet trojúhelníků terénu ovlivňovala pozice kamery, ale průměrně obsahoval 181 dlaždic s 2895 vrcholy. Jednotlivé listy trávy byly generovány s rozestupem 0,25 jednotek a celkový počet závisel nejvíce na rozlišení vykreslované plochy. Mírně byl ovlivněn i povrchem terénu. Počet seedů shrnuje tabulka 5.1

Jméno	Trojúhelníky	Vrcholy	Instancí	Σ trojúhelníků
Rock 2	356	237	65	23 140
Camp fire	7 202	17 391	3	21 606
Axe	548	373	2	1096
Tree 1	1 576	1 073	657	1 035 432
Tree 2	1 754	3 750	629	1 103 266
Tree Aut. 1	1 596	1 071	635	1 013 460
Tree 3	1 442	938	651	938 742
Tree Aut. 2	2 630	5 251	676	1 777 880
Fir tree	736	593	166	122 176
Dead branch	1 500	1 828	85	97 500
Dead tree	2 625	3 600	172	619 200
Stump	2 999	8 997	1	2 999
Stump 2	8 704	4 518	1	8 704

Tabulka 5.2: Počet vrcholů a trojúhelníků pro použité modely. Barevně jsou odlišené jednotlivé sady, každá následující sada obsahuje i tu předchozí.

5.2 Testovací prostředí

Testování probíhalo na dvou sestavách:

1. Sestava
   - **OS:** Windows 10 Education
   - **CPU:** AMD FX-8300 @ 3,3 GHz
   - **RAM:** 16GB
   - **GPU:** GeForce GTX 1060 6GB
2. Sestava
   - **OS:** Windows 7 Pro
   - **CPU:** Intel Core i7-4790K @ 4.00 GHZ
   - **RAM:** 16GB
   - **GPU:**
     (a) AMD Radeon RX480 8 GB
     (b) AMD Radeon R9 Furry X
     (c) GeForce GTX 780 Ti

Metodika měření se skládala z těchto kroků:

1. Zapnutí aplikace s požadovaným nastavením, načtení scény, počkání na ustálení scény
2. Zapnutí měření FPS
3. Náhodný průchod centrální oblastí scény
4. Vypnutí měření FPS
5. Zapsání naměřených dat, ukončení aplikace

5.3 Výsledky měření

Měření probíhalo na všech kombinacích sad objektů a vybraných třech rozlišeních obrazovky. Nejdříve se měřila sestava 1 a rychlost vykreslování uvádí tabulka 5.3. Další měření probíhalo již na sestavě 2 s kartami ve stejném pořadí, v jakém byly uvedeny v popisu sestavy. Jejich naměřené hodnoty jsou uvedené postupně v tabulce 5.4, v tabulce 5.5 a v tabulce 5.6.

Sada obj.	Rozlišení	Min. FPS	Max. FPS	Avg FPS
1	1280 x 720	67	71	69,094
1	1680 x 1050	41	50	43,909
1	1920 x 1080	36	41	37,703
2	1280 x 1720	41	44	42,833
2	1680 x 1050	28	32	29,303
2	1920 x 1080	27	31	28,213
3	1280 x 720	31	33	32,353
3	1680 x 1050	24	25	24,609
3	1920 x 1080	23	26	24,094

Tabulka 5.3: Rychlost vykreslování s GTX 1060 6GB.

Sada obj.	Rozlišení	Min. FPS	Max. FPS	⌀ FPS
1	1280 x 720	33	36	33,284
1	1680 x 1050	28	29	28,667
1	1920 x 1080	27	29	28,250
2	1280 x 1720	13	17	16,000
2	1680 x 1050	13	14	13,640
2	1920 x 1080	13	14	13,523
3	1280 x 720	10	11	10,500
3	1680 x 1050	9	10	9,500
3	1920 x 1080	9	10	9,150

Tabulka 5.4: Rychlost vykreslování s AMD Radeon RX480.

Sada obj.	Rozlišení	Min. FPS	Max. FPS	⌀ FPS
1	1280 x 720	52	60	56,618
1	1680 x 1050	39	43	41,103
1	1920 x 1080	35	37	36,281
2	1280 x 1720	23	24	23,276
2	1680 x 1050	20	21	20,296
2	1920 x 1080	18	19	18,594
3	1280 x 720	15	16	15,462
3	1680 x 1050	13	15	14,000
3	1920 x 1080	13	14	13,500

Tabulka 5.5: Rychlost vykreslování s AMD Radeon R9 Furry X.

Sada obj.	Rozlišení	Min. FPS	Max. FPS	⌀ FPS
1	1280 x 720	66	70	67,720
1	1680 x 1050	39	45	42,483
1	1920 x 1080	32	38	35,350
2	1280 x 1720	38	39	38,316
2	1680 x 1050	26	30	27,813
2	1920 x 1080	23	36	25,212
3	1280 x 720	27	29	28,177
3	1680 x 1050	22	24	23,358
3	1920 x 1080	21	22	21,450

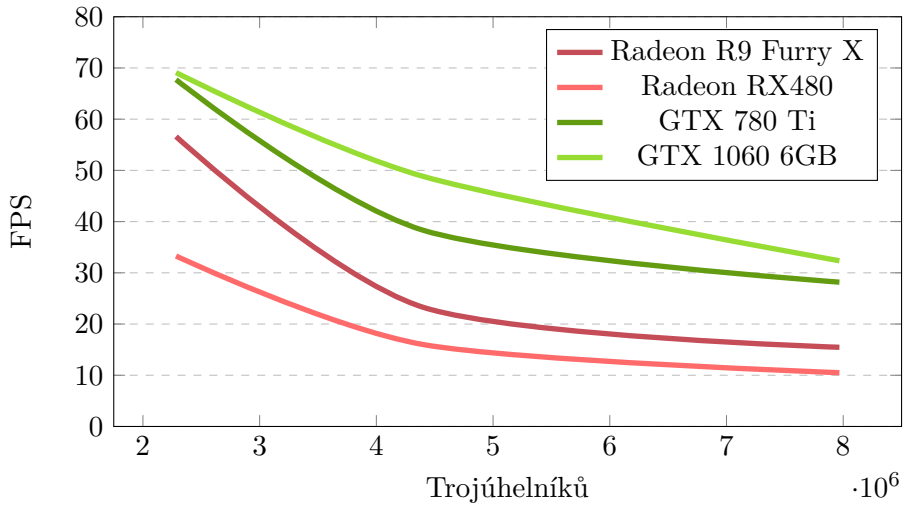
Tabulka 5.6: Rychlost vykreslování s GeForce GTX 780 Ti.

První sestava byla také použita k bližší inspekci náročnosti aplikace, jejíž výsledky jsou v tabulce 5.7. Pro zjišťování vytížení CPU a RAM byl použit nástroj ProcessExplorer od společnosti Sysinternals Software. Utilizace GPU byla zjišťována přes nástroj EXPERTool II dodávaný přímo výrobcem grafické karty, společností Gainward.

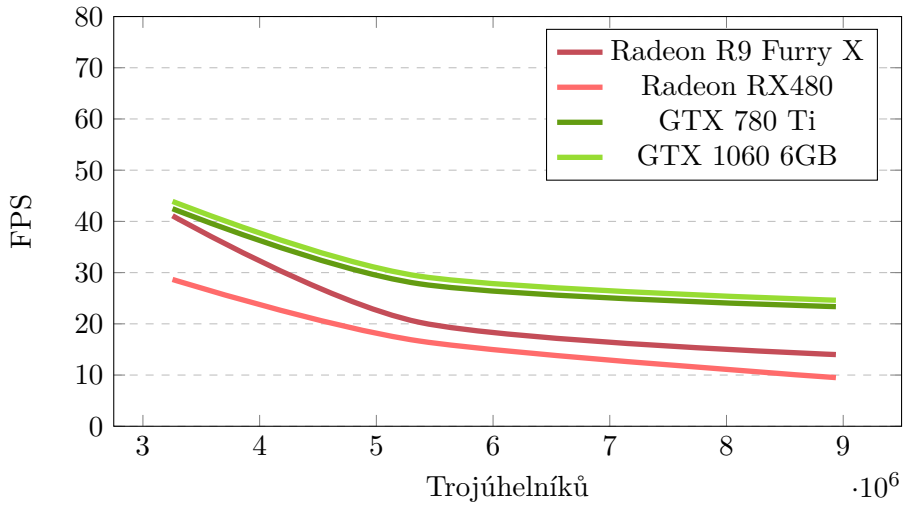
Sada obj. číslo	Rozlišení	% utilizace GPU jádra	% utilizace GPU VRAM	% utilizace CPU	MB utilizace RAM
1	1280 x 720	91	17	13,340	615
1	1680 x 1050	94	55	15,500	743
1	1920 x 1080	96	23	13,340	778
2	1280 x 1720	92	14	12,430	729
2	1680 x 1050	95	15	12,900	848
2	1920 x 1080	96	17	13,050	890
3	1280 x 720	92	14	12,970	854
3	1680 x 1050	95	15	13,150	973
3	1920 x 1080	96	17	13,610	1 023

Tabulka 5.7: Naměřené hodnoty při vykreslování za pomoci testovací sestavy 1.

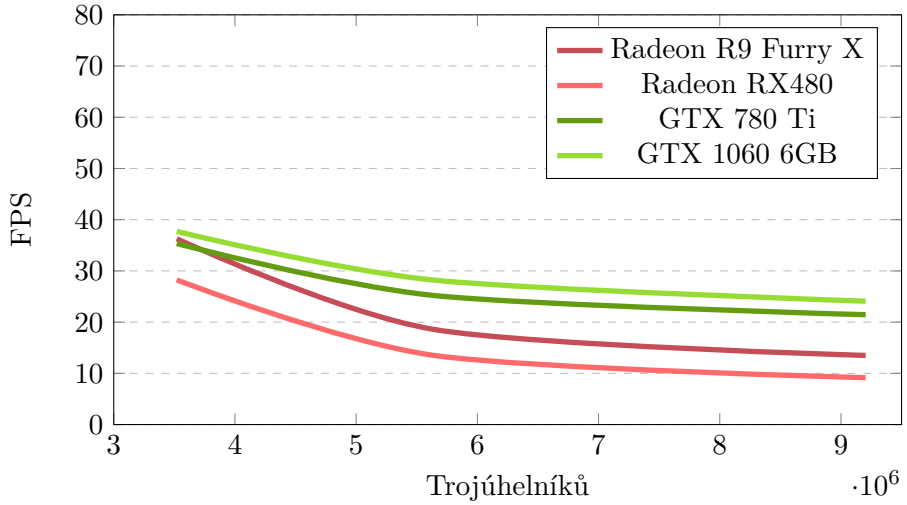
Následně jsem z naměřených hodnot vytvořil graf pro každé rozlišení (1280 x 720 graf 5.1, 1680 x 1050 graf 5.2, 1920 x 1080 graf 5.3). Tyto grafy ukazují pokles výkonnosti s rostoucím počtem objektů scény. Dále ukazují srovnání výkonnosti jednotlivých výrobců a jednotlivých grafických karet.



Graf 5.1: Graf znázorňující počet FPS v závislosti na počtu trojúhelníků scény při rozlišení 1280 x 720.



Graf 5.2: Graf znázorňující počet FPS v závislosti na počtu trojúhelníků scény při rozlišení 1680 x 1050.



Graf 5.3: Graf znázorňující počet FPS v závislosti na počtu trojúhelníků scény při rozlišení 1920 x 1080.

Kapitola 6

Závěr

V rámci práce byly shrnuty moderní přístupy ke tvorbě grafických efektů. Dále byly diskutovány stěžejní problémy aktuálních API. Také jsem vytvořil funkční prototyp herního světa s fyzikálně simulovanou oblohou, výpočtem osvětlení využívajícím CSM a další postprocesové efekty, výpočtem kolizí, procedurálně generovanou trávou a světem obsahujícím velké množství objektů. Výpočet oblohy poměrně kopíroval reálnou oblohu a vytvořil tak oku lahodící scenérii. Díky CSM spojenými s PCF byly vytvořené stíny dostatečné jemné, s lehkou simulací měkkých stínů. Vykreslovaný terén využívá výškovou mapu s rozlišením 4096 x 4096, kdy jeden pixel odpovídá jednomu vrcholu. Práce celkově obsahuje okolo 10 000 řádků kódu.

Velký počet objektů scény se nakonec ukázal jako Achillova pata aplikace. Premisa výrazného nezrychlení scény pomocí frustrum cullingu byla odhalena jako mylná. Dále, přestože se nepřímé vykreslování jeví jako spásná myšlenka, jeho sekvenční přístup má i slabiny. Rozšířením práce by mohlo být rozbití jednoho objektu, který spravuje a vykresluje modely celé scény, na několik objektů odpovídajících jednotlivým částem grafu scény. Tyto objekty by se daly efektivněji vykreslovat a ignorovat ty, které nejsou vidět. Nemusí se jednat přímo o streamování objektů, jelikož s pamětí u moderních karet problém nebyl.

Pozitivním zjištěním byl malý dopad generování trávy na rychlost celé aplikace. Tato tráva poskytovala iluzi husté trávy tvořené jednotlivými stébly. Nedostatkem, kterým mnou implementovaná tráva trpí, je prorůstaní skrz objekty. Toto je způsobeno stylem výběru bodů pro seedy, protože jsou vybírány jen na základě materiální textury terénu. V dalším vývoji procedurálně generované trávy by bylo vhodné zohledňovat i pozice objektů.

Velice překvapivým zjištěním, avšak v negativním smyslu, byly naměřené hodnoty FPS při vykreslování scény. Při tomto měření se ukázaly AMD karty až dvakrát pomalejší oproti kartám od firmy Nvidia. Kdy dokonce karta z rozdílné výkonnostní kategorie porazila AMD kartu. Tudíž se celá záležitost jeví jako problém rozdílnosti obou architektur. Tento fenomén by si zasluhoval hlubší prozkoumání, ale bohužel v této práci není na něj již místo. Naopak pozitivně zapůsobilo srovnání starší a novější generace Nvdia. A celkově na kartách Nvidia aplikace dosahovala výkonů, které lze považovat za dostatečné pro hraní.

Možných rozšíření práce lze nalézt nepřeberné množství. Předně výše zmíněné upravené vykreslování dělené dle prostorových částí. Dále je to navázání na vzniklou práci rozšířením na herní engine např. přidáním animací, počítačově řízených postav, vylepšení kolizí pro práci i nad jednotlivými trojúhelníky, destrukce prostředí. Potažmo vytvořením celé hry nad rozšířeným jádrem.

Literatura

[1] Annen, T.; Mertens, T.; Bekaert, P.; aj.: Convolution shadow maps. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, Eurographics Association, 2007, s. 51–60.
URL http://jankautz.com/publications/csmEGSR07.pdf

[2] Bruneton, E.; Neyret, F.: Precomputed Atmospheric Scattering. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, s. 1079–1086, doi:10.1111/j.1467-8659.2008.01245.x.
URL http://dx.doi.org/10.1111/j.1467-8659.2008.01245.x

[3] Cantlay, I.: Directx 11 terrain tessellation. *Nvidia whitepaper*, ročník 10, 2011: str. 12, [Online; navštíveno 5.11.2016].
URL http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf

[4] Dimitrov, R.: Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.
URL http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

[5] Eisemann, E.; Schwarz, M.; Assarsson, U.; aj.: *Real-Time Shadows*. Natick, MA, USA: A. K. Peters, Ltd., první vydání, 2011, ISBN 1568814380, 9781568814384.

[6] Ericson, C.: The Gilbert-Johnson-Keerthi algorithm. Technická zpráva, Technical report, Sony Computer Entertainment America, 2005.

[7] Everitt, C.; Foley, T.; McDonald, J.; aj.: Approaching Zero Driver Overhead. [Online; navštíveno 4.1.2017].
URL http://gdcvault.com/play/1020791/

[8] Gribb, G.; Hartmann, K.: Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.
URL http://www.cs.otago.ac.nz/postgrads/alexis/planeExtraction.pdf

[9] Kajiya, J. T.: The rendering equation. In *ACM Siggraph Computer Graphics*, ročník 20, ACM, 1986, s. 143–150.

[10] Lokovic, T.; Veach, E.: Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 2000, s. 385–392.

URL
http://pellacini.di.uniroma1.it/teaching/topics08/papers/lokovic00.pdf

[11] Mittring, M.: Finding Next Gen: CryEngine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA: ACM, 2007, ISBN 978-1-4503-1823-5, s. 97–121, doi:10.1145/1281500.1281671.
URL http://doi.acm.org/10.1145/1281500.1281671

[12] Nishita, T.; Sirai, T.; Tadamura, K.; aj.: Display of the Earth Taking into Account Atmospheric Scattering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, New York, NY, USA: ACM, 1993, ISBN 0-89791-601-8, s. 175–182, doi:10.1145/166117.166140.
URL http://doi.acm.org/10.1145/166117.166140

[13] Pavlík, V.: Animovaný skybox v OpenGL. 2013.
URL http://hdl.handle.net/11012/54825

[14] Perlin, K.: An Image Synthesizer. *SIGGRAPH Comput. Graph.*, ročník 19, č. 3, Červenec 1985: s. 287–296, ISSN 0097-8930, doi:10.1145/325165.325247.
URL http://doi.acm.org/10.1145/325165.325247

[15] Perlin, K.: Improving Noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-521-1, s. 681–682, doi:10.1145/566570.566636.
URL http://doi.acm.org/10.1145/566570.566636

[16] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005, ISBN 0321335597.

[17] Preetham, A. J.; Shirley, P.; Smits, B.: A Practical Analytic Model for Daylight. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, ISBN 0-201-48560-5, s. 91–100, doi:10.1145/311535.311545.
URL http://dx.doi.org/10.1145/311535.311545

[18] Reeves, W. T.; Salesin, D. H.; Cook, R. L.: Rendering Antialiased Shadows with Depth Maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA: ACM, 1987, ISBN 0-89791-227-6, s. 283–291, doi:10.1145/37401.37435.
URL http://doi.acm.org/10.1145/37401.37435

[19] Shreiner, D.; Sellers, G.; Kessenich, J. M.; aj.: *OpenGL programming guide*. Upper Saddle River, NJ: Addison-Wesley, eighth edition. vydání, [2013], ISBN 03-217-7303-9.

[20] Widmark, M.: Terrain in Battlefield 3: A modern, complete and scalable system. 3 2012, přednáška z GDC [Online; navštíveno 5.11.2016].
URL http://www.frostbite.com/wp-content/uploads/2013/05/GDC12_Terrain_in_Battlefield3.pdf

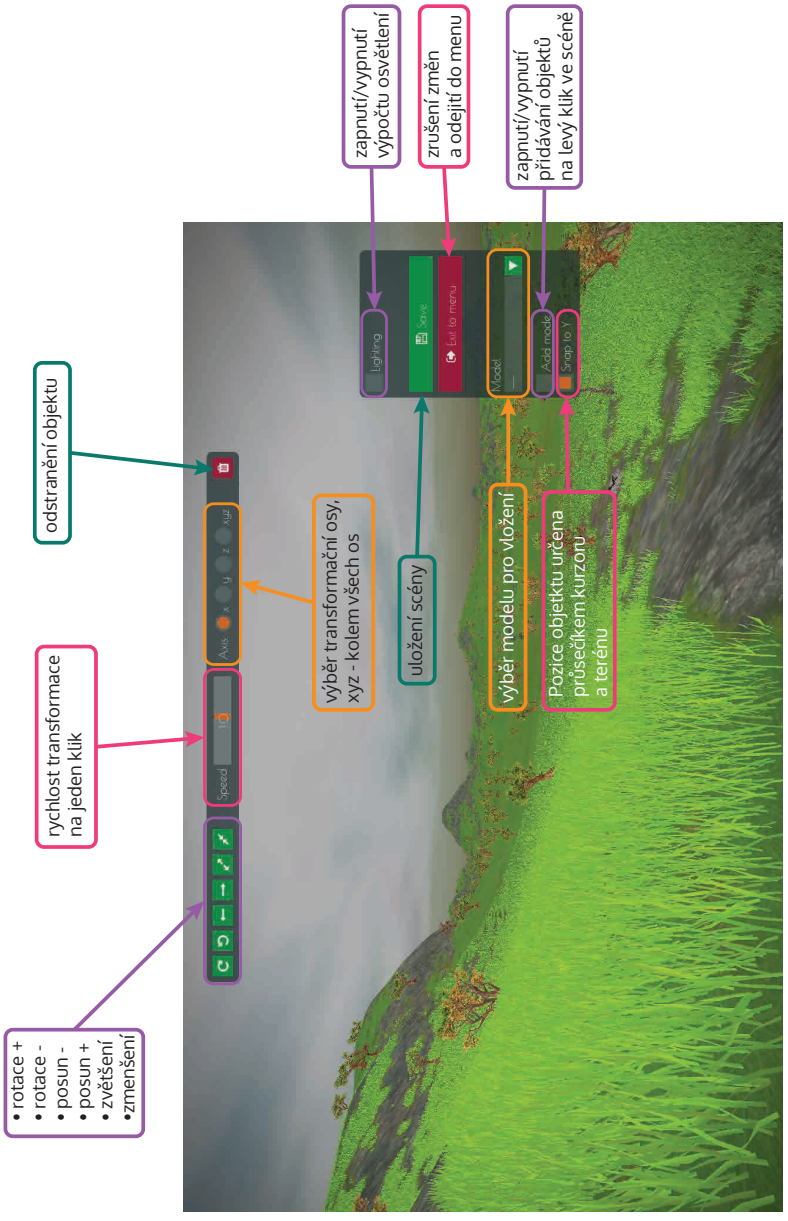
[21] Wright, R. S.; Sellers, G.; Haemel, N.: *OpenGL superbible*. Upper Saddle River, NJ: Addison-Wesley, sixth edition / vydání, [2014], ISBN 03-219-0294-7.

[22] Zachmann, G.; Langetepe, E.: *Geometric data structures for computer graphics*. Eurographics Assoc., 2003.
URL http://www.informatik.uni-bremen.de/~zach/papers/talks/geom_data_structures/siggraph03_course16_onscreen.pdf

Přílohy

Příloha A

Popis rozhraní editoru



Příloha B

Obsah DVD a návod na spuštění

B.1 Spuštění aplikace

Pro spuštění aplikace je nutno nakopírovat složku `/bin` na místo, kde je povolen i zápis. Je to z důvodu zápisu do konfiguračních souborů.

Ve složce `/bin` je soubor `Launcher.exe`, který spouští aplikaci. Jedná se o bat soubor převedený na exe, jehož jedinou činností je spuštění souboru `SAR Officer.exe` ze složky `/bin/bin` .

B.2 Obsah DVD

Obsah DVD je složen z následujících adresářů a souborů:

/Bin
: Zkompilovaná verze herního světa a datové zdroje.

/Src
: Zdrojové soubory aplikace včetně projektu ve VS 2015, všech datových zdrojů a knihoven. Dále je zde i složka `help` se souborem `doxygen`. Dokumentaci lze generovat jen Doxygenem verze <= 1.8.11.

/Help
: HTML dokumentace vygenerovaná doxygenem.

/Doc
: Technická zpráva dokumentace.

/Tex
: Zdrojové kódy technické zprávy včetně zdrojových obrázků.

Trailer.mp4
: Video plakát aplikace. Jeho rozlišení je 1920 x 1080.