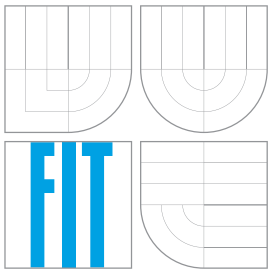
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

GRAPHIC INTRO 64KB USING OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR OBRTLÍK

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. ADAM HEROUT, Ph.D.

BRNO 2016

Zadání bakalářské práce/18269/2015/xobrtl02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Obrtlík Petr**
Obor: Informační technologie
Téma: **Grafické intro 64kB s použitím OpenGL**
Graphics Intro 64kB Using OpenGL
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s fenoménem grafického intra s omezenou velikostí.
2. Prostudujte knihovnu OpenGL a její nadstavby.
3. Popište vybrané techniky použitelné v grafickém intru s omezenou velikostí.
4. Implementujte grafické intro s použitím OpenGL, aby velikost spustitelné verze nepřesáhla 64kB.
5. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, experimenty směřující k vyřešení bodu 4.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: **Herout Adam, doc. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

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 bakalářská práce je zaměřena na problematiku tvorby grafického intra s omezenou velikostí. Práce popisuje využívané metody pro tvorbu grafického intra. Zabývá se procedurálním generováním objektů a textur, jednoduchým generováním částic, vykreslováním vodní hladiny s reflexí a refrakcí, vytvářením stínů, osvětlení, nastavením kompilátoru a komprimací výsledného souboru. Výsledkem je kreslená krajinka simulující průběh dne. Vytvořené grafické intro má velikost menší než 64kB.

Abstract

This bachelor's thesis deals with issue of graphic intro with limited size. The work describes methods used for creating of graphic intro. It deals with procedural generation of objects and textures, simple generation of particles, rendering water surface with reflection and refraction, shadows, lighting, compiler settings and compression output file. The result is a cartoon landscape simulating progress of the day. Created graphic intro have size smaller than 64kB.

Klíčová slova

grafické intro, OpenGL, omezená velikost, Perlinův šum, skybox, částicové systémy, Phongův osvětlovací model, stínovací techniky, cel-shading, billboarding, komprese

Keywords

graphic intro, OpenGL, limited size, Perlin noise, skybox, particle systems, Phong lighting model, shadow mapping, cel-shading, billboarding, compresion

Citace

OBRTLÍK, Petr. *Grafické intro 64kB s použitím OpenGL*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Herout Adam.

Grafické intro 64kB s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Obrtlík
16. května 2016

Poděkování

Rád bych poděkoval prof. Ing. Adamu Heroutovi, Ph.D. za jeho cenné rady během celého projektu a rodině za psychickou podporu.

Obsah

1 Úvod **2**
- 1.1 Historie inter . . . 2
- 1.2 Cíl práce . . . 2

2 OpenGL a metody používané v grafickém intru s omezenou velikostí **3**
- 2.1 OpenGL . . . 3
- 2.2 Procedurální generování . . . 5
- 2.3 Osvětlovací model . . . 9
- 2.4 Billboarding . . . 11
- 2.5 Skybox . . . 11
- 2.6 Částicové systémy . . . 12
- 2.7 Cel-shading . . . 13
- 2.8 Shadow mapping . . . 13

3 Implementace Intra **15**
- 3.1 Knihovny . . . 15
- 3.2 Generování terénu . . . 15
- 3.3 Generování vody . . . 17
- 3.4 Viditelnost . . . 20
- 3.5 Skybox . . . 20
- 3.6 Slunce . . . 22
- 3.7 Oblaka . . . 22
- 3.8 Dým z komínu . . . 24
- 3.9 Tvorba chaloupky a mola . . . 24
- 3.10 Hudba . . . 25
- 3.11 Pohyb kamery ve scéně . . . 26

4 Metody pro omezení výsledné velikosti aplikace **27**
- 4.1 Nastavení překladače . . . 27
- 4.2 exe packery . . . 27

5 Výsledek **29**
- 5.1 Rychlost vykreslování . . . 29
- 5.2 Výsledná scéna . . . 30

6 Závěr **31**

Literatura **32**

Kapitola 1

Úvod

Grafické intro je upoutávka či demo, které má zaujmout a prezentovat jak programátorské schopnosti, tak i umělecké nadání svého tvůrce. Zpravidla se dříve objevovala před startem aplikací a obsahovala zajímavé scény, efekty a zvukový doprovod.

Velikost 64kB se na první pohled může zdát velmi malá při představě, že se jedná o grafickou aplikaci, zvláště v dnešní době, kdy grafické aplikace, jako jsou počítačové hry, přesahují i desítky gigabajtů dat. Rozdíl je v tom, že veškerá data modelů se generují až za běhu programu a nejsou zde žádné statické modely, které by zabíraly spoustu místa.

1.1 Historie inter

Vznik prvních grafických inter je spojený se vznikem počítačového pirátství. Po prolomení obrany proti kopírování aplikace vkládali autoři inter svá díla před samotnou aplikaci. Cílem bylo se podepsat, aby každý věděl, kdo aplikaci prolomil. Někdy se i stávalo, že grafické intro vypadalo lépe než samotná počítačová hra.

V dnešní době jsou intra většinou vytvářena jako samostatná díla, kde je hlavní důraz kladen na uměleckou a technologickou složku díla. Autoři mezi sebou pořádají rozličné soutěže v různých kategoriích podle velikostí.

1.2 Cíl práce

Cílem této práce je vytvoření jednoho takového intra s velikostí omezenou na 64kB. V mém intru se bude jednat o přírodní scenérii. Základním prvkem bude terén s rybníkem uprostřed scény, chaloupkou a molem. Okraj bude ohraničen kopci, které budou mizet v dálce. Ve scéně se bude střídat den a noc, kde se budou lišit některé objekty. V noci se bude svítit v chaloupce, dýmit z komína a na obloze budou vidět hvězdy. Ve dne budou poletovat mraky na obloze a zobrazovat jednotlivé stíny objektů na terénu. Výsledkem práce je grafické intro, jehož velikost nepřesáhla 64kB. Nebude taktéž příliš zaměřeno na reálnost, ale ztvárněno spíše v naivním malířském stylu.

Kapitola 2

OpenGL a metody používané v grafickém intru s omezenou velikostí

Kapitola se zabývá problematikou tvorby grafických inter s omezenou velikostí a metodami, pomocí kterých se výsledné intro vytváří. Jsou zde popsány prvky OpenGL, způsoby vytváření modelů a metody pro osvětlení scény.

2.1 OpenGL

OpenGL je zkratka pro Open Graphic Library a jedná se o platformě nezávislé aplikační rozhraní pro tvorbu 2D a 3D grafiky. Umožňuje vykreslování různých primitiv, jako jsou například body, přímky a obdélníky pixelů.

Knihovna OpenGL byla navržena firmou SGI (Silicon Graphics Inc.) jako API (aplikační programové rozhraní). Z programátorského hlediska se OpenGL chová jako stavový automat. To znamená, že během zadávání příkazů pro vykreslování lze průběžně měnit vlastnosti vykreslovaných primitiv, jako například barvu a průhlednost.

OpenGL nezajišťuje práci s okny, kvůli tomu se musí sáhnout po nějaké knihovně, která to dokáže, jako například GLUT, WinAPI nebo SDL. Jelikož ale GLUT a SDL nejsou součástí Windows, musela by se přikládat knihovna, proto jsem se rozhodl využít WinAPI, která je již součástí Windows.

2.1.1 GLSL

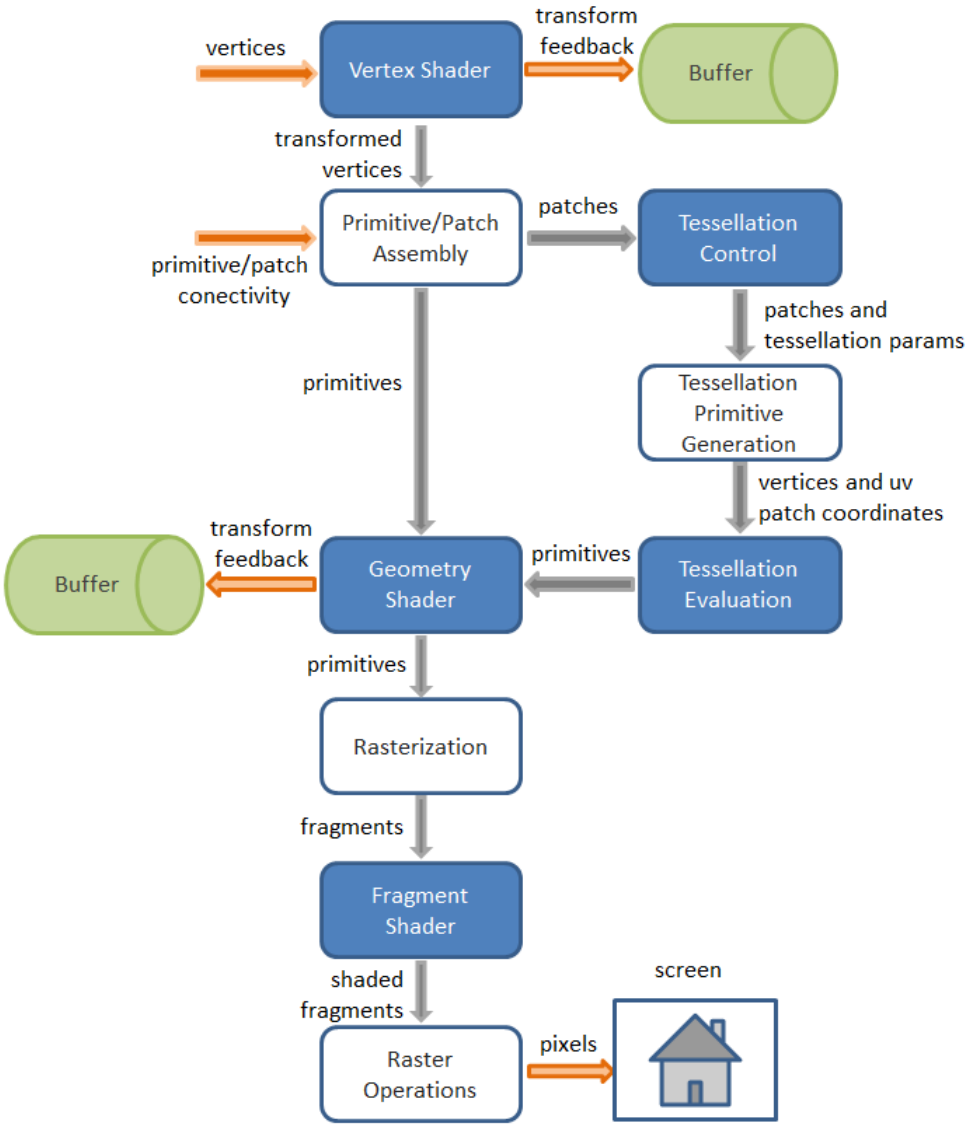
OpenGL Shading Language je nedílnou součástí OpenGL API. Jedná se o programovací jazyk velmi podobný jazyku C, obsahující specifické prvky pro grafiku. Slouží pro psaní tzv. shaderů.

GLSL obsahuje skalární datové typy a operátory známé z jazyka C, výjimku tvoří ukazatele, které zde nejsou podporovány. Důležitou součástí jsou vektory a matice. GLSL podporuje jedno- až čtyř-složkové vektory (*vec1* - *vec4*) a 2x2 až 4x4 matice (*mat2* - *mat4*). Dále je obsažen datový typ *sampler*, který slouží pro uložení textury. Pro rychlý přístup k prvkům vektorů se zde nachází operátor tzv. "swizzle". Pomocí tohoto operátoru lze přistupovat k prvkům vektoru v daném pořadí. Například *vec4.xyz* vytvoří *vec3* bez složky w, případně můžeme zaměnit pořadí, např. *vec4.yxz* zamění x za y a obráceně.

2.1.2 Shadery

Shader je malý počítačový program sloužící k řízení jednotlivých částí programovatelného grafického řetězce grafické karty. Existuje několik typu shaderů, každý pro specifickou funkci a účel. Kód shaderu může být uložen v samostatném souboru nebo jako textový řetězec v rámci programu. Pro větší přehlednost a lepší práci s nimi se většinou shadery umisťují do samostatných souborů. Při práci jsem je nejprve také ukládal do souborů, ale nakonec jsem je umístil do textového řetězce v rámci programu. Kvůli tomu aby pro spuštění aplikace nebyly třeba žádné další soubory.

Jednotlivé shadery jsou řazeny za sebe a výstup jednoho shaderu je vstupem dalšího shaderu.



Obrázek 2.1: Vykreslovací řetězec (převzato z [3])

Na obrázku 2.1 je znázorněný tzv. *pipilene* (vykreslovací řetězec), to je zřetězení shaderů za sebou. Modře jsou znázorněny programovatelné shadery a bíle ty, ke kterým nemá programátor přístup. Jak již bylo zmíněno každý shader má vlastní funkci, nyní si některé

shadery více přiblížíme.

Vertex Shader

Vertex Shader je základní shader, který zpracovává vrcholy přicházející ze strany CPU. Ty zde mohou byt transformovány např. pomocí modelové, projekční či pohledové matice. Modelová matice se využívá k manipulaci s objekty (modely) ve scéně, pohledová pro nastavení pozice kamery a projekční pro nastavení perspektivní nebo ortogonální projekce. Určuje se zde tedy výsledná poloha primitiva ve scéně.

Lze zde dosáhnout i různých grafických efektů, jako například simulace pohybu vodní hladiny.

Geometry shader

Na rozdíl od vertex shaderu umožňuje geometry shader přidávání a odebírání vrcholů. Využívá se často pro generování jednoduché vegetace (např. trávy) nebo pro billboarding popsaný v kapitole 2.4.

Fragment shader

Z hlediska vykreslování je to nejdůležitější shader. Určuje se zde výsledná barva bodu. Mezi nejčastější operace prováděné v tomto shaderu patří například aplikace textury na objekt nebo výpočet osvětlení.

2.2 Procedurální generování

Jelikož grafické intro má omezenou velikost, musí být veškerý obsah vytvářen procedurálně, tedy pomocí algoritmů. Pokud bychom načetli nějaký model případně texturu ze souboru, zabrala by většinu limitující velikosti nebo i více místa a na jiný obsah by nezbylo místo.

Výhodou procedurálního generování může být i to, že výsledný objekt lze ovlivnit jednoduše pomocí změny parametrů. Velmi důležité je, aby výsledná scéna byla vždy stejná, proto je nutné využívat místo čísel náhodných čísla pseudonáhodná, tedy generována deterministicky.

2.2.1 Generování náhodných čísel

Uplatnění náhodných čísel v informatice je široké. Největší uplatnění mají zejména v oborech šifrování, počítačových simulací nebo modelování. Náhodná čísla jako taková se velmi těžko generují a využívají se k tomu většinou fyzikální jevy. Softwarové generátory náhodných čísel vytváří pseudonáhodná čísla. Tyto posloupnosti nejsou tedy skutečně náhodné, ale pouze tak vypadají. Do jisté míry jde náhodnost přidat pomocí hodin v počítači, které poskytují dostatečný faktor náhodnosti, ale pro modelovaní se příliš nevyužívají, protože výsledek se při každém spuštění liší.

S řešením problému generování náhodných čísel přišel Ken Perlin, který vytvořil metodu zvanou Perlinův šum. Ta pro stejný vstup vytvoří pokaždé stejný výsledek a pro podobné vstupy vytváří i podobné výsledky.

Jeden z nejznámějších pseudonáhodných generátorů je lineární kongruentní generá-

tor [11]:

$$X_{n+1} = (aX_n + b) \mod m, \tag{2.1}$$

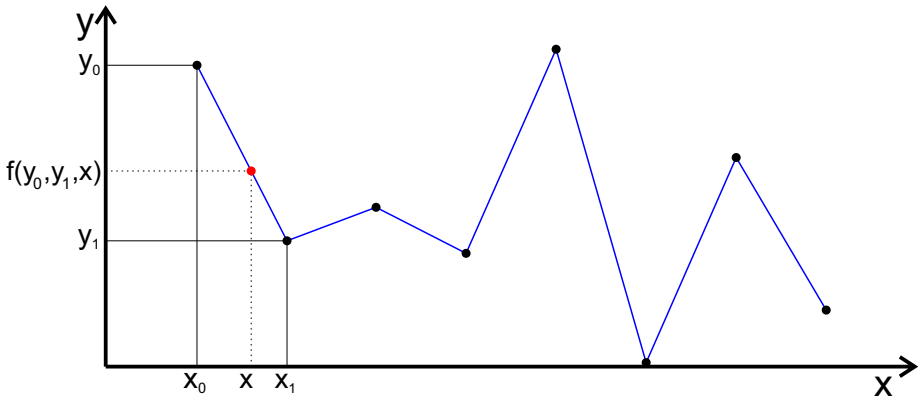
kde a, c a m jsou vhodně zvolené konstanty a m zároveň určuje omezení intervalu shora, x_n je vstupní hodnota a x_{n+1} výstupní hodnota. Omezení rozsahu generátoru závisí na architektuře počítače.

2.2.2 Interpolace

Interpolace je matematický proces, za pomocí kterého vypočítáváme body na křivce mezi body které známe, tudíž nám stačí jen pár bodů pro popsání celé křivky. Existuje mnoho typů interpolace, které se liší přesností a časovou náročností.

Lineární interpolace

Nejjednodušší a nejméně náročná metoda je Lineární interpolace



Obrázek 2.2: Lineární interpolace

Jak je vidět na obrázku 2.2, jestliže existují dva body (x_0, y_0) a (x_1, y_1), tak lineární interpolací mezi nimi je přímka.

Body na přímce se vypočítají pomoci následujícího vztahu:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0} \tag{2.2}$$

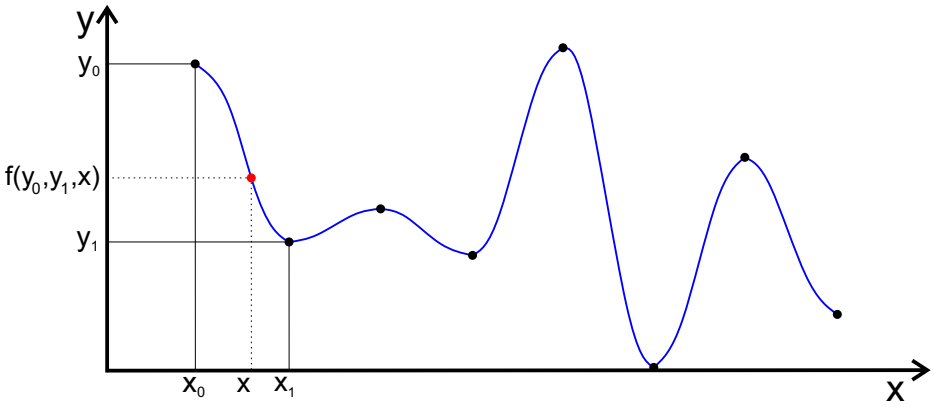
Po úpravě rovnice se získá vztah:

$$f(y_0, y_1, x) = y_0(1 - x) + y_1 x \tag{2.3}$$

Výhodou lineární interpolace je rychlost, v praxi se zejména využívá tam, kde není tak důležitá přesnost, ale je kladen důraz na vyšší rychlost výpočtu. Nevýhodou jsou ostré hrany, jak si lze všimnout na obrázku 2.2.

Kosinová interpolace

Jedná se o vylepšení lineární interpolace, která řeší problém ostrých přechodů v bodech



Obrázek 2.3: Kosinová interpolace

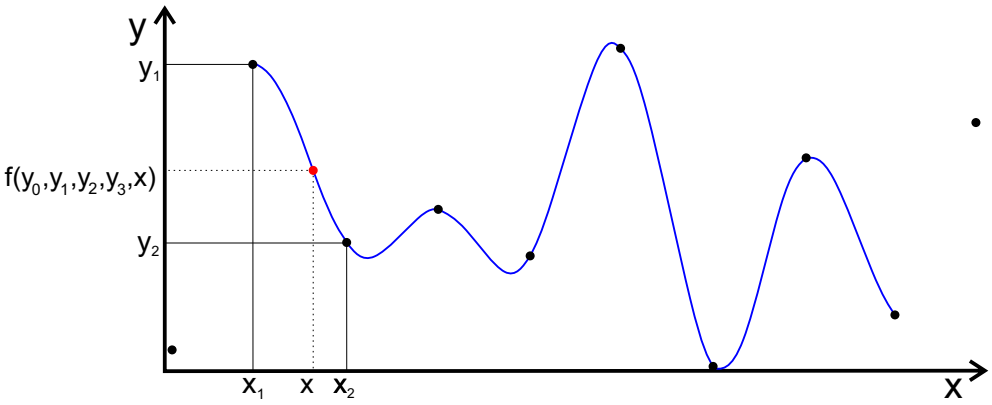
za cenu jen o něco pomalejšího výpočtu. Jedná se o řešení založené na použití funkce kosinus mezi dvěma body. Řešení je podobné lineární interpolaci.

$$f(x) = \frac{1 - \cos(x\pi)}{2} \tag{2.4}$$

$$f(y_0, y_1, x) = y_0(1 - f(x)) + y_1 f(x) \tag{2.5}$$

Kubická interpolace

Kubická interpolace vytváří velmi jemné výsledky, i když na úkor rychlosti, místo dvou bodů se zde potřebují čtyři, ale výsledek této metody je velmi podobný kosinově interpolaci.



Obrázek 2.4: Kubická interpolace

$$P = (y_3 - y_2) - (y_0 - y_1) \tag{2.6}$$

$$f(y_0, y_1, y_2, y_3, x) = Px^3 + ((y_0 - y_1) - P)x^2 + (y_2 - y_0)x + y_1 \tag{2.7}$$

2.2.3 Perlinův šum

Perlinův šum [10] je metoda pro generování grafického šumu, kterou roku 1985 představil Ken Perlin. V podstatě se jedná o deterministický generátor náhodných čísel vhodný pro generování grafického šumu. Deterministický znamená, že pokaždé se dostane stejný výsledek pro stejný vstup. Výsledkem generátoru jsou pseudonáhodné hodnoty v rozsahu <-1,1>.

Výstup generátoru by nemusel být dostačující, proto je dále interpolován mezi okolními body, což zajistí již hladší výstup. V tomhle případě byla vybrána kosinová interpolace, která je dostatečně rychlá a vytváří dostačující výsledek.

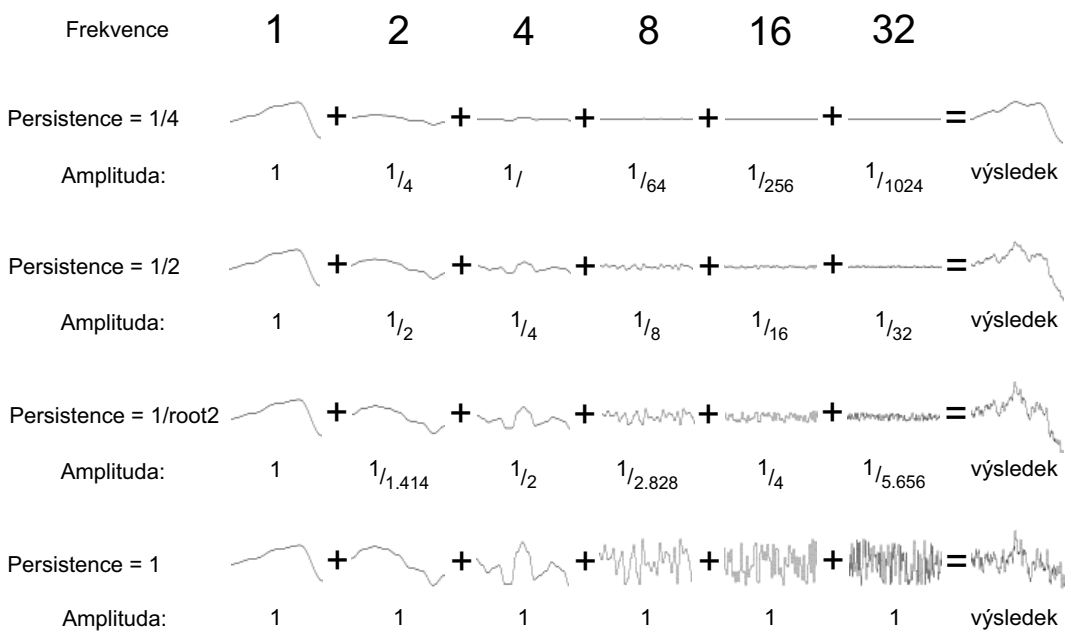
Dalším krokem je součet několika takových výstupů dohromady, neboli oktáv, kde každá funkce má jinou amplitudu a frekvenci. Každá z nich znázorňuje určitou úroveň detailů. Amplitudu mezi jednotlivými šumy lze vyjádřit jako

$$a = p^i \tag{2.8}$$

a frekvenci jako

$$f = 2^i, \tag{2.9}$$

kde p je persistence.



Obrázek 2.5: Vliv persistence na výsledek (převzato z [10])

Na obrázku 2.5 lze vidět vliv persistence na amplitudu a výsledný součet. Lze tu taky vidět, že pokud zůstává amplituda stejná, persistence je rovna jedné, a mění se jen frekvence, tak vyhlazování nemá příliš smysl, protože mezi jednotlivými body jsou velmi ostré přechody. Těmto stavům by bylo dobré se vyhýbat, protože nemají velký význam. Dále je zde vidět že se výsledek od určitého počtu oktáv (šumových funkcí) příliš neliší. V praxi se vhodný počet oktáv určuje experimentálně.

Na obrázku 2.6 jsou vlevo znázorněny šumové funkce, které se sčítají. Napravo je znázorněn výsledný šum vzniklý jejich součtem. Výsledkem součtu šumových funkci je jeden Perlinův šum, kterým je možné napodobit různé přírodní materiály.

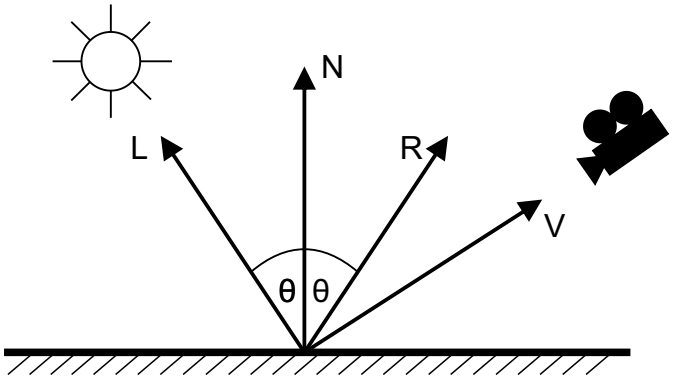
Obrázek 2.6: Sčítání šumových funkí do jedné (převzato z [10])

2.3 Osvětlovací model

Osvětlovací modely v počítačové grafice popisují intenzitu světla v jednotlivých bodech scény. Osvětlovacích modelů existuje spousta, rozdělují se na fyzikální a empirické modely.

2.3.1 Phongův osvětlovací model

Phongův osvětlovací model je emperický, tedy není založen na fyzikálních vlastnostech, ale i přesto vytváří velmi realistické efekty. Má velké uplatnění v real-time grafice, hlavně kvůli tomu, že je výpočetně nenáročný. Phongův model byl navržen vietnamským vědcem Bui Tuong Phongem roku 1973 v rámci jeho disertační práce [18].



Obrázek 2.7: Vektorové složky osvětlovacího modelu

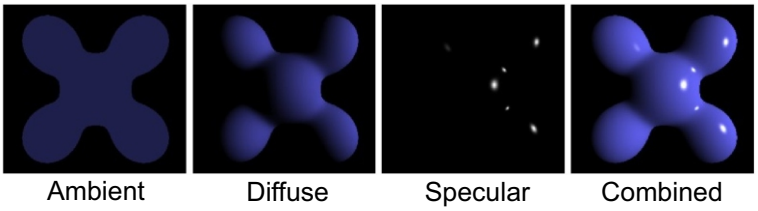
Na obrázku 2.7 jsou znázorněny důležité vektory pro výpočet. Vektor $\overrightarrow{N}$ je normálou v bodě objektu, vektor $\overrightarrow{L}$ je vektor ke zdroji světla a vektor $\overrightarrow{R}$ je jeho odraz. V poslední řadě vektor $\overrightarrow{V}$, který znázorňuje vektor k pozorovateli, případně kameře.

Odraz světla se v Phongově modelu skládá ze tří světelných složek, přesněji ambient, difúzní a spekulární. Výsledný odraz je pak součet těchto složek:

$$C = C_a + C_d + C_s \tag{2.10}$$

Světelná složka ambient

Světelná složka ambient vyjadřuje okolní světlo, které není přímo vyzařováno ze světelného zdroje. V Phongově osvětlovacím modelu je to světlo, které osvětluje objekt ze všech směrů nezávisle na poloze a orientaci zdrojů světla. Zajišťuje tedy, že i část objektu, která



Obrázek 2.8: Jednotlivé složky Phongova osvětlovacího modelu (převzato z [20])

není osvětlena, bude mít jinou barvu než černou (pokud to není požadováno). Je dána vztahem:

$$C_a = I_a \cdot k_a, \tag{2.11}$$

kde I_a je intenzita okolního osvětlení scény a k_a je odrazivý koeficient, který nabývá hodnoty v intervalu $\langle 0, 1 \rangle$, kde 0 značí, že se okolní světlo od povrchu objektu vůbec neodráží, naopak 1 značí, že se světlo zcela odráží od povrchu. Určuje tedy výsledný odstín.

Difúzní světelná složka

Difúzní složka představuje odraz světla pocházejícího ze zdroje světla. Od povrchu tělesa se odráží do všech směrů. Pomocí difúzní složky můžeme zobrazit již trojrozměrné objekty, ale matného vzhledu. Difúzní složka je dána vztahem:

$$C_d = I_d \cdot k_d(\overrightarrow{N} \cdot \overrightarrow{L}), \tag{2.12}$$

kde I_d je intenzita difúzního osvětlení scény, k_d je stejné jako k_a u ambient složky, tedy odrazivý koeficient se stejným intervalem. $\overrightarrow{N}$ je normála v bodě a $\overrightarrow{L}$ je směr dopadu paprsku. Čím více se normalizovaný vektor $\overrightarrow{L}$ blíží k normalizovanému vektoru $\overrightarrow{N}$, tím je větší množství odraženého světla.

Spekulární světelná složka

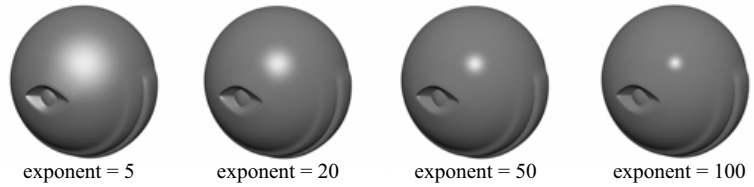
Spekulární (lesklá) složka udává intenzitu té části světla, která se od tělesa odráží převážně v jednom směru podle zákona odrazu. Podle vztahu:

$$C_s = I_s \cdot k_s(\overrightarrow{V} \cdot \overrightarrow{R})^n, \tag{2.13}$$

kde I_s a k_s mají stejný význam jak v ambient a difúzní složce, $\overrightarrow{V}$ udává směr odkud bod na povrchu tělesa pozorujeme a $\overrightarrow{R}$ je odražený směr světla, který se vypočítá pomocí vztahu:

$$\overrightarrow{R} = 2(\overrightarrow{N} \cdot \overrightarrow{L})\overrightarrow{N} - \overrightarrow{L}, \tag{2.14}$$

a n značí Phongův exponent, který určuje ostrost odrazu. Čím je vyšší, tím jsou odlesky menší, ale intenzivnější.

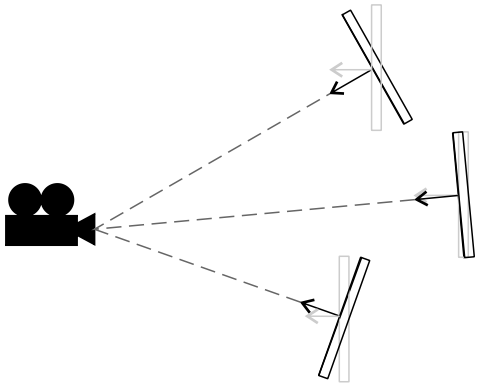


Obrázek 2.9: Vliv Phongova exponentu na ostrost a intenzivnost odlesku (převzato z [13])

2.4 Billboarding

Billboarding [4] je technika, ve které jsou 3D objekty zobrazující efekty nanášeny na 2D plochu. Billboarding se používá například k nahrazení složité geometrie nanesením textury na čtverec, tvořený dvěma trojúhelníky, a ušetří tak počet polygonů na scéně, tudíž i výpočetní výkon.

Billboarding se často využívá k modelování vegetace, kouře, ohně a jiných objektů. 3D dojem se docílí tím, že čtverec je neustále natočen čelem ke kameře a uživatel si díky tomu neuvědomí, že se jedná pouze o čtverec, ale připadá mu jako složitý 3D model. Problémem může být například textura stromu, kde při přeletu kamery nad objektem je výsledný dojem ztracen. Tato technika se využívá například i u částicových systémů, které jsou popsány v kapitole 2.6.



Obrázek 2.10: Billboarding - šedě jsou znázorněny původní směry natočení textur, černě již natočené směrem ke kameře.

2.5 Skybox

Skybox [21] je velká krychle, která obklopuje celou scénu a poskytuje iluzi, že okolní svět se zdá být rozlehlejší než ve skutečnosti. Skládá se z šesti textur (případně z pěti, pokud se spodní textura nevykresluje), které na sebe navazují. Myšlenka této metody je ve vykreslení vzdálených objektů, jako jsou hvězdy, města, oblaka, krajina a jiné v závislosti na scéně. Tyto objekty nejsou reprezentovány polygony, ale pouze texturami nanášenými na krychli nebo kvádr.

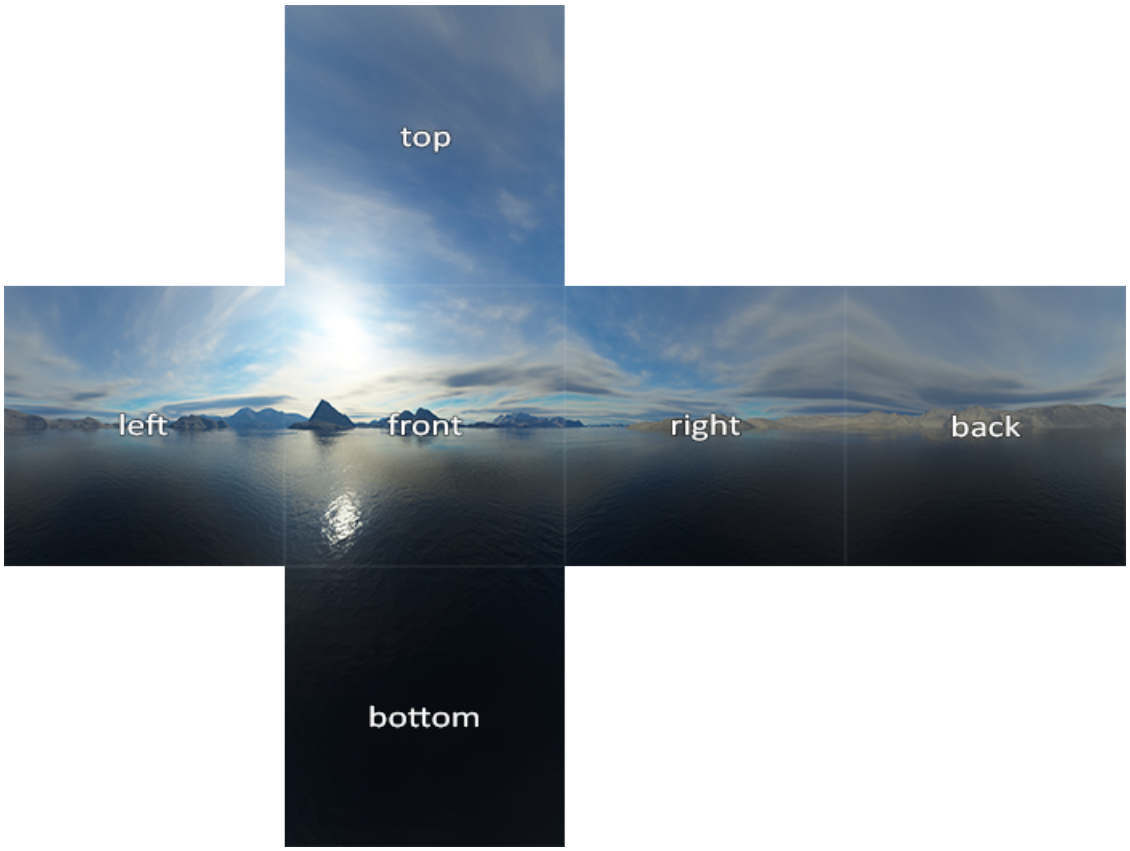
K namapování textury na krychli se využívá technika zvaná cube mapping. Kamera je následně umístěna do středu krychle a při jejím pohybu zůstává poloha krychle vůči kameře stejná. Toho se docílí ve vertex shaderu následovně:

```
gl_Position = projection * view * vec4(position, 1.0);
```

tedy vynecháním matice pro model. Toto ale ještě nezajistí vše. Momentálně se krychle nachází v poloze se středem v bodě *vec3(0,0,0)*, ale nepohybuje se zároveň s kamerou. Toho je možné docílit vynulováním tzv. *"translation components"* v matici *"view"*.

$$view = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.15}$$

Na matici 2.15 je znázorněna možná *"view"* matice, kde x,y a z jsou *"translation components"*. Vynulováním těchto komponent lze docílit toho, že střed krychle bude vždy na poloze kamery.



Obrázek 2.11: Ukázka Skyboxu (převzato z [21])

2.6 Částicové systémy

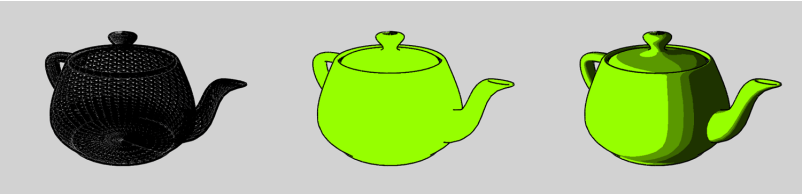
Částicové systémy [19] se v grafice využívají k popisu modelů, které jsou příliš členité nebo se mění takovým způsobem, kterým nelze definovat jejich povrch. Na částice často působí fyzické síly a podle toho se pohybují nebo mění svůj tvar. Využívají se například pro modelování dýmu, ohně, rostoucí vegetace, deště, padání sněhu a jiné.

Částice v počítačové grafice označují hmotné těleso o malých rozměrech. Každá částice může mít fyzikální a grafické vlastnosti, kde mezi fyzikální může patřit například poloha,

hmotnost, rychlost nebo směr pohybu. Mezi grafické vlastnosti pak může patřit například textura, průhlednost nebo barva, pokud se nepoužívá textura. Dále má každá částice životnost, to je doba po jakou je částice aktivní, po uplynutí životnosti zaniká. Pro zobrazování částic se často využívá billboarding popsaný v kapitole 2.4.

2.7 Cel-shading

Cel-shading [15] je metoda pro vizualizaci osvětlení modelu, která přidává nerealistický nádech. Jedná se tedy o druh vykreslování, kdy výsledné objekty vypadají jako ručně malované. Běžné hodnoty osvětlení jsou vypočteny pro každý pixel a následně kvantovány na určitý počet diskrétních odstínu, které pak vytvoří charakteristický plochý vzhled. Cel-shading je hojně využíván ve videohrách. Nejčastěji se uplatňuje v konzolových hrách, například u her Borderlands[1] nebo The Legend of Zelda: The Wind Waker[2].



Obrázek 2.12: Ukázka Cel-shadingu

2.8 Shadow mapping

Shadow mapping [9] je metoda pro generování stínů v počítačové grafice. Tento koncept představil Lance Williams v roce 1978 [22]. Tato technika potřebuje dva vykreslovací průchody scény. Při prvním průchodu se vykresluje z polohy světla a zaznamenává se vzdálenost jednotlivých fragmentů od zdroje světla do hloubkové mapy, která je uložena do textury. Při druhém průchodu se scéna kreslí již z pohledu kamery a pomocí lineární transformace se zkontroluje umístění v hloubkové mapě a z ní se vypočítá vzdálenost. Jestliže je vzdálenost bodu od světla větší jak v hloubkové mapě, tak se jedná o bod, který není vidět a je ve stínu. Pokud je vzdálenost menší, bod je osvětlen.

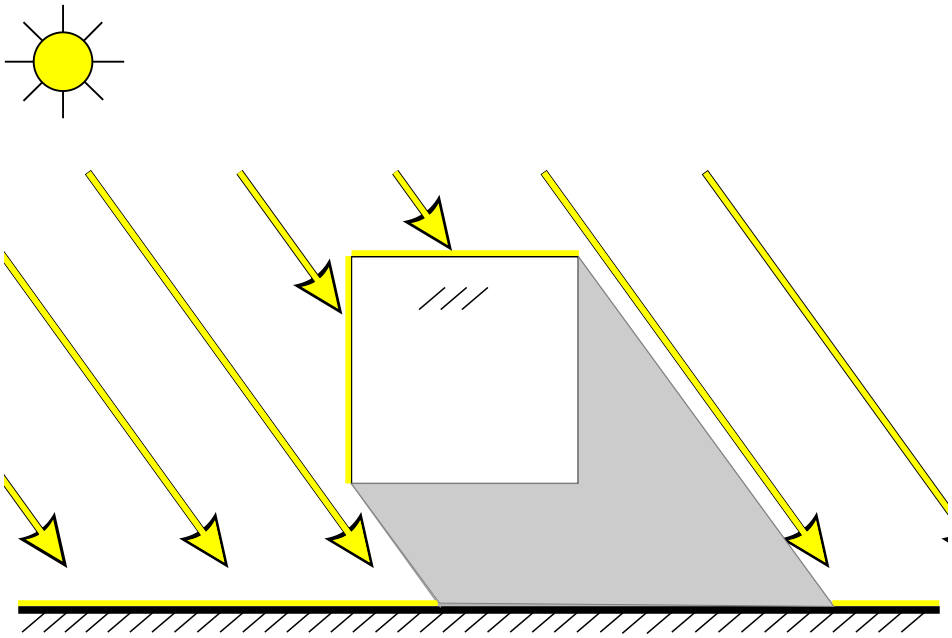
Tato metoda sama o sobě ještě nemusí být dostačující a může docházet k aliasingu, hrany budou příliš ostré. Ostrost stínu zároveň závisí i na velikosti stínové mapy a na vzdálenosti objektu od světla.

Další problém, který nastává, je tzv. Shadow acne. Shadow acne [17] je problém, který vzniká nepřesností hloubkové mapy. Několika sousedícím bodům může odpovídat stejný bod hloubkové mapy. Toto se nejčastěji řeší pomocí odečtení malé hodnoty od vzdálenosti získané z mapy. Hodnota musí být vhodně zvolena, jinak by docházelo ke zkreslení stínů.

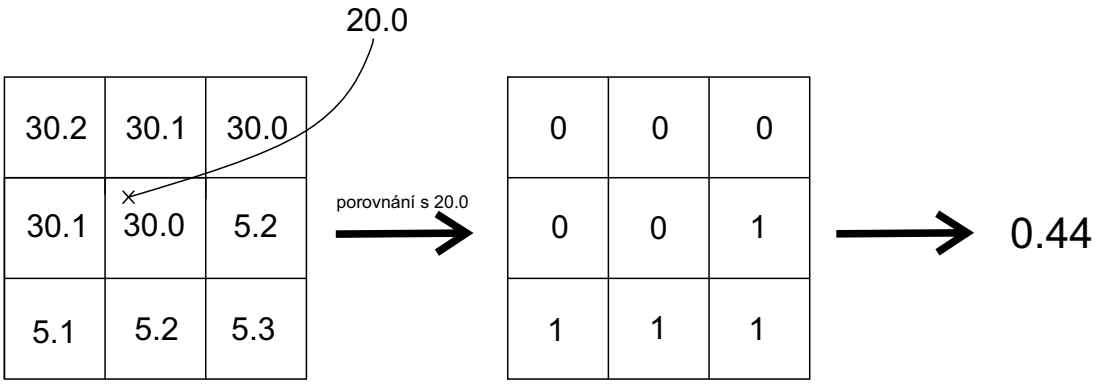
Byly vytvořeny tedy techniky, které se snaží zabránit aliasingu. Jedna z těchto metod je Percentage Closer Filtering (PCF) [9], [12]. Je to vzorkovací metoda, kde se testují okolní fragmenty, zda leží ve stínu nebo nikoliv, poté se pomocí aritmetického průměru vypočítá výsledná hodnota zastínění zpracovávaného fragmentu.

[1] RPG (Role playing game) střílečka (FPS - First-person shooter) vyvinuta firmou Gearbox Software v roce 2009, dostupná z http://borderlandsthegame.com/

[2] Konzolová hra pro Nintendo, vyvinuta v roce 2002, dostupná z http://zelda.com/windwaker/



Obrázek 2.13: Shadow mapping s ortografickou projekcí. Žluté šipky znázorňují vzdálenost uloženou v hloubkové mapě. Šedě je znázorněna zastíněná oblast.



Obrázek 2.14: Princip PCF pro jádro 3x3. Vlevo je znázorněna hloubková mapa s uloženými vzdálenostmi nejbližších bodů. Při testování bodu ve vzdálenosti 20.0 se tato hodnota porovná s blízkými hodnotami uloženými v hloubkové mapě. Výsledkem porovnání je nula nebo jednička. Výsledná hodnota zastínění je poté vypočítána pomocí aritmetického průměru.

Kapitola 3

Implementace Intra

V této kapitole se budu zabývat samotnou implementací grafického intra a způsobu využití metod popsaných v kapitole 2.

3.1 Knihovny

Jak již bylo dříve zmíněno, OpenGL neumožňuje vytvoření okna, ale jen způsob zobrazování. Kvůli tomu se musí sáhnout na nějakou knihovnu, která umožňuje vytvoření okna. Existuje velká spousta knihoven, které toto umožňují. Zprvu byla aplikace vyvíjena pomocí knihovny GLUT, ale nakonec bylo použito WinAPI. Změna byla provedena kvůli nutnosti přikládání externí knihovny, která se ve Windows nenachází.

Nebyla použita žádná knihovna, která by inicializovala OpenGL kontext 3.0 a vyšší, proto bylo potřeba načíst potřebné funkce z hlavičkového souboru *glext.h*. Vytažení funkcí je realizováno pomocí funkce *wglGetProcAddress*, která vrací adresu OpenGL extension funkce. Vytažení funkce například pro *glUniform1i* pak vypadá takto:

```
PFNGLUNIFORM1IPROC glUniform1i;
glUniform1i= (PFNGLUNIFORM1IPROC)wglGetProcAddress("glUniform1i");
```

Pro práci s vektory a maticemi využívám knihovnu glm, která práci s nimi velmi ulehčuje. Přehrávání hudby zajišťuje knihovna libv2 od německé skupiny Farbrausch [1].

3.2 Generování terénu

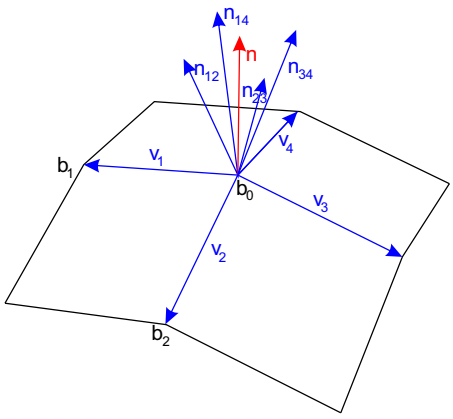
Pro generování terénu je použita výšková mapa, vytvořena za pomoci Perlinova šumu. Nejprve je potřeba vytvořit mřížku bodů, které budou reprezentovat jednotlivé vrcholy primitiv. Jednotlivé body jsou generovány v cyklu, kde souřadnice x a z jsou postupně zvětšovány o konstantní hodnoty a souřadnice y je vypočtena pomocí 2D Perlinova šumu. Šum je nastaven tak, aby generoval kopcovitý terén s prohlubní uprostřed, která bude reprezentovat podvodní terén. Geometrie terénu je tedy generována na CPU.

Zároveň se generují i jednotlivé indexy trojúhelníků (odkazy do různých bufferů), aby poté bylo možné správně vykreslit celý terén pomocí příkazu:

```
glDrawElements();
```

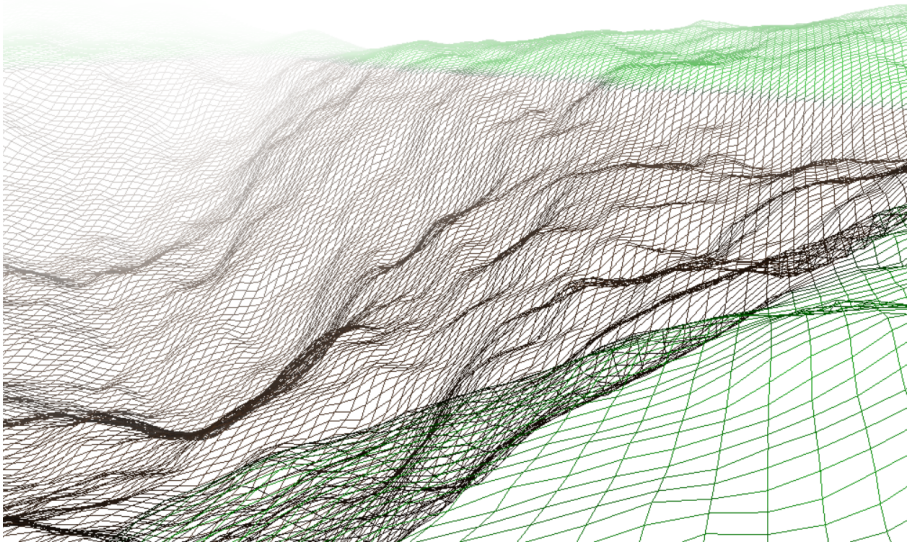
Jednotlivé body jsou pak tedy v bufferu pouze jednou, ale použity mohou být i vícekrát. Dále je potřeba vypočítat normály v každém jednom bodě, kvůli výpočtu Phongova osvětlovacího

modelu a vypočítání odstínu barvy. Při výpočtu normál je nutné se dívat i na okolní body, jak je znázorněno na obrázku 3.1.



Obrázek 3.1: Výpočet normál

Vektor v_1 se vypočítá odečtením bodu b_0 od b_1 a v_2 odečtením b_0 od b_2. Pomocí těchto dvou vektorů a vektorového součinu se vypočítá normála n_{12}, která je posléze normalizována. Výpočet pro normály n_{23}, n_{34} a n_{14} je obdobný. Tyto čtyři normály jsou sečteny a normalizovány, což ve výsledku vytvoří normálu n pro bod b_0. Ilustrace jednotlivých bodů a vektorů lze vidět na obrázku 3.1.



Obrázek 3.2: Výšková mapa vygenerovaná pomocí 2D Perlinova šumu

Na obrázku 3.2 je znázorněn vygenerovaný terén. Lze si zde všimnout, že terén má dvě barvy. Zelenou, která představuje trávu a hnědou, která představuje hlínu pod hladinou,

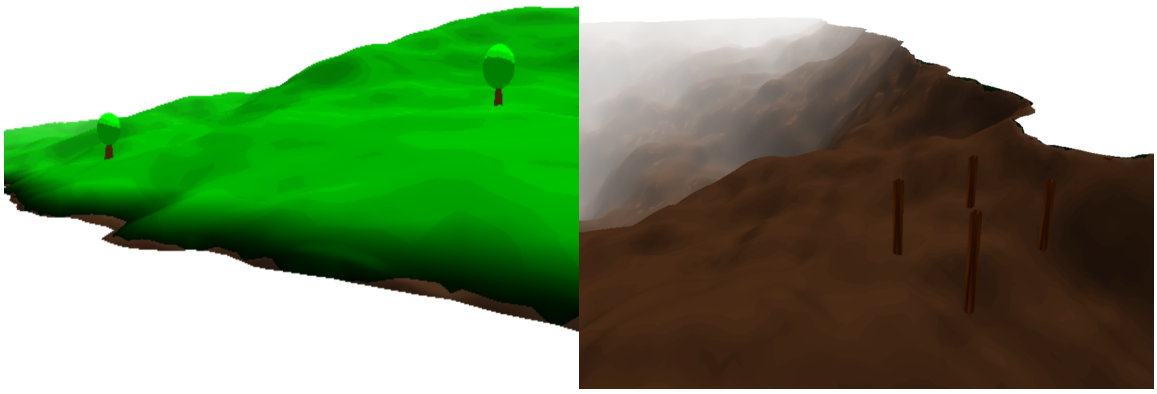
jež je generována těsně nad vodou, čímž je docíleno, že hnědou barvu lze vidět ještě i kousek nad hladinou. Nakonec se pomocí cel-shadingu rozdělí odstíny barev (podle Phongova osvětlovacího modelu) a jsou tak obarveny jednotlivé body.

Pro přidání noční atmosféry je přimíchán modrý odstín a slabé noční osvětlení, které je kolmé na plochu, aby trochu vynikla okolní krajina a nebyla pouze jednobarevná.

3.3 Generování vody

Pro generování vody jsem zvolil jiný přístup než u ostatních objektů a zaměřil se na realističnost. Jedná se o zcela plochý čtverec, který se za pomoci vertex a fragment shaderu změní na vlnící se plochu s odrazem světla.

Jako první je nutné si přichystat dvě textury. Jednu, jež bude znázorňovat vše pod hladinou a bude použita pro refrakci. Druhou, jež znázorní vše nad hladinou a bude použita zase na reflexi. Tyhle dvě textury budou nanášeny na čtverec, ale ještě předtím je deformujeme pomocí tzv. DuDv mapy [14]. DuDv mapa, taktéž nazývána "uv-displacement map", se používá k posunutí souřadnic textury, a tím vytvoří odchylku pro určitý bod. Pro uložení textur potřebujeme dva vykreslovací průchody navíc, pro každou texturu jednou, ale místo vykreslování do okna se výsledek vykreslí do framebufferu.



(a) Textura reflexe znázorňující vše nad hladinou (b) Textura refrakce znázorňující vše pod hladinou

Obrázek 3.3: Ukázky vytvořených textur pro reflexi a refrakci

Jak je na obrázku 3.3 vidět, obsah scény kterou nepotřebujeme, se ořezává. Toho se docílí nastavením ořezávací roviny. Pro reflexní texturu se vykresluje jen vše, co je nad hladinou, tedy to, co je nad nulovou výškou. Naopak pro refrakční texturu jen to, co je pod hladinou.

Ořezání bylo docíleno nastavením:

```
gl_ClipDistance[0];
```

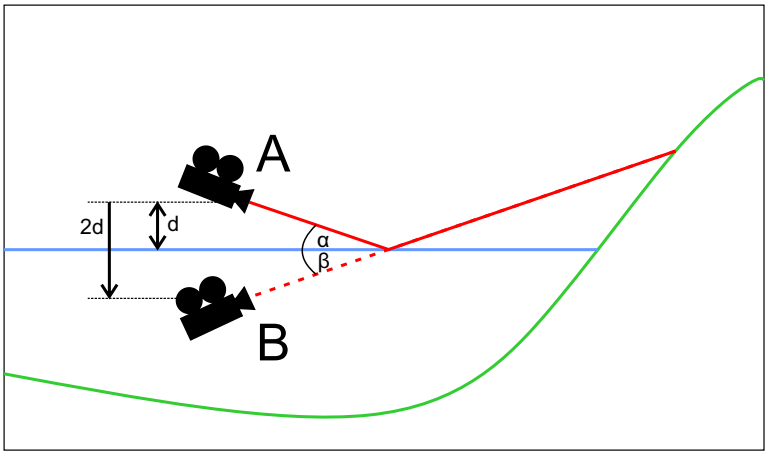
ve vertex shadru. První se musí ale na straně CPU povolit pomocí příkazu:

```
glEnable(GL_CLIP_DISTANCE0);
```

Dále se musí určit, podle jaké roviny se bude scéna ořezávat. Rovina je dána rovnicí:

$$Ax + By + Cy + D = 0, \tag{3.1}$$

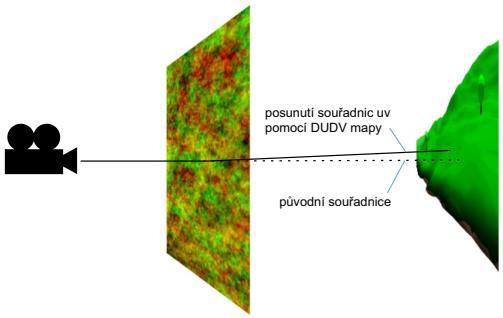
kde A, B a C určují normálu roviny a D určuje výšku, ve které se rovina nachází. V tomto případě budeme využívat normály $(A, B, C) = (0, 1, 0)$ a $(A, B, C) = (0, -1, 0)$, hodnota D bude nulová, protože vodní hladina je vytvořena v nulové výšce. Tímto se docílí, že se vykreslí jen daná půlka scény. Dále je potřeba zajistit, aby se reflexní textura vykreslovala správně. Dosáhne se toho přesunutím kamery na správné místo.



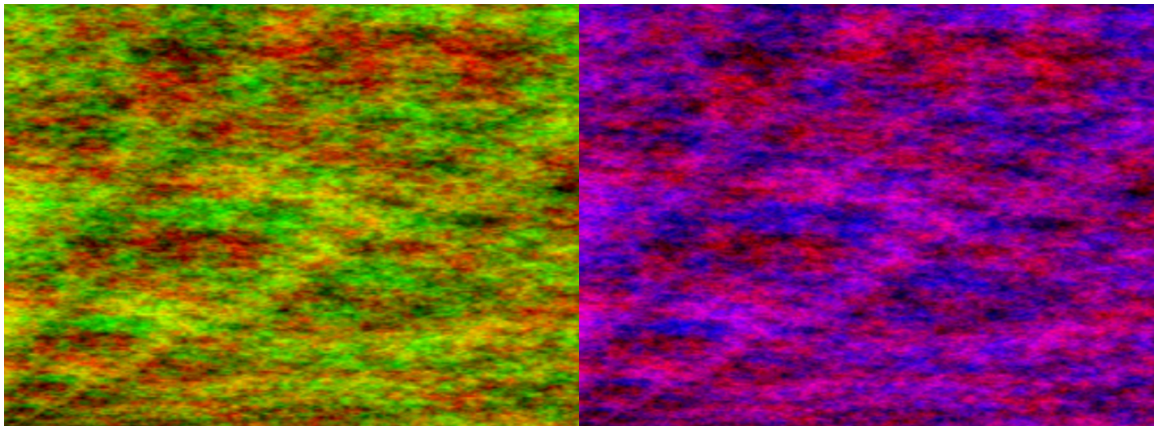
Obrázek 3.4: Přesunutí kamery pro vykreslení do textury pro reflexi

Na obrázku 3.4 je znázorněno, jak se kamera z polohy A přesune do polohy B, odkud se už reflexní textura vykreslí správně. Jako první se vypočítá, v jaké výšce oproti hladině se kamera nachází. Tato proměnná se vynásobí dvěma a kamera se o tuto hodnotu posune směrem dolů. Dále je třeba změnit úhel α na úhel β, k tomu nám stačí vynásobit úhel α mínus jedničkou. Po vykreslení do textury se musí kamera vrátit na původní hodnoty, jinak by s každým vykreslením samovolně měnila polohu. U refrakční textury se kamera přesouvat nemusí, je vykreslována z původní polohy kamery.

Pro efekt vlnící se hladiny je využita tzv. DuDv mapa, pomocí které se zkresluje reflexe na hladině. DuDv mapa je jednoduchá textura, která se skládá ze zelené a červené barvy, kde barvy představují offsety pro souřadnice textury. Princip lze dobře pochopit z následujícího obrázku.

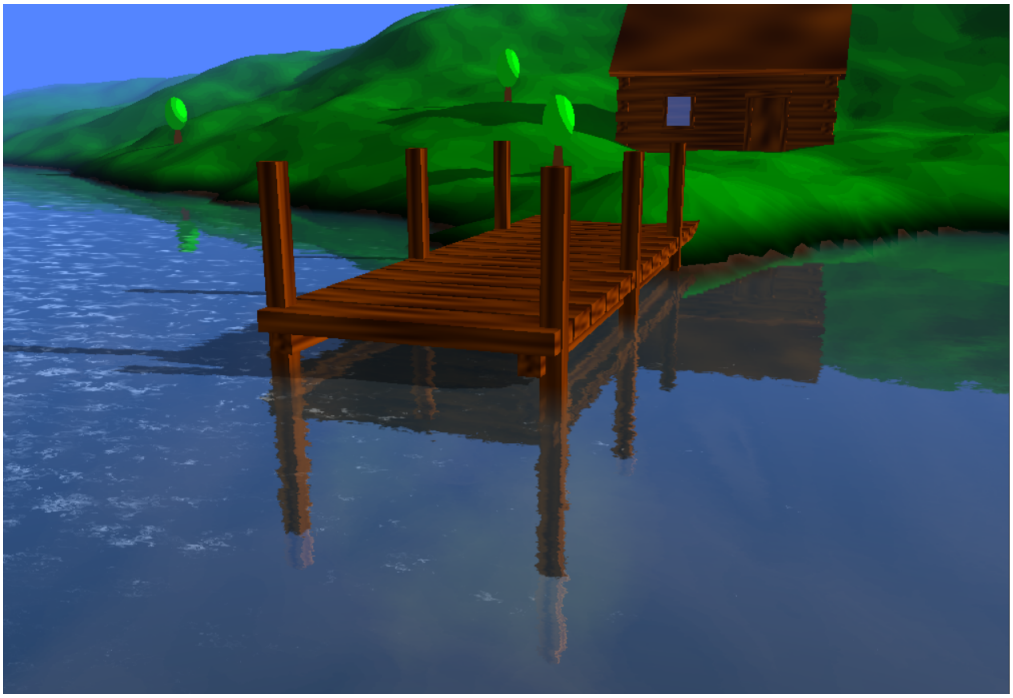


Obrázek 3.5: Znázornění principu DuDv mapy. Barvy v DuDv mapě znázorňují offsety pro souřadnice reflexní textury. Místo bodu na původních souřadnicích se vykreslí bod na posunutých souřadnicích.



(a) DuDv mapa (b) Normálová mapa

Obrázek 3.6: Textury pro vlnění a světelný odraz od hladiny, vytvářeny na základě pokusů a výsledném dojmu ve scéně.



Obrázek 3.7: Kompletní vykreslení vodní hladiny

Aby zkreslení nebylo statické, ale aby se vytvářel dojem vln, je potřeba měnit polohu souřadnic textury. Proto je implementována proměnná *MoveFactor*, která se při každém vykreslení zvětšuje o hodnotu *MoveSpeed*. Výsledná podoba pak v shaderu může vypadat nějak následovně:

```
vec2 distortion = (texture(dudvMap, vec2(textureCoords.x +
moveFactor, textureCoords.y + moveFactor)).rg * 2.0f - 1.0f);
```

Intenzita odraženého a lomeného světla se počítá na základě natočení kamery. Pokud je kamera natočena kolmo k vodě, je vidět plně refrakční textura. Když je kamera natočena

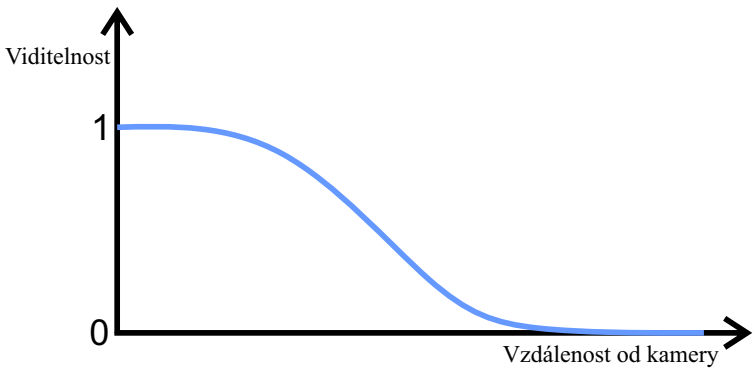
rovnoměrně s vodní hladinou, je vidět pouze reflexní textura. Toho je docíleno pomocí skalárního součinu mezi vektorem ke kameře a normálou vodní hladiny, která je konstantní *vec3(0.0f,1.0f,0.0f)*. Výsledná barva vodní hladiny je pak následující:

```
Color = mix(reflect, refract, refractionFactor);
```

Jako poslední je třeba přidat ještě odraz světla od hladiny. K tomu lze využít normálovou mapu, jedná se o velmi podobnou texturu jako v případě DuDv mapy, ale s tím rozdílem, že obsahuje všechny barevné složky. Intenzita jednotlivých barevných složek v normálové mapě určuje výslednou normálu.

3.4 Viditelnost

Viditelnost [8] objektů v dálce je počítána na základě jejich vzdálenosti od kamery. Výsledná barva objektu v dálce je ovlivněna barvou oblohy, a to vytváří dojem mizící krajiny. Objekty ve velké vzdálenosti od kamery jsou vykreslovány stejnou barvou jakou má obloha, takže jsou kompletně skryty před zrakem. Objekty v blízkosti jsou vykreslovány jejich obvyklou barvou. Všechny objekty mezi nimi jsou vykreslovány smícháním barev oblohy a obvyklou barvou v závislosti na vzdálenosti od kamery.



Obrázek 3.8: Graf závislosti vzdálenosti a viditelnosti

Na obrázku 3.8 je vidět, že závislost mezi vzdáleností a viditelností je exponenciální, což dává více realistický výsledek než lineární závislost. Vztah mezi vzdáleností a viditelností je dán následovně:

$$Visibility = e^{-(distance*density)^{gradient}}, \tag{3.2}$$

kde proměnná *distance* je vzdálenost bodu od kamery, *density* určuje hustotu mlhy. Čím větší je její hodnota, tím se zmenšuje viditelná plocha scény. Naopak čím menší, tím se zobrazují vzdálenější objekty. Hodnota *gradient* určuje, jak rychlý bude přechod mezi plnou viditelností a nulovou viditelností.

3.5 Skybox

Scéna je obklopena skyboxem, který se mění v závislosti na poloze slunce. Ve dne se jedná pouze o modrou plochu, která se od určitého umístění slunce začíná měnit v noční oblohu.

(a) Hvězdná obloha s násobením konstantou 5.0

(b) Hvězdná obloha s násobením konstantou 15.0

Obrázek 3.9: Ukázky vygenerovaných hvězd

Noční oblohu tvoří tmavě modrá barva a hvězdy. Hvězdy jsou vytvářeny pomocí 3D Perlinova šumu, který je posléze prahován. Výsledek je tedy tvořen jen nejvyššími hodnotami šumu, amplitudou.

Perlinův šum je volán následovně:

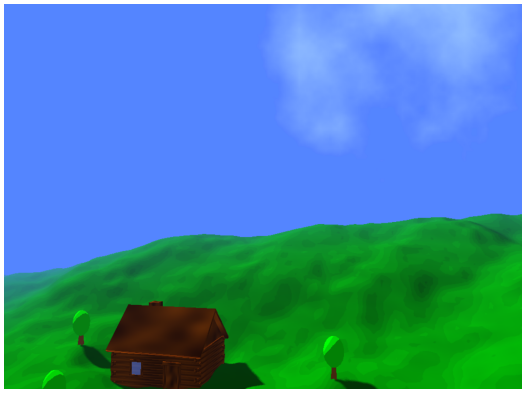
```
noise = snoise(TexCoords * 15.0)*0.5 + 0.5;
```

kde *TexCoords* je vektor znázorňující bod v prostoru, který je vynásobený konstantou. Na obrázku 3.9 lze vidět vliv konstanty na celkovou podobu hvězdné oblohy. Jelikož ale Perlinův šum generuje hodnoty v intervalu $\langle -1, 1 \rangle$ a funkce je využívána pro výpočet výsledné barvy, je třeba výsledek posunout do intervalu $\langle 0, 1 \rangle$.

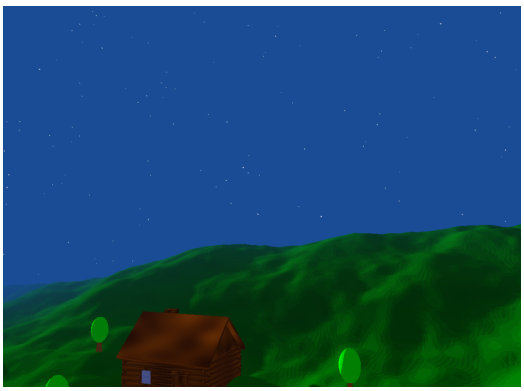
Pokud je slunce těsně nad obzorem, začne se obloha měnit do noční podoby a toho je docíleno následovně:

```
color = mix(color, fColor, light.y / 400.0f);
```

kde *color* je barva hvězd a *fColor* je barva pozadí.



(a) Denní obloha

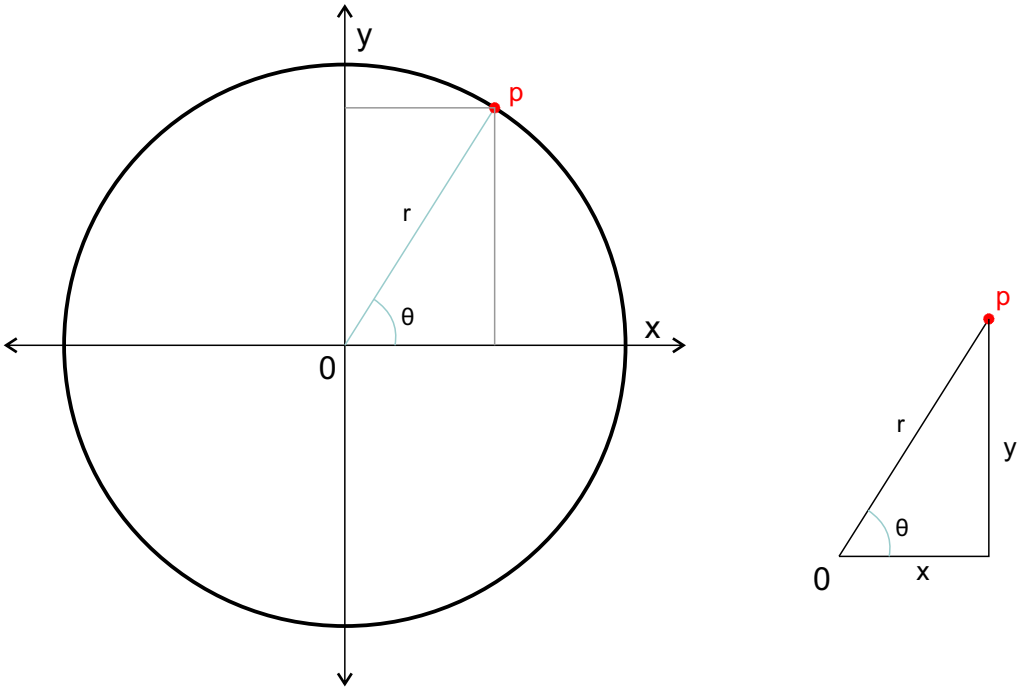


(b) Obloha během přechodu a objevující se hvězdy

Obrázek 3.10: Ukázky skyboxu během dne a stmívání. Noční oblohu lze vidět na obrázku 3.9b.

3.6 Slunce

V denní části intra se objevuje slunce, které je modelováno jako koule a nachází se na poloze zdroje světla. Poloha světla se mění při každém vykreslení. Slunce se pohybuje po kružnici. Na obrázku 3.11 vlevo je dobře vidět trojúhelník znázorněný na stejném obrázku vpravo,



Obrázek 3.11: Výpočet polohy slunce. Bod p je poloha slunce, r vzdálenost od počátku souřadnic a úhel θ je vypočtený pomocí funkcí cos a sin v závislosti na čase.

pomocí kterého se dá jednoduše zjistit vztah pro výpočet polohy slunce. Úhel θ se mění v závislosti na čase a poloměr r určuje vzdálenost slunce od středu souřadnicového systému.

Výsledný vztah, pomocí kterého se slunce posouvá, je následující:

$$Light.position.x = \sin(time) * r \tag{3.3}$$

$$Light.position.y = \cos(time) * r \tag{3.4}$$

Jelikož se jedná o pohyb po kružnici, stačí pouze dvě souřadnice. Souřadnice z je konstantně nastavena na hodnotu 200. Poté stačí jen pomocí funkce přesunout objekt na vypočítanou polohu.

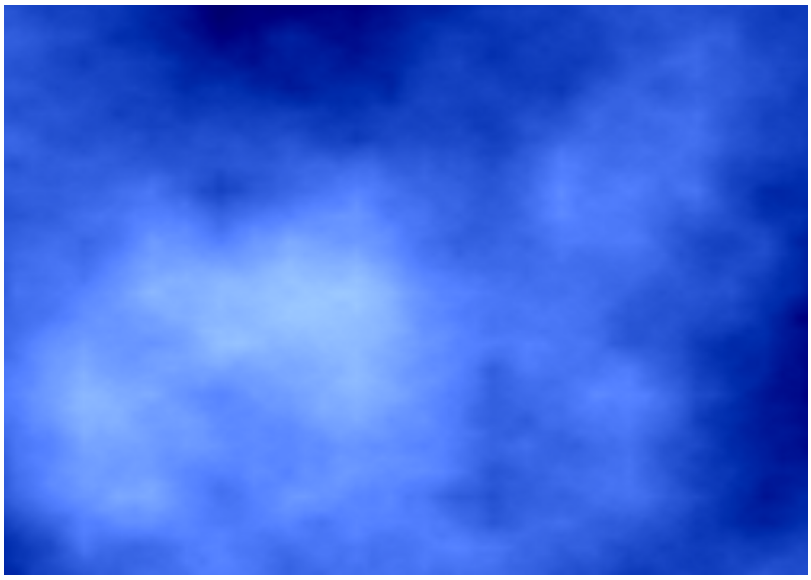
```
model = glm::translate(model, Light.position);
```

3.7 Oblaka

Jelikož je obloha ve dne jednobarevná, rozhodl jsem se přidat oblaka aspoň v podobě pohybující se textury. Textura je vytvořena pomocí billboardingu popsaného v kapitole 2.4, tedy je neustále natočena směrem ke kameře.

Pro vytvoření textury oblaku jsem si implementoval pomocný program, ve kterém se výsledný šum mění pomocí kláves na klávesnici, a na výstup jsem si vypisoval jednotlivé proměnné. Mezi proměnné, které se měnily, patří například barvy, frekvence, amplituda, počet oktáv a změna souřadnic x a y.

Výsledná textura je vidět na obrázku 3.12. Pokud bychom ale tuhle texturu rovnou použili, byl by to přinejmenším divně poletující čtverec na obloze. Proto ve fragment shaderu využívám funkci *discard*. Jedná se o funkci, která zahodí fragment a ten se nevykreslí. Výsledný oblak na obloze lze vidět na obrázku 3.13.



Obrázek 3.12: Výsledná vygenerovaná textura pro oblak a kouř

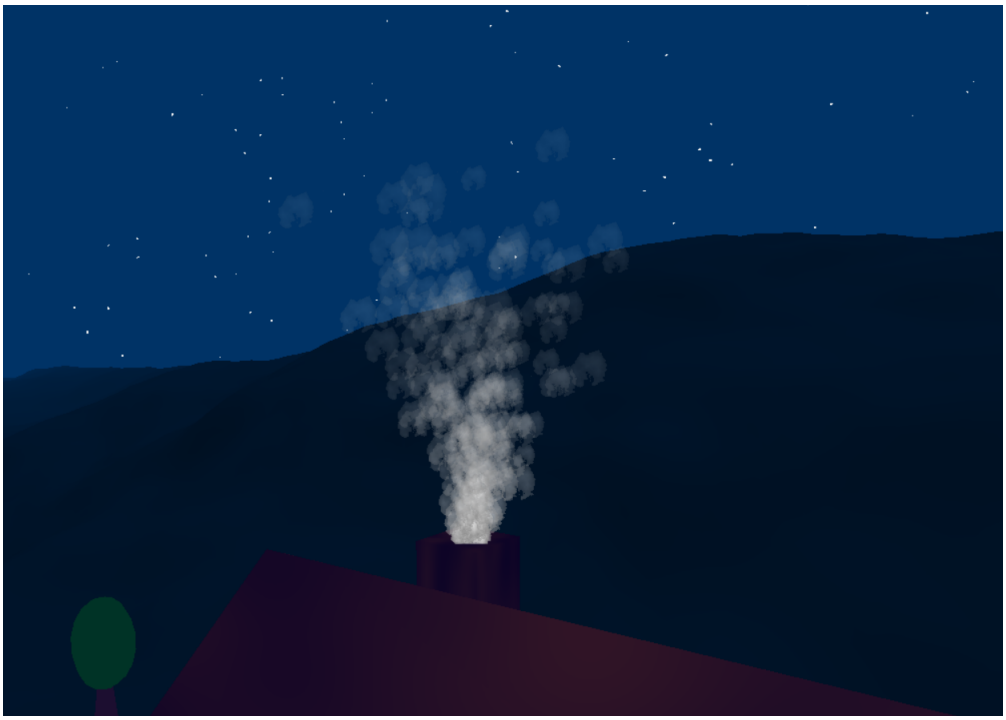
Obrázek 3.13: Výsledná vygenerovaná textura na obloze po provedení funkce *dicard*

3.8 Dým z komínu

Dým z komínu se v intru objevuje v noci. Jedná se o částicový systém. Částice jsou generovány uvnitř komínu a pomocí billboardingu je vytvářen 3D dojem.

Každá částice má svoji životnost, která se při každém vykreslení zmenšuje. Pokud životnost klesne pod nulovou hranici, částice zaniká. Při vytvoření částice se jí vygeneruje náhodný směr, rychlost a velikost. Vygenerovaným směrem se bude pohybovat až do doby, kdy jí vyprší životnost.

Pro větší realističnost bylo přidáno mizení dýmu na základě výšky, kterého si lze všimnout i na obrázku 3.14. Textura použita pro kouř je totožná s tou, která je využita pro zobrazení oblak (obrázek 3.13).



Obrázek 3.14: Ukázka dýmu vytvořeného pomocí částicového systému

3.9 Tvorba chaloupky a mola

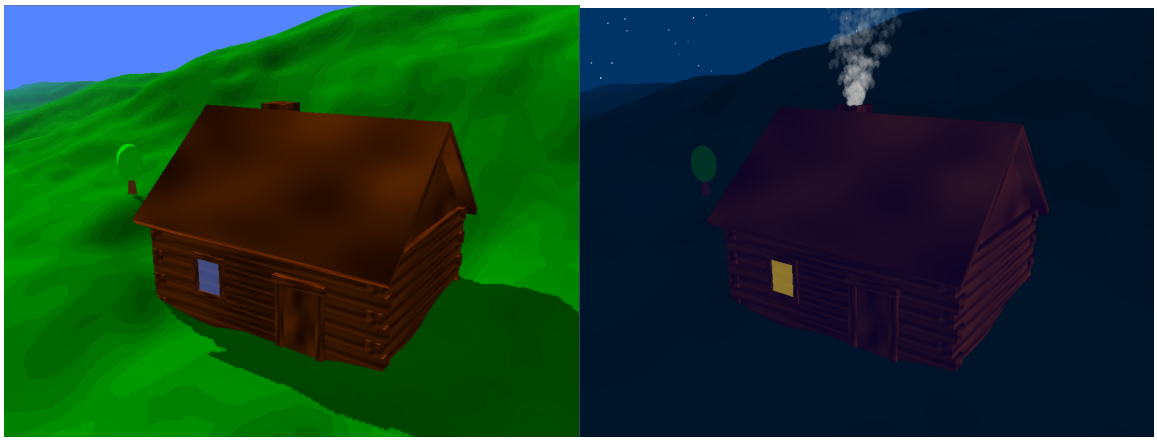
Tvorba chaloupky a mola je velmi podobná. Jedná se o skládání jednoduchých krychlí, které jsou transformovány na různé rozměry pomocí funkce:

```
glm::scale(glm::mat4 model, glm::vec3 scale);
```

Testoval jsem zde, zda je lepší jednotlivé vektory pro polohu a škálování uložit napevno do pole nebo jednotlivé polohy určit v cyklu na základě výpočtu. Ukázalo se, že množství bodů není tak velké, aby mělo význam dělat jejich umístění pomocí algoritmu, ba dokonce výsledná velikost byla ještě větší než v případě uložení vektorů do pole.

Jednotlivou polohu kostiček jsem určoval pomocí pomocné kostičky, se kterou jsem pohyboval v prostoru a měnil její tvar. Po nalezení správné pozice a velikosti jsem si vypsal její souřadnice a rozměry.

Chaloupka na rozdíl od mola obsahuje i okno. První myšlenka, kterou jsem chtěl realizovat, bylo na okně míchat barvu s reflexí, jenže k tomu by bylo třeba vykreslit danou scénu ještě jednou navíc. Při pokusu s reflexí docházelo již k velkým skokům ve výkonu, tak jsem se nakonec rozhodl okno modelovat jen jako průhledný čtverec, který má rozdílnou barvu ve dne a v noci.



(a) Chaloupka ve dne (b) Chaloupka v noci

Obrázek 3.15: Ukázka chaloupky ve dne a v noci, molo lze vidět na obrázku 3.7

3.10 Hudba

Jelikož se jedná o grafické intro, nedílnou součástí je i hudba. Vzhledem k omezené velikosti celé aplikace nelze do intra vložit hudbu ve formátu mp3 nebo podobném. Hudba by zabrala více kapacity než je limitující velikost. Použitelný by mohl být například formát MIDI, který si ukládá pouze seznam jednotlivých not, ale nedosahuje příliš velkých kvalit. Pro přehrávání hudby jsem si zvolil knihovnu libv2 od německé skupiny Farbrausch [2], která je vytvořena speciálně pro grafická intra s omezenou velikostí, kvalitou podobnou formátům jako je například mp3. Ke knihovně byl dodáván i ukázkový překladač napsaný v jazyce C. Díky knihovně je pak možné soubor s hudbou přímo vložit do spustitelného souboru a následně i přehrávat.

Jelikož je část kódu v assembleru, je nutné při překladu přeložit i assemblerský kód. Proto bylo třeba do Visual Studia přidat překladač pro assembler. Vybral jsem si překladač YASM [7], ve kterém byla napsána i ukázka přibalená ke knihovně libv2.

Jako hudbu jsem si vybral skladbu Patient Zero od Melwyn and Little Bitchard[1].

Na jednom z testovaných počítačích docházelo k padání aplikace z důvody hudby, proto byla přidána možnost spustit aplikaci bez zvukového doprovodu. Toto spuštění je možné provést z příkazového řádku zadáním parametru *--no-music*.

[1] Tuto hudbu a spoustu dalších lze stáhnout na stránkách:
http://www.mmnt.net/db/0/0/ftp.untergrund.net/users/Freefall/V2/Modules/Others

3.11 Pohyb kamery ve scéně

Aby se jednalo o dynamické intro musel být implementován taky i pohyb kamery. Jako první jsem si implementoval pohyb kamery pomocí kláves, obdobných jak ve spoustě počítačových her W, S, A a D a natočení kamery pomocí stisku a držení tlačítka myši.

Otáčení kamery funguje na tom principu, že se vezme poslední poloha kurzoru a od ní se pak počítá rozdíl v souřadnicích. Při stisknutí se jako poslední kurzor nastavuje místo stisknutí. Výpočet pro natočení kamery je následující:

```
front.x = cos(glm::radians(pitchh)) * cos(glm::radians(yaww));
front.y = sin(glm::radians(pitchh));
front.z = cos(glm::radians(pitchh)) * sin(glm::radians(yaww));
camera.Front = glm::normalize(front);
```

Proměnná *pitchh* a *yaww* jsou globální proměnné pro natočení v ose x a y, které se vypočítají na základě této rovnice:

$$yaww+ = (xPos - lastX)sensitivity, \tag{3.5}$$

kde $xPos$ je aktuální poloha a $lastX$ je poloha kurzoru v předchozím průchodu vykreslování, $sensitivity$ je pak citlivost otáčení. Obdobná rovnice je pro natočení v ose y, jen se prohodí proměnné za ty, které pracují s osou y.

Pro pohyb kamery ve scéně se ukládá pozice kamery do proměnné a při stisku klávesy se provede daná operace pro posun. Pro posun vpřed:

```
Position += Front * MovementSpeed;
```

kde vektor $Front$ udává směr natočení kamery a $MovementSpeed$ rychlost posunu.

Aby se v intru nemuselo pohybovat manuálně, musel jsem implementovat automatický pohyb kamery ve scéně. Pro toto jsem si musel zvolit způsob, jakým se bude kamera automaticky posouvat v čase. Nakonec jsem si zvolil pohyb po interpolační křivce Catmull-Rom popsané v [16].

Jelikož má intro omezenou velikost, nemůžu si ukládat polohu kamery pro každý krok času. Místo toho si uložím klíčové body a mezi nimi budu interpolovat. Klíčový bod je vektor, který se skládá z polohy kamery a natočení, jedná se tedy o pětici. Pro interpolaci mezi body v_1 a v_2 se potřebují i body v_0 a v_3, tedy předchozí a následující body. Výpočet je dán pak následovně:

$$q(t) = 0.5(1.0, t, t^2, t^3) \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{3.6}$$

Hodnoty v_0, v_1, v_2 a v_3 jsou klíčové body, hodnota $q(t)$ je interpolovaná hodnota mezi body v_1 a v_2. Parametr t může nabývat hodnot v rozsahu $t \in \langle 0, 1 \rangle$. Když $t = 0$, platí, že $q(t) = v_1$ a pokud $t = 1$, tak $q(t) = v_2$.

Kapitola 4

Metody pro omezení výsledné velikosti aplikace

Jak již bylo zmíněno v grafickém intru s omezenou velikostí je kladen důraz na výslednou velikost spustitelného souboru. Pro zmenšení výsledné aplikace existuje spousta způsobů, jak jej minimalizovat. Velkou roli hraje nastavení překladače. Další možností, jak zmenšit výslednou aplikaci je zkomprimovat pomocí speciálních "exe packerů". Další způsoby jsou ve stylu psaní kódu. Velmi vhodné je psát co nejobecněji, aby daná konstrukce šla využít na více místech a nemusel se daný kód opakovat.

4.1 Nastavení překladače

Pro práci jsem si vybral prostředí Visual Studio 2015, které nabízí velkou škálu možných nastavení. Největší roli mají parametry nastavení překladače a linkeru, pomocí kterých se přepínají různé optimalizace. Vzhledem k omezení výsledné velikosti souboru jsem se zaměřil na jejich optimalizaci.

Jedním z nejdůležitějších parametrů pro nastavení překladače optimalizující velikost je /O1, který vytváří malý kód. Další parametr, který ma vliv na výslednou velikost je odstranění parametru /GS kontrolující přetečení bufferů. Existuje spousta dalších parametrů, které zmenšují výslednou velikost, ale redukce velikosti již není tak značná.

Další optimalizace velikosti je možná přes nastavení linkeru. Velikost ovlivní například parametr /INCREMENTAL, který určuje zda je zapnuto nebo vypnuto inkrementální linkování. Další parametr, který ovlivní výslednou velikost je vypnutí informací pro debugger, tedy odstranění parametru /DEBUG. Jiné možné parametry a nastavení lze najít na stránce [5].

Kromě nastavení překladače a linkeru, bylo také potřeba nastavit překladač pro assembler, jak již bylo zmíněno v kapitole 3.10. Aby se spustitelný soubor překladače nemusel vkládat do složky, kde jsou obsaženy původní programy pro Visual Studio, bylo třeba změnit soubor *vsyasm.props* a v něm položku *YASMPATH* na relativní umístění pro *vsyasm.exe*.

4.2 exe packery

I přes veškeré nastavení překladače a metody pro omezení výsledné velikosti aplikace, je aplikace stále větší než 64kB, což nesplňuje základní podmínku. Proto se výsledný soubor musí zmenšit pomocí tzv. exe packerů.

Jsou to programy vytvořené za účelem minimalizovat výslednou aplikaci. Existuje celá řada těchto komprimačních souborů, některé jsou vytvořeny speciálně pro grafická intra do 64kB. Exe packery vytvoří ze spustitelného souboru program nový, který obsahuje komprimovaná data a dekomprimační kód, který z komprimovaných dat vytvoří originální program a ten pak spustí.

Pro svou práci jsem si vybral exe packer UPX [6]. Jedná se o konzolový program, který má spoustu nastavitelných parametrů. Pomocí exe packeru se výsledná aplikace zmenšila na třetinu své původní velikosti, což bylo dostačující pro dosažení podmínky. UPX bylo spuštěno s těmito parametry:

```
upx -9 --ultra-brute -o graphicintro.exe Intro.exe
```

Parametr *-9* udává nejlepší kompresi ohledně velikosti a parametr *–ultra-brute* ještě vylepšuje komprimaci za cenu rychlosti.

	Před komprimací	**Po komprimaci UPX**	
		-9	**-9 –ultra-brute**
bez hudby	99 kB	43 kB	41 kB
s hudbou	167 kB	53 kB	49 kB

Tabulka 4.1: Porovnání výsledků

Kapitola 5

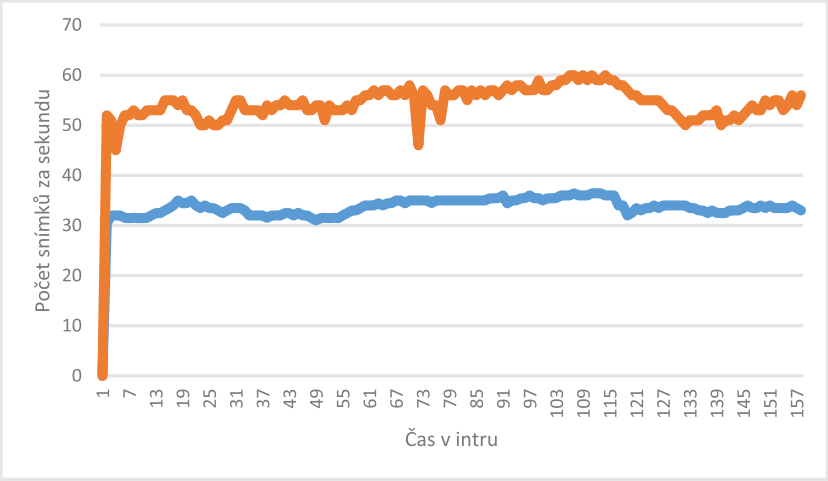
Výsledek

V této kapitole budou uvedeny výsledky práce, jako je rychlost nebo ukázky z intra.

5.1 Rychlost vykreslování

Pro výpočet snímků za sekundu využívám funkci *GetTickCount()*. Pomocí níž zaznamenávám aktuální a poslední čas, při kterém byla překročena sekunda. Pokud je rozdíl mezi aktuálním a posledním časem větší jak jedna sekunda, vypočítá se počet snímků pomocí proměnné *Frames*, která je inkrementovaná při každém vykreslení scény. Po výpočtu se tato proměnná vynuluje pro další výpočet a poslední čas se aktualizuje aktuálním časem.

Hlavní zátěž na rychlost vykreslování je zde voda, která potřebuje 2 průchody vykreslování navíc. Jako experiment jsem zkusil, jak velký rozdíl ve snímcích za sekundu bude, pokud se nebude vykreslovat voda. Na následujícím grafu lze vidět průběh fps[1] v intru.



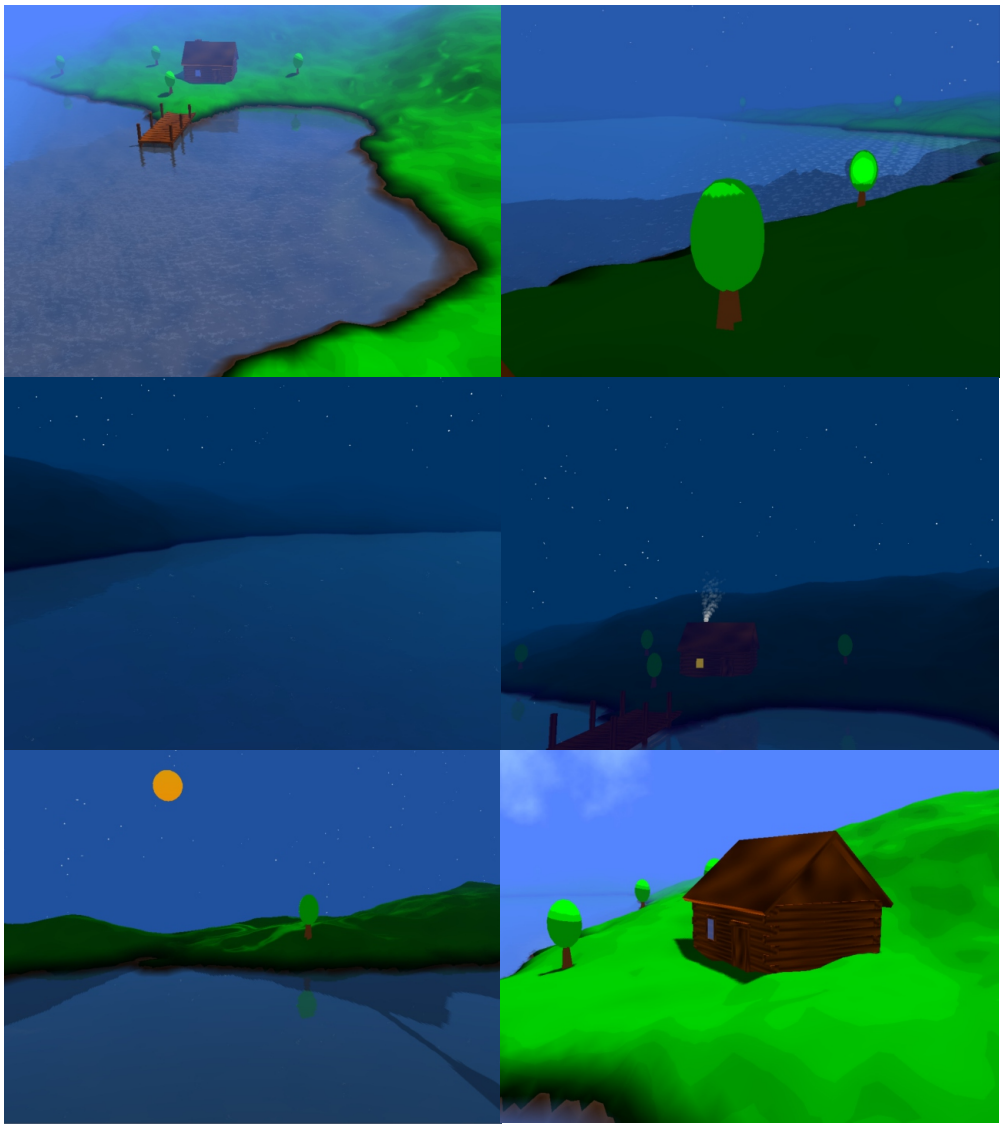
Obrázek 5.1: Počet snímků za sekundu. Modře je znázorněno fps s vodou, oranžově bez vody.

Rychlost vykreslování je vyšší jak 30 snímků za sekundu, takže intro je plynule bez viditelných záseků i přesto, že se celá scéna vykresluje čtyřikrát.

[1] fps - frames per second (počet snímků za sekundu)

5.2 Výsledná scéna

Finální scéna se skládá ze 4 kroků. Jako první se vykreslí stíny stromů, chaloupky a mola do textury, dalším krokem je vykreslení scény do reflexní a refrakční textury. Následně se již vykreslí finální scéna s nanesenými stíny a texturami na terénu a vodě.



Obrázek 5.2: Snímky z intra

Intro končí pohledem na chaloupku a čeká se na stisk klávesy "ESC" nebo na doznění hudby. Během této doby je čas zrychlený, takže celý den proběhne v rámci pár sekund.

Kapitola 6

Závěr

Cílem této práce bylo vytvořit grafické intro s velikostí, která nepřesáhne 64kB. Výsledné intro má velikost 49kB, takže s rezervou 15kB splnilo podmínku. Pro účely demonstrace technik využívaných pro grafická intra s omezenou velikostí jsem si vybral krajinku, ztvárněnou spíše v naivním malířském stylu než v reálné skutečnosti, která simuluje průběh dne. Před začátkem práce jsem měl velmi malé vědomosti ohledně tvorby grafických aplikací, a proto jsem bral projekt jako možnost naučit se a vyzkoušet si další oblast v daném studijním oboru, která mě vždy zajímala.

Při tvorbě jsem se seznámil s možnostmi OpenGL a procedurálním generování objektů. V práci jsem využil metody jako:

- Procedurální generování (Perlinův šum)
- Phongův osvětlovací model
- Billboarding
- Skybox
- Částicové systémy
- Cel-shading
- Shadow mapping

Během práce na intru mě napadala spousta možných rozšíření. Mezi nejzajímavější by mohla patřit simulace ročních období, přidání proměnlivého počasí, více vegetace, zvířectvo nebo i rybář chytající ryby. Práce byla pro mě novou, velmi zajímavou zkušeností s velkým přínosem, ale i výzvou do dalšího studia, ve kterém budu rád pokračovat.

Literatura

[1] Farbrausch. [online]. [cit. 2016-04-14]. Dostupné z: http://www.farbrausch.de/.

[2] Farbrausch: V2 synthesizer syste. [online]. [cit. 2016-04-25]. Dostupné z: http://1337haxorz.de/products.html.

[3] OpenGL 4 Pipeline. [online]. [cit. 2016-04-06]. Dostupné z: http://www.lighthouse3d.com/2011/03/opengl-4-1-pipeline/.

[4] opengl-tutorial - Billboards. [online]. [cit. 2016-04-07]. Dostupné z: http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/billboards/.

[5] Reducing Executable Size. [online]. [cit. 2016-04-27]. Dostupné z: http://www.catch22.net/tuts/reducing-executable-size.

[6] UPX : Ultimate Packer for eXecutables. [online]. [cit. 2016-04-27]. Dostupné z: http://upx.sourceforge.net/.

[7] YASM: The Yasm Modular Assembler. [online]. [cit. 2016-04-25]. Dostupné z: http://yasm.tortall.net/.

[8] BUBNAR, M.: Fog Outside. [online]. [cit. 2016-05-10]. Dostupné z: http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=15.

[9] EISEMANN, E.; SCHWARZ, M.; ASSARSSON, U.; aj.: *Real-Time Shadows*. A. K. Peters, Ltd. Natick, MA, USA, 2011, ISBN 1568814380 9781568814384.

[10] ELIAS, H.: Perlin Noise. [online]. [cit. 2016-04-14]. Dostupné z: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[11] ENTACHNER, K.: Linear Congruential Generator: LCG. [online]. [cit. 2016-05-09]. Dostupné z: http://random.mat.sbg.ac.at/results/karl/server/node3.html.

[12] FERNANDO, R.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004, ISBN 0321228324.

[13] FERNANDO, R.; KILGARD, M. J.: *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003, ISBN 0321194969.

[14] HUMPHREY, B. D.: Realistic Water Using Bump Mapping and Refraction. [online]. [cit. 2016-05-04]. Dostupné z: https://hydrogen2014imac.files.wordpress.com/2013/02/realisticwater.pdf.

[15] HUTCHINS, A.; KIM, S.: Advanced Real-Time Cel Shading Techniques in OpenGL. [online]. [cit. 2016-05-10]. Dostupné z: http://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S12/final_projects/hutchins_kim.pdf.

[16] JOY, K. I.: CATMULL-ROM SPLINES. 2002.

[17] PHARR, M.; FERNANDO, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005, ISBN 0321335597.

[18] PHONG, B. T.: *Illumination of Computer-Generated Images*. Dizertační práce, University of Utah, UTEC-CSs-73-129, July 1973.

[19] REEVES, W. T.: Particle systems—a technique for modeling a class of fuzzy objects. *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, ročník 17, č. 3, 1983: s. 359–375.

[20] VERGNE, R.: Image synthesis and OpenGL: lighting. [online]. [cit. 2016-05-09]. Dostupné z: http://romain.vergne.free.fr/teaching/IS/SI04-lighting.html.

[21] de VRIES, J.: Learn OpenGL - Cubemaps. [online]. [cit. 2016-05-11]. Dostupné z: http://learnopengl.com/#!Advanced-OpenGL/Cubemaps.

[22] WILLIAMS, L.: Casting curved shadows on curved surfaces. *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ročník 12, č. 3, 1978: s. 270–274.