Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Modelování a vykreslování objemových mraků pro herní aplikace

Plzeň, 2010 Vladimír Geršl

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky
Akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Vladimír GERŠL**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačová grafika a výpočetní systémy**

Název tématu:	**Modelování a vykreslování objemových mraků pro herní aplikace**

Zásady pro vypracování:

1. Seznamte se s existujícími řešeními daného problému a shrňte jejich výhody a nevýhody.
2. Na základě tohoto souhrnu a dalších požadavků ze strany zadavatele navrhněte vlastní řešení.
3. Po schválení zadavatele řešení implementujte a vyzkoušejte na požadovaném typu dat.
4. Dosažené výsledky zhodnoťte.
5. K vytvořenému programovému vybavení sepište uživatelskou a programátorskou dokumentaci.

Rozsah grafických prací:	**dle potřeby**
Rozsah pracovní zprávy:	**min. 40 stran původního textu**
Forma zpracování diplomové práce:	**tištěná**

Seznam odborné literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Ing. Petr Vaněček, Ph.D.** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**29. srpna 2008**
Termín odevzdání diplomové práce:	**21. května 2009**

Prof. Ing. Jiří Křen, CSc.
děkan

Prof. Ing. Jiří Šafařík, CSc.
vedoucí katedry

V Plzni dne 11. září 2008

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

Vladimír Geršl

V Plzni dne 17. 7. 2010

Poděkování

Je autorovou milou povinností poděkovat všem, kteří svým dílem přispěli ke vzniku této diplomové práce. Na prvním místě by rád poděkoval vedoucímu Ing. Petru Vaněčkovi, PhD. a externímu konzultantovi Ing. Michalovi Varnuškovi, PhD., bez jejichž trpělivosti a osobního přístupu by nevznikla. Další díky patří Ing. Václavu Purchartovi za odbornou pomoc a konstruktivní konzultace, při nichž vznikla řada důležitých nápadů. V neposlední řadě se sluší též poděkovat autorově přítelkyni a rodině za to, že vytvářeli motivující zázemí a poskytovali maximální podporu v průběhu celého studia. Mimoto také autor děkuje svému současnému zaměstnavateli, firmě Cauldron, za poskytnutý časový prostor pro dokončení této práce.

Abstrakt v češtině

Tato práce představuje ucelený, funkční a v několika směrech nový postup, jak vykreslovat objemové mraky. To znamená mraky, kterými lze prolétávat. Nesmíme však zapomenout na její hlavní poslání – jde o algoritmus určený pro využití v počítačových hrách. Neobsahuje tedy žádné simulace fyzikálních procesů, dynamických změn tvaru a ani se nesnaží vykreslovat všechny známé typy mraku. Její alfou a omegou je rychlost vykreslování (s využitím GPU) a snaha – typická pro herní průmysl – o realističnost pouze do potřebné míry.

Jak zní tedy přesné zadání, které tato práce popisuje? Vykreslení co nejvěrněji vypadajícího objemového mraku s osvětlením v reálném čase s minimálními nároky na výpočetní výkon.

Základem algoritmu je vysílání paprsků 3D texturou. Jejich výsledek se poté dvouprůchodově zobrazí na imposteru, neboli billboardu, jehož obsah je aktualizován pouze za určitých předpokladů. Těmito mraky je poté zaplněna obloha a je na ně aplikován vítr a osvětlení v reálném čase. Součástí tohoto dokumentu je tedy mimo vysvětlení esenciálního algoritmu též popis doplňujících technik (osvětlení, vítr, řízení úrovně detailů aj.) a popsání postupů pro tvorbu doprovodného obsahu (např. modelu letadla, textur).

Klíčová slova: cumulus, objemový mrak, imposter, billboard, textura, GPU, vertex shader, pixel shader, úroveň detailů, osvětlení v reálném čase

Abstract in English

This thesis represents a compact, function and in some ideas new algorithm how to render volumetric clouds. It means the clouds in which you can fly. However, we have to keep in our minds the main accent of this work – this is an algorithm for computer games. No physical processes simulations, dynamic shape changing or various types of clouds are contained. The Alpha and Omega is rendering speed (with a strong usage of GPU) and realistic impression for an average player (which is typical for all parts of game engine).

What is the exact input then? It is to render – as realistically as possible – the volumetric cloud with real-time lightning for minimal computing power.

The basis of the algorithm is ray-casting thru 3D texture and two-passes render of this to an imposter. The imposter is basically a billboard which isn't updated every frame, but just if it is needed. With these clouds we fill the sky in. After that wind and real-time lightning is applied on them. You will find in this document detail expression of main algorithm as well as principles of expletory techniques (lightning, wind, level of detail etc.) and supporting content (e.g. airplane model, textures data).

Key word: cumulus, volumetric cloud, imposter, billboard, texture, GPU, vertex shader, pixel shader, level of detail, real-time lightning

Obsah

ORIGINÁL ZADÁNÍ 2
PROHLÁŠENÍ 4
PODĚKOVÁNÍ 5
ABSTRAKT V ČEŠTINĚ 6
ABSTRACT IN ENGLISH 7
OBSAH 8
1 ÚVOD 10
1.1 CÍL PRÁCE 11
1.2 POPIS JEDNOTLIVÝCH KAPITOL 12
2 TEORETICKÁ ČÁST 14
2.1 REÁLNÉ MRAKY 15
2.1.1 Vznik mraku 15
2.1.2 Druhy mraků 15
2.1.3 Cumulus 17
2.1.4 Základní optické vlastnosti mraku 18
2.2 VIRTUÁLNÍ MRAKY – HISTORIE 20
2.3 VIRTUÁLNÍ MRAKY – VYPOZOROVANÉ VLASTNOSTI 23
2.4 VIRTUÁLNÍ MRAKY – VYČTENÉ VLASTNOSTI 28
2.4.1 Modelování mraku 28
2.4.2 Zobrazování 31
2.4.3 Další užitečné metody 33
2.5 SUMARIZACE POZNATKŮ 37
3 REALIZAČNÍ ČÁST 40
3.1 TVORBA PODKLADŮ 40
3.1.1 Model letadla 41
3.1.2 SkyBox a slunce 44
3.2 PROGRAMOVÁ REALIZACE 49
3.2.1 Programovatelná pipeline 49
3.2.2 Generování mraku 51
3.2.3 Vykreslování mraku 53
3.2.4 Vykreslování mraku na GPU 54
3.2.5 Průlet mrakem 56
3.2.6 Impostery 59
3.2.7 Zjišťování viditelnosti mraku 63
3.2.8 Osvětlení mraku 65
3.2.9 SkyBox, slunce a vítr 70

3.2.10 Alfa průhlednost 71
3.2.11 Řízení úrovně detailů 75

4 EXPERIMENTY A VÝSLEDKY 78

4.1 EXPERIMENTY A VÝBĚR VHODNÉHO ŘEŠENÍ 79
4.2 VÝSLEDNÁ MĚŘENÍ NAŠEHO PROGRAMU 80

5 ZÁVĚR 83

PŘEHLED ZKRATEK 85

PŘEHLED OBRÁZKŮ 86

PŘEHLED TABULEK 88

PŘEHLED GRAFŮ 89

PŘEHLED VÝPISŮ PROGRAMŮ 90

PŘEHLED VZORCŮ 91

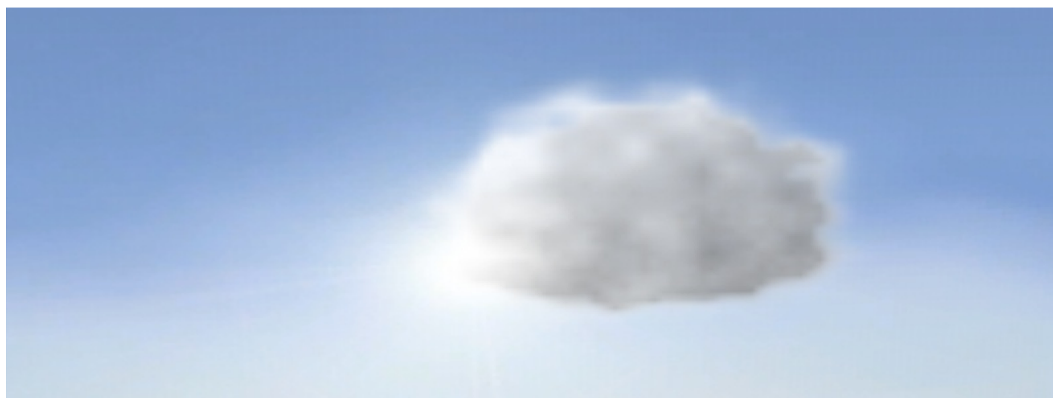
CITOVANÁ LITERATURA 92

PŘÍLOHA A - GRAFICKÉ VÝSTUPY 94

PŘÍLOHA B - HLAVNÍ PIXEL SHADER 96

PŘÍLOHA C - UŽIVATELSKÁ PŘÍRUČKA 100

1 Úvod



Obrázek 1.1: Ukázka výstupu našeho programu – mrak typu Cumulus.

Počítačové hry[1] prodělaly za relativně krátkou dobu své existence obrovský vzestup a nyní je toto odvětví virtuální zábavy stejně výdělečné jako celý filmový průmysl. Jeho důležitost je nesporná a plní různorodé poslání. Od odreagování a zlepšení postřehu, přes edukativní přínos dětem, či poskytnutí zážitků zdravotně handicapovaným až k tomu, že nabízí možnost prožívat vysněná dobrodružství a některým lidem vynahrazuje šeď běžného života. S možnými výdělky z her se zvětšuje i konkurence, stoupají nároky na hry a tím i výdaje. Zatímco ještě před dvaceti lety tvořil hru jednotlivec, nyní se na výrobě podílejí týmy čítající zpravidla desítky a nezřídka kdy i stovky vývojářů, z nichž se každý soustředí na specifickou oblast vývoje: grafika, fyzika, ozvučení, herní logika, herní design, atd. My se zaměříme na první jmenovanou, a to grafika.

[11] Obecným termínem *počítačové hry* označuje autor všechny hry určené pro současné herní PC a herní konzole XBOX 360 a PlayStation 3. Pro tyto tři platformy jsou v podstatě shodné nároky na vizuální podobu a z vlastní zkušenosti může autor potvrdit, že i vývoj samotné hry probíhá téměř najednou. Zjednodušeně tedy, co z hlediska výpočetní náročnosti funguje na jedné z těchto platforem, funguje i na ostatních.

Naopak do této skupiny nebudeme pro naše účely řadit Nintendo WII, které je na úrovni výkonu herních konzolí minulé generace a ani kapesní konzole typu PlayStation Portable a Nintendo DS.

Trendy grafiky se v současných videohrách rozdělují na dva hlavní proudy. Na jedné straně proud spíše umělecký, který představuje odklon ke grafice stylizované (*komiksová*, *abstraktní*, *retro*, *noir* apod.), a na straně druhé snaha o co největší realističnost a přiblížení virtuálního prostředí tomu opravdovému. Opět si zde upřesněme zaměření této práce – jedná se o druhý jmenovaný proud.

1.1 Cíl práce

V upřesnění buďme ještě důslednější a zkusme si představit typický produkt – hru – ve které bude mít tato práce využití. Představme si například leteckou akční hru z druhé světové války s realistickou grafikou. Co by měla obsahovat? Rozhodně především adrenalinové letecké souboje. A co je pro tyto souboje potřebné? Jsou to rozhodně letadla a vzdušný prostor, ve kterém se pohybují. Ten vyplňují mraky, kterými lze prolétávat, ve kterých se lze skrýt a ze kterých také může hráče překvapit nepřítel.

A právě tvorba takových mraků je popsána v tomto dokumentu. Pojďme si nyní konkretizovat, co takový jeden mrak rozhodně musí splňovat. Musí být osvětlen, posouván větrem, musí vypadat realisticky, a to jak při pohledu na celý mrak, tak zároveň i při průletu skrze něj. A těmito mraky (viz Obrázek 1.1) je třeba vyplnit celou oblohu.

Toho všeho je třeba dosáhnout za co nejmenší výpočetní výkon. Hra totiž mimo mraků obsahuje stovky dalších objektů, a proto si můžeme „ukrojit“ opravdu jen pár procent výkonu počítače.

Jak vidno, nestojíme před jednoduchým úkolem a bude třeba udělat řadu ústupků od realističnosti ve prospěch snížení vykreslovací náročnosti. Budeme hledat přesně tu hranici, kdy mrak stále vypadá dobře a již se dá reálně využít.

To je totiž jediný možný přístup při tvorbě počítačové hry, a přesně tak funguje každá část současných komerčních herních *enginů*[2]. Pokud si jakoukoli moderní hru rozebereme, zjistíme, že je každý prvek udělán přesně tak, aby pro průměrného hráče – konzumenta – vypadal věrohodně, dobře, a tvořil tak kompaktní herní svět, a přitom se dal vykreslit na současném hardware. Ten je – i přes dramatický nárůst v posledních letech – stále příliš pomalý, aby se na něm dala v reálném čase (dále *real-time*) vykreslovat reálná geometrie, reálně osvětlená a s reálnou fyzikou. Do doby, než bude možno *real-time* vykreslovat celé virtuální prostředí technikou, která se přibližuje principu, jakým vidí skutečný svět lidské oko (např. techniky založené na způsobu pokročilého *ray-tracingu*), bude vždy tvorba grafiky pro počítačové hry hledáním kompromisů a ústupků.

1.2 Popis jednotlivých kapitol

Základem této práce je pět číslovaných kapitol. První je *Úvod*, poté následuje *Teoretická část*, ve které jsou probírány možné postupy řešení a jsou vybrány ty nejlepší pro tuto práci, které budou implementovány. Postup a principy implementace jsou zachyceny v kapitole třetí, nazvané *Realizační část*. Dále

[2] *Herní engine* je jádro, na kterém je postavena celá hra. Většinou zaštiťuje grafiku, ozvučení, stejně jako správu zdrojů, vstupů, výstupů, ovládání atd. Je v nich standardně implementován skriptovací jazyk (často na bázi jazyků Lua, JavaScript, či C), kterým se pak popisuje herní logika. Samotný herní engin bývá kvůli rychlosti zpravidla naprogramován v jazyce C++ . S přibývajícím výkonem, nároky na přenositelnost či efektivitu vývoje se již objevují enginy psané v modernějších jazycích (např. C# a Java).
Kvůli rychlosti vykreslování také většinou herní enginy počítají maximum věcí na GPU. Uživatelsky herní engine často vypadá jako 3D editor (typu 3ds MAX), do kterého lze umísťovat objekty, skripty (herní logiku), a který je snadno modulovatelný.
Mezi známé světové herní enginy patří Unreal Engine od Epic Games, či id Tech od id Software, mezi českými a slovenskými uveďme CPAL3D od Centauri, či CloakNT od týmu Cauldron. Více informací viz (Anonym, 2010).

následují *Experimenty a výsledky*, kde jsou uvedeny měření rychlosti zde popsaného řešení, jakož i další srovnání a zhodnocení kvality závěrečných výstupů (viz Obrázek 1.1). V poslední číslované, páté, kapitole *Závěr* najdeme poté jak celkovou rekapitulaci, tak dále zhodnocení, nakolik bylo docíleno vytyčených cílů a pomocí jakých dalších úprav by ještě možné naše stávající řešení vylepšit.

K této diplomové práci je také přiložen disk DVD, který obsahuje mimo jiné i spustitelný program s naším řešením a různé doprovodné materiály.

2 Teoretická část

Na začátku této kapitoly je třeba zmínit, že autor využívá existujících dílčích řešení minimálně a snaží se většinou přijít s vlastním nápadem, či rozluštěním daného problému. Je sice pravděpodobné, že s některými nápady přišel již někdo dříve, ale to lze v této oblasti jen velmi těžko zjistit, jelikož uceleně publikovaných a veřejně přístupných řešení tohoto problému není mnoho. Pokud totiž většina herních vývojových studií nějaký postup vymyslí a zahrne ho do svého herního *enginu*, drží toto svoje know-how pod pokličkou. To má dva prozaické důvody. Zaprvé snaha být o krok před konkurencí a neulehčovat jí zbytečně práci a zadruhé – a to je ještě častější – velká část týmů málo dokumentuje svůj *engine*, což dokonce platí i u *enginů*, které jsou komerčně prodejné. Uzavřené funkční celky – mezi které patří i mraky – jsou sice v herním *enginu* obsaženy, ale tvůrci těchto částí již většinou netráví další čas psaním komentářů, tutoriálů apod. Některé popsané postupy však přece jenom existují (naštěstí existuje i sorta vývojářů, pohybujících se obvykle po různých diskusních fórech, kteří vykazují snahu pomáhat ostatním). Z tohoto důvodu je logické rozdělit kapitolu na sekce *Vyčtených vlastností* a neméně důležitou sekci *Vypozorovaných vlastností*. V té si projdeme několik současných her a pozorováním zjistíme, jakým způsobem jsou v nich mraky tvořené, a zkusíme si odvodit řešení některých dílčích problémů. Oběma těmto sekcím bude ještě předcházet sekce *Reálné mraky*, ve které si nejprve řekneme pár fundamentálních informací o tom, jak mraky vznikají, čím jsou tvořeny a jaké jsou jejich základní typy a vlastnosti a také sekce *Virtuální mraky – Historie*, kde si ve zkratce ukážeme vývoj mraků v počítačových hrách, čímž zdůrazníme posun, který se zde odehrál. Nabyté zkušenosti a znalosti si poté shrneme v závěrečné sekci této kapitoly s názvem *Sumarizace poznatků*.

2.1 Reálné mraky

Oblak neboli mrak je nepravidelný útvar v atmosféře, který je tvořen drobnými kapkami vody nebo krystalky ledu. Jeho tvar se neustále mění. Příčinou toho jsou již zmíněné krystaly, které vlivem gravitace mají tendenci padat na zemský povrch a dále vzduch, který neustále proudí buď vlivem větru (proudění vzduchu z místa vyššího atmosférického tlaku do místa nižšího tlaku) nebo teploty (vzduch s vyšší teplotou stoupá vzhůru). Z dálky je mrak vnímán jako souvislý objekt od bílé po tmavě šedou až k tmavé barvě s nádechy fialové barvy, ale má-li člověk možnost fyzicky prolétávat mračnem, zjistí, že jde vlastně o mlhu.

2.1.1 Vznik mraku

Obecně a velmi zjednodušeně se dá říci, že mrak vzniká kondenzací nasycených vodních par. Teplý vzduch obsahující vodní páry stoupá výše do atmosféry, kde klesá atmosférický tlak. Vzduch se rozpíná, a tím se i ochlazuje. Jakmile dosáhne rosného bodu, dochází ke kondenzaci nasycených vodních par. Jako rosný bod je označována teplota, při níž je relativní vlhkost vzduchu maximální, tedy obsahuje 100% vody (viz (Lička, et al., 2010)). Vše pak také závisí na přítomnosti kondenzačních jader, hustotě, velikosti částic atd.

2.1.2 Druhy mraků

Mraky můžeme rozdělit do dvou základních kategorií:

- slohy (*stratus*), což je v podstatě mlha vyskytující se v nejnižších výškách,
- kupy (*cumulus*), vznikající na studené frontě.

Rozeznáváme 10 základních typů mraků (viz Tabulka 2.1). Z hlediska složení rozlišujeme mraky vodní, ledové a smíšené. Výška výskytu jednotlivých

druhů mraků závisí na podnebném pásu, neboť atmosféra a troposféra nejsou na Zemi stejně široké, ale jejich výška se s přibývající zeměpisnou šířkou zmenšuje (tzn. nejširší vrstvy jsou v oblasti rovníku, nejnižší v polárních oblastech). Výšky uvedené v Tabulka 2.1 jsou vztažené k našemu mírnému podnebnému pásu.

LATINSKÝ NÁZEV	ČESKÝ NÁZEV	ZKRATKA	VÝŠKA [km]
Cirrus	Řasa	Ci	8-13
Cirrocumulus	Řasová kupa	Cc	5-13
Cirrostratus	Řasová sloha	Cs	5-13
Altocumulus	Vyvýšená kupa	Ac	1.5-7
Altostratus	Vyvýšená sloha	As	2-7
Stratocumulus	Slohová kupa	Sc	0.1-2
Stratus	Sloha	St	0-2
Cumulus	Kupa	Cu	0-2
Nimbostratus	Dešťová sloha	Ns	2-7

Tabulka 2.1: Druhy mraků, jejich latinský i český název, zkratka a výška, ve které se v mírném podnebném pásu vyskytují. (Fillinger, et al., 2010)

V této práci nás bude zajímat *Cumulus*, který je v našem podnebném pásu nejčastěji se vyskytující. Pro hráče-konzumenta je typickým představitelem mraku, a proto se právě na jeho generování později zaměříme.

2.1.3 Cumulus

Podmínky ke vzniku tohoto typu mraku jsou různé. Měl by ale existovat alespoň minimální teplotní rozdíl mezi různými druhy zemského terénu. Na to má vliv různá schopnost absorpce slunečních paprsků vlivem vegetace, hornin atd. Nejpříznivější terén ke vzniku *cumulu* je členitý bez zvýšeného výskytu vodních ploch, které se ohřívají nejpomaleji. *Cumulus* se snadno vytváří i v městské zástavbě (viz Obrázek 2.1), protože stavební plochy se zahřívají poměrně rychle. Velice často se také vyskytují v místech pahorkatin a vrchovin.

Velikost *kumulu* je také velmi různá, maximálně může být až 10 km široký a 20 km vysoký. Převládající barva je bílá, zejména z pohledu ze strany a shora. Při pohledu zdola se jeví jako šedý s bílými okraji. Obecně platí pravidlo, že čím je mrak vyšší, tím je tmavší. Příčinou jsou sluneční paprsky, které procházejí mrakem. Okraje jsou jasně ohraničené, ale při procesu zániku mraku se ostré hrany ztrácí. Zpravidla se všechny pohybují jedním směrem a rychle mění svůj tvar. Oblak je až na výjimky nesrážkovým typem mraku.

Obrázek 2.1: Mrak typu cumulus nad městskou zástavbou. Fotografie pořízená 9. května 2010, 17:50 hod, Bratislava, Slovenská Republika.

Cumulus se dále dělí na čtyři podkategorie:

1) *Cumulus fractus* (zcela bílý, často je prvotním nebo konečným stádiem kumulu, menší rozměr),

2) *Cumulus humilis* (nízký, vertikální a horizontální rozměr bývá rovnoměrný, dalším vertikálním vývojem přechází do následující formy),
3) *Cumulus mediocris* (z pohledu zdola rovná základna, tmavší barvy se středním vertikálním vývojem),
4) *Cumulus congestus* (velké zastoupení šedi v základně, vrcholky jasně bílé, značný vertikální vývoj, dále již přechází do formy *cumulonimbus* – viz Tabulka 2.1).

Druhý a třetí jmenovaný bývá využíván v bezmotorovém letectví. Pokud letadlo najde výstupní proud vzduchu, efektivně nabírá výšku. Pokud letadlo tohoto využívá vícenásobně, může doletět až stovky kilometrů. Obecně pro leteckou dopravu jsou nebezpečné kumuly uvedené jako poslední, neboť v této fázi vývoje jsou pohyby vzduchu tak prudké, že mohou letadlo vážně poškodit. Je to jeden z možných zdrojů turbulencí. Více podrobností viz (Jankovič, et al., 2006).

2.1.4 Základní optické vlastnosti mraku

V dalších kapitolách této práce se budeme setkávat s pojmy, které je potřeba nejprve objasnit. Týkají se optických a fyzikálních vlastností mraku. Mrak totiž řadíme mezi takzvané opticky aktivní prostředí (stejně jako např. mlhu nebo kouř). Při průchodu tímto prostředím nastává rozptyl světla (*scattering*).

Rozptyl světla se v reálu projevuje jevy záře kolem světelných objektů, měkkými stíny, osvětlením tmavých oblastí apod. (viz (Ondřej, 2007)). Jak již víme, mrak je tvořen ohromným počtem miniaturních částic. Každá tato částice při interakci se světelnou energií část této energie absorbuje (*absorbtion*) a přemění ji na jiný druh energie (např. teplo) a část opět vyzáří (to je právě společný jmenovatel opticky aktivních prostředí). Právě vyzáření a absorpce dohromady tvoří jev zvaný rozptyl světla. Pohasnutí (*extinction*) poté chápeme

jako součet absorpce a rozptylu a všechny tyto tři veličiny jsou úměrné hustotě částic (*density*) – viz (Harris, 2002).

Koeficient rozptylu světla neboli albedo vyjadřuje procentuální propustnost částice (viz Vzorec 2.1).

$$Albedo = \frac{Rozptyl\ světla}{Pohasnutí}$$

Vzorec 2.1: Albedo, nebo také koeficient rozptylu světla.

Výše zmíněná částice se tedy chová podobně jako bodový zdroj světla (viz níže). Pokud by neinteragovala s ostatními a pouze vyzářila část světelné energie, jednalo by se o jev, který nazýváme jednoduchý rozptyl světla (*single scattering*). Její světlo však osvětluje částice okolní, které opět jeho část pohltí a vyzáří, a proto v reálu dochází k jevu, který nazýváme vícenásobný rozptyl světla (*multiple scattering*). To je tedy rozptyl světla z několika okolních částic.

Částice nerozptyluje světlo pod všemi úhly stejně – záleží na její velikosti a dalších faktorech, obecně však lze říci, že nejvíce světla se buďto zrcadlově odrazí, anebo projde skrze částici přímo ve směru, z kterého přišlo. Toto chování poté vyjadřuje fázová funkce (*phase function*). Ta má řadu aproximací – některé se využívají pro simulaci prostředí s malými částicemi (Rayleighova fázová funkce, viz Vzorec 2.2), jiné spíše pro velké částice (viz *Mie Scattering* - (McLinden, 1999)).

$$p(\alpha) = \frac{3}{4}(1 + \cos(\alpha)^2)$$

Vzorec 2.2: Rayleighova fázová funkce pro aproximaci průchodu světla částicí - α představuje úhel mezi zdrojem světla a pozorovatelem.(Harris, 2002)

Pokud výše zmíněnou funkci pro aproximaci vyzáření světla částicí v osvětlovacím modelu neuvažujeme, jedná se o tzv. izotropní rozptyl světla (tzn. rozptyl nezávislý na směru – *isotropic scattering*), naopak rozptyl světla

využívající fázové funkce se nazývá neizotropní (*anisotropic scattering*) a vytváří mj. typické stříbrné orámování Slunce, pokud se na něj podíváme skrze mrak.

2.2 Virtuální mraky – Historie

Letecké simulátory vznikaly již před rokem 1980 a byly jedny z vůbec prvních počítačových her. Tím pádem obloha, mraky a jejich zobrazení prodělaly velmi významný vývoj.

Mezi první letecké simulátory řadíme subLOGIC Flight Simulator 1 z roku 1979, který ještě neobsahoval žádné zobrazení oblohy ani mraků (viz Obrázek 2.2).

Obrázek 2.2: subLOGIC Flight Simulator 1, 1979. Pravděpodobně první legendární simulátor. (Havlík, 2010)

Obrázek 2.3: MS Flight Simulator 3.0, 1988. Jedno z prvních zobrazení mraků v počítačových hrách. (Havlík, 2010)

V pozdějších verzích se začala obloha odlišovat jinou barvou, popř. jiným stupněm šedi. První pokusy o zobrazení mraků se však objevily až s příchodem Microsoft (dále MS) Flight Simulator 3.0 o devět let později. Obrázek 2.3 ukazuje mraky, které byly tvořeny bílými kruhy vykreslovanými na popředí.

Ještě ten samý rok však vyšla další přelomová hra F-19 Stealth Fighter od společnosti MicroProse. Na této hře mimo jiné pracoval i dnes již legendární vývojář a vizionář Sid Meier. Mraky zde již byly zpracovány vektorově, v podobě dvou šedivých šestiúhelníků nad sebou. Ty byly pevně pozicované na obloze, vrchní byl světlejší, spodní tmavší (viz Obrázek 2.4).

Obrázek 2.4: F-19 Stealth Figther, 1988. Průkopník vektorových mraků.

Obrázek 2.5: F-15 Strike Eagle II, 1989. Další zvýšení realističnosti oblohy, přidáním mlžného oparu.

Stejného principu využíval i následník F-15 Strike Eagle II, který navíc přidával mlžný opar na horizontu (viz Obrázek 2.5).

V roce 1991 přišla další verze MS Flight Simulatoru, s podobnými technologiemi jako F-15 Strike Eagle II, avšak mraky nabízely větší variabilitu odstínů šedi (viz Obrázek 2.6). V následující verzi z roku 1993 poté jednolité šestiúhelníky vystřídaly textury nanesené na čtvercích paralelně umístěných nad terénem (viz Obrázek 2.7). Tento způsob zobrazení poté používala řada her a časem se v podstatě jen zvyšovalo rozlišení textur.

Obrázek 2.6: MS Flight Simulator 4.0, 1991. Zobrazení mraků jako šestiúhelníků s různými stupni šedi.

Obrázek 2.7: MS Flight Simulator 5.0, 1993. Pro zobrazení mraků již použity textury. (Havlík, 2010)

V roce 2000 vyšla (opět pod taktovkou MS) hra Crimson Skies, která mimo perfektní hratelnosti zpracovávala již mraky „moderním“ způsobem –

pomocí řady billboardů čelně otočených k hráčovi, pokrytých texturou a shluknutých do větších celků. Jak je vidět na Obrázek 2.8, textury jsou neosvětlené, mají nízké rozlišení i barevnou paletu, ale šlo již vlastně o přístup, který se v notně vylepšené podobě používá stále.

Obrázek 2.8: Crimson Skies, 2000. Vysoko hodnocená hra s – na svou dobu – velmi dobrou grafikou. Mraky již tvořeny jako billboardy s nanesenou texturou. Bohužel se billboardy uhýbají kameře, takže jimi nelze proletět.

Toto bylo pro zajímavost a lepší představu něco z historie tvorby virtuálních mraků (pro více podrobností můžete čerpat z (Havlík, 2010)), a nyní se přenesme již do současnosti.

2.3 Virtuální mraky – vypozorované vlastnosti

Nejprve bylo třeba vyzkoušet moderní hry a zjistit, jak v nich mraky vypadají, jak se chovají a jak jsou tvořeny. Zaměřili jsme se na různé aspekty mraků a oblohy, které nás z pohledu této práce zajímají a lze je vypozorovat. Shrnuli jsme si je v přehledné tabulce (viz Tabulka 2.2). Z tohoto se poté dá odvodit řada důležitých vlastností. Zmíněnými aspekty jsou *stavba mraku* (z čeho je mrak tvořen), *vzhled mraku cumulus* (jak vypadá v této hře mrak typu *cumulus*), *osvětlení mraku*, *průlet mrakem* (efekt při průletu), *řízení úrovně detailů mraku* (různě detailní zobrazení při odlišné vzdálenosti od mraku), *způsob mizení mraku v dálce*, *pohyb mraku* (vítr, popř. dynamická změna tvaru), *rozmístění mraků po obloze* a *hodnocení celkového dojmu* (kde si shrneme, jak na nás obloha vizuálně působí a ohodnotíme si ji jedním až deseti z maximálního počtu deset bodů).

Jelikož ve specifikacích externího konzultanta této práce (a zároveň zadavatele) stálo, že řešení má být navrhnuto především s ohledem na vizuální atraktivitu, bylo i při výběru testovaných her hlavním kritériem grafická úroveň daného titulu. Proto jsme se rozhodli pro Microsoft Flight Simulator X a Tom Clancy’s HAWX. Obě představují nynější vrchol vizuálního zpracování, jedná se o AAA tituly[3] a každý míří do jiného spektra trhu. Zatímco MS Flight Simulator X je zástupcem rodiny realistických simulátorů, Tom Clancy’s HAWX je velmi arkádový, s minimem ovládacích prvků a značně zjednodušeným letovým modelem. Oba tituly disponují atraktivním zpracováním oblohy i mraků.

[3] Viz citace z Oborového Projektu V. Geršla (viz (Geršl, 2008)): „Pojmem AAA titul jsou označovány nejdražší hry z hlediska vývoje s největší kvalitou – skvělá grafika, hratelnost, dobré odladění, drahé na vývoj, vývoj několik let, prodej za plnou cenu.“

Obrázek 2.9: Ukázka zpracování mraků cumulus ve hře MS Flight Simulator X.

Obrázek 2.10: Ukázka zpracování mraků cumulus ve hře Tom Clancy's HAWX.

	MS FLIGHT SIMULATOR X	**TOM CLANCY'S HAWX**	**AIR CONFLICTS**
Stavba mraku	Cca deset náhodně otočených billboardů.	Přibližně deset až dvacet různorodě otočených billboardů.	Jeden až několik desítek billboardů natáčejících se ke kameře.
Vzhled mraku Cumulus	Téměř shodný stupeň šedi základny i vrchu mraku. Tvarově ale velmi povedené.	Mrak připomínající tvarově cumulus, který má velké rozdíly světlosti.	Nahrubo vyskládané billboardy. Pouze jedna textura!
Osvětlení mraku	Barva billboardu vzhledem k pozici slunce.	Dle pozice od slunce s následně použitým (zřejmě single) scatteringem.	Žádné.
Průlet mrakem	Billboardy v blízkosti se přestanou natáčet a částečně se zprůhledňují. Nekryjí letadlo.	Billboardy se zprůhlední cca na 50% a poté jimi proletíme. Nekryjí však letadlo.	Někdy se povede proletět, jindy se billboardy uhýbají kameře.
Řízení úrovně detailů mraku	Textura uložená v různých velikostech. Se vzdáleností se zvyšuje počet billboardů.	V dálce imposter s texturou mraku, zblízka billboardy + různě velké textury.	Textura uložená v různých velikostech.
Způsob mizení mraku v dálce	Zmenšují se až k horizontu.	Postupné zprůhledňování v mlžném oparu na horizontu.	Po určité vzdálenosti se v oparu zprůhlední.
Pohyb mraku	Žádný, nebo těžko postřehnutelný.	Žádný, nebo těžko postřehnutelný.	Žádný.
Rozmístění mraků po obloze	Libovolné, či načtené z reálných dat.	Víceméně pravidelně rozmístěné shluky.	Mraky se shlukují do větších celků. Jinak libovolné.
Hodnocení celkového dojmu (max. 10/10)	Řada rozličných pěkných mraků, bez zajímavého efektu prolétávání. Často chybné vyhodnocení viditelnosti a zmizení/prolnutí mraku. 6/10.	Ač rozhodně nedosahují propracovanosti databáze mraků Flight Simulatoru, jedná se o nejvíce efektní mraky. Stále však viditelné billboardy. 8/10.	Pouze jedna textura, žádné osvětlení a prolétávání. 3/10.

Tabulka 2.2: Vypozorované vlastnosti oblaků a oblohy ve hrách Microsoft Flight Simulator X, Tom Clancy's HAWX a Air Conflicts.

Obrázek 2.11: Ukázka zpracování mraků cumulus ve hře Air Conflicts.

Testovací trojlístek doplňuje ještě hra Air Conflicts od slovenských 3Division, která naopak pro porovnání zastupuje menší projekty s nízkými náklady na vývoj i výnosy z prodeje (tzv. *budget* hry).

Na Obrázek 2.9, Obrázek 2.10 a Obrázek 2.11 jsou zachyceny mraky z výše zmíněných her. Každá z těchto her přistupuje k dílčím řešením jinak a stejně je zpracování mraků u všech poměrně atraktivní. Přece jen je však vidět, že hra Air Conflicts za oběma komerčními tituly ztrácí. Co například není patrné z obrázku, je, že zatímco v MS Flight Simulatoru X (dále MSFS X) a Tom Clancy's HAWX (dále TC HAWX) jsou všechny mraky dosažitelné (lze k nim doletět), tak v Air Conflicts (dále jen AC) jsou vzdálenější mraky pouze texturou „v nekonečnu" a doletět se dá jen ke tmě v popředí.

Zhodnoťme nyní námi vyzkoušené hry. Bohužel je dopředu třeba říci, že ani jedna neodpovídá autorově představě o ideálním vzhledu mraků. Najdou se

však i dílčí řešení, kterými se inspirovat lze. Pojďme si tedy postupně projít tabulku vlastností a porovnat je vždy v přístupu k řešení daného problému.

Stavba mraku je v MSFS X i TC HAWX tvořena řadou různorodě natočených billboardů, nenatáčejících se ke kameře, zatímco v AC se tyto billboardy natáčejí. To v kombinaci s pouze jednou texturou vypadá nevábně. Různorodě natočené billboardy tedy vítězí.

Vzhled mraku typu *cumulus* je poté, dle subjektivního názoru autora, nejpovedenější ve hře TC HAWX. Sice zřejmě není tak realistický, ale přesně splňuje očekávání hráče. Velké barevné rozdíly, stínování, hezký tvar. Mraky v MSFS X jsou naopak blíže opravdové předloze, co se týká tvaru, ale výsledek není tak efektní.

Osvětlení a stínování mraku je opět nejslabší ve hře AC, ale ani MSFS X v tomto směru nikterak nevyniká a je zřejmě největší slabinou této hry. Zato v TC HAWX vypadá např. pohled na slunce skrze mrak velmi dobře.

Průlet mrakem není udělán dobře v žádné hře. Nejhorší je AC, kde se billboardy většinou uhýbají před kamerou. Ve zbývajících hrách zase, i když vidíme letadlo před sebou, nikdy nevletí do mraku. Co ale stojí za zmínku, je zprůhledňování billboardu, ke kterému se blížíme. Toho využívají hry MSFS X i TC HAWX a u obou vypadá velmi dobře.

Řízení úrovně detailů mraku je vlastnost, která se velmi špatně vypozorovává a vnitřní mechanismy mnohdy ani vypozorovat nelze. Přesto vypadá, že všechny hry využívají technologii takzvaných *MIP Map*[4]. MSFS X k tomu přidává zvyšování počtu billboardů se vzdáleností a TC HAWX zřejmě vzdálené billboardy ještě vykreslí pouze do jednoho, a tím ušetří další vykreslovací čas. Vypadá to tedy, že i v této disciplíně má navrch TC HAWX, což

[4] Viz citace z (Rstralberg, 2008): „MIP mapy jsou vlastně předpočítaná kolekce textur, které vycházejí z textury původní. Podmínkou pro tvorbu MIP map je, že původní textura musí mít délku stran rovnou k-té mocnině čísla 2 (přičemž k je libovolné). MIP mapu potom tvoří kolekce textur, které mají vždy poloviční délku strany než předchozí textura. Takto se dělí do té doby, dokud není délka alespoň jedné strany textury rovna jedné. Když je velikost původní textury tedy například 256 × 256 pixelů, vytvoří se z něj dalších 8 textur o velikostech 128 × 128, 64 × 64, 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, 1 × 1. Potom se všech devět textur uloží do jednoho obrázku, ze kterého se v programu potom načítají vždy nejbližší textury, mezi nimiž se interpoluje."

potvrzuje i fakt, že tato hra je dostupná i na mobilních platformách (např. pro zařízení iPhone, iPod Touch a iPad), a i tam vypadají mraky velmi dobře.

V MSFS X se oblaka postupně zmenšují až k horizontu. Tento způsob mizení mraku působí daleko lépe, než zprůhlednění vzdáleného mraku, které využívají zbývající dvě hry.

Pohyb mraku nemá žádná hra, popřípadě je velmi těžko pozorovatelný. Rozmístění po obloze podle aktuálních dat oblačnosti, které využívá MSFS X, je sice velmi zajímavé, ale z čistě vizuálního hlediska působí stejně věrohodně, jako náhodné rozmístění mraků ve hře TC HAWX.

Z tohoto shrnutí nám tedy vychází jako vítěz TC HAWX. Samozřejmě záleží na úhlu pohledu – mraky v MSFS X jsou určitě sofistikovanější z pohledu počtu typů mraků, možnosti rozmístit je dle opravdové oblačnosti atd., ale z pohledu zážitku pro hráče a efektnosti na našeho vítěze ztrácí.

2.4 Virtuální mraky – Vyčtené vlastnosti

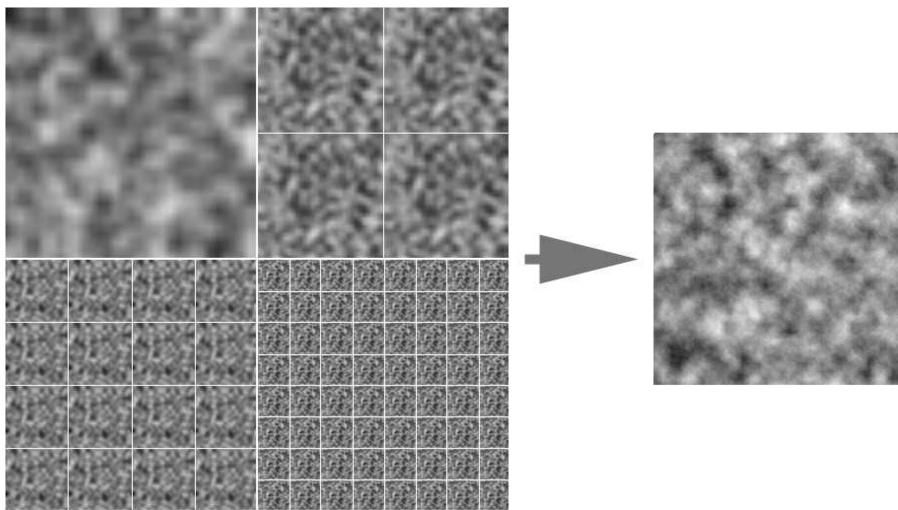
Většina prací pojednávajících o tvorbě mraků v počítačové grafice rozděluje problematiku na dva logické celky, které lze nalézt i v názvu této diplomové práce - modelování mraku a jeho následné vykreslování. Pojďme si tedy zmapovat některé tyto metody.

2.4.1 Modelování mraku

Modelování mraku pojednává o generování a organizování dat, která jsou použita pro reprezentaci mraku v počítači.

T. Roden a I. Parberry (Roden, et al., 2005) například modelují mrak za použití čtyř textur – první udává tvar, druhá náhodný šum, třetí detail a čtvrtá gradient pro západ slunce. Tato metoda však není určena pro 3D objemové mraky a slouží jen pro generování 2D mraků.

H. Elias a M. Fairclough (Elias, et al., 1998) využívají Perlinův šum[5] pro generování procedurálních textur mraku. Jelikož je však jeho generování v reálném čase příliš výpočetně složité, dosahují urychlení tím, že si nejprve předpočítají šum, a ten nanesou na textury o velikosti 32×32, 64×64, 128×128 a 256×256. Výsledná textura je poté tvořena kombinací těchto textur (viz Obrázek 2.12). Perlinův šum dává velmi slušné výsledky, ale stále se bohužel bavíme o tvorbě 2D textury, ani tato práce nepojednává o modelování 3D mraku.



Obrázek 2.12: Obrázek ukazuje kombinaci textur 4 velikostí do výsledné textury. Šum na texturách je generován pomocí Perlinovy šumové funkce. Převzato z (Elias, et al., 1998).

M. J. Harris (Harris, 2002) modeluje mraky pomocí částicových systémů. Jak uvádí, částicové systémy jsou nástrojem, který vznikl pro modelování objektů neurčitého tvaru (mraky, kouř, oheň apod.). Jednotlivá částice v této práci má poté podobu přibližné koule s Gaussovým rozložením hustoty (nejvíce uprostřed a směrem ke krajům hustoty ubývá). Nese si tyto parametry: souřadnice středu, poloměr, hustotu a barvu. S částicemi různé velikosti a hustoty poté vyplní náhodně objem mraku (nebo nechá vyplnění na ruční práci uživatele).

Z. Prokop (Prokop, 2007) využívá uniformní rozdělení prostoru na voxely. To je paralela k pixelu jako nejmenšímu elementu 2D prostoru a

[5] Šum je primitivum, kterým můžeme vyrobit například řadu textur (voda, mramor, mraky atd.). Pokud vyjádříme šum matematickou funkcí, lze produkovat procedurální textury. Perlinův šum lze poté interpretovat jako sumu šumové funkce, které se mění velikost a intenzita. Viz (Perlin, 1999).

představuje nejmenší element v 3D prostoru. Většinou má podobu krychle. V této práci představuje mrak tzv. buněčný automat a jeden voxel poté jednu buňku tohoto automatu. Má svoje parametry *mrak*, *vlhkost* a *aktivace*. Všechny tři jsou logické proměnné (nabývají pouze hodnot ano = 1 a ne = 0). Poté se nastaví první jmenovaná na nulu, ostatní dvě náhodně a spustí se simulace. Ta je tvořena logickými pravidly viz Vzorec 2.3, převzatými z (Dobashi, et al., 2000). Pokud po skončení simulace (neboli proběhnutí daného počtu iterací) ve výsledném mraku buňce vyjde *mrak* = 1, nanese se na ni jedna z 16 pevně definovaných textur.

$$vlhkost(i,j,k,t_{i+1}) = vlhkost(i,j,k,t_i) \wedge \neg aktivace(i,j,k,t_i)$$
$$mrak(i,j,k,t_{i+1}) = mrak(i,j,k,t_i) \vee \neg aktivace(i,j,k,t_i)$$
$$aktivace(i,j,k,t_{i+1}) = \neg aktivace(i,j,k,t_i) \wedge vlhkost(i,j,k,t_i) \wedge f_{aktivace}(i,j,k)$$

$$\begin{aligned} f_{aktivace}(i,j,k) = {} & aktivace(i+1,j,k,t_i) \vee aktivace(i,j+1,k,t_i) \vee aktivace(i,j,k+1,t_i) \\ & \vee aktivace(i-1,j,k,t_i) \vee aktivace(i,j-1,k,t_i) \vee aktivace(i,j,k-1,t_i) \\ & \vee aktivace(i-2,j,k,t_i) \vee aktivace(i+2,j,k,t_i) \vee aktivace(i,j-2,k,t_i) \\ & \vee aktivace(i,j+2,k,t_i) \vee aktivace(i,j,k-2,t_i) \end{aligned}$$

Vzorec 2.3: Přechodová pravidla pro simulaci tvorby mraku. (Dobashi, et al., 2000)

Dalším problémem je také tvar samotného objemu. Ten samozřejmě nelze nechat například ve tvaru krychle a často se definuje nějakým primitivem. Například u výše zmíněného buněčného automatu je jednou z metod využití elipsoidu tak, že k jeho okrajům bude klesat pravděpodobnost počátečního nastavení vlhkosti a aktivace na hodnotu 1 a za hranice elipsoidu bude vždy 0 - viz (Kučera, 2007). Další možností je kupříkladu využití místo elipsoidu Wyvillovu kubickou mapovací funkci pro pokles intenzity se vzdáleností viz Vzorec 2.4 převzatý z (Man, 2007).

$$f(x) = \begin{cases} -\frac{4}{9}r^6 + \frac{17}{9}r^4 - \frac{22}{9}r^2 + 1, & r \leq 1 \\ 0, & r > 0 \end{cases}$$

Vzorec 2.4: Wyvillova kubická funkce. (Man, 2007)

Jak vidíme, je téma modelování mraku velmi obsažné. Hovoří totiž za prvé o datové reprezentaci mraku (tj. částicový systém či uniformní dělení na voxely),

za druhé o vyplnění objemu mraku (náhodné nebo simulace fyzikálních procesů), za třetí o tvaru tohoto objemu a za čtvrté i o tvorbě textury (obzvláště u 2D mraků, kdy vlastně celé modelování mraku spočívá právě v tvorbě textury).

2.4.2 Zobrazování

Mezi základní techniky řadíme zobrazování jedné až několika vrstev billboardů na plochu oblohy (ať již je plochá či kopulovitě vydutá), bez simulace procesu osvětlení apod. T. Roden a I. Parberry (Roden, et al., 2005) využívají tohoto postupu, který provádějí na procesoru grafické karty (GPU). Osvětlení počítají pouze prolnutím předpřipravené textury mraků s předpřipravenou texturou gradientu (viz 2.4.1 Modelování mraku) – výsledek poté vidíme na Obrázek 2.13.

Obrázek 2.13: Ukázka výstupu práce, kde je vykreslování mraků maximálně zjednodušeno - v podstatě se jedná jen o vykreslování několika plochých textur nad sebou. Převzato z (Roden, et al., 2005).

Pokud však upustíme od takto absolutně zjednodušených metod a budeme se snažit soustředit na ty, které na určité úrovni simulují fyzikální a optické charakteristiky mraku (viz 2.1.4 Základní optické vlastnosti mraku) zjistíme, že vykreslování je velmi obtížné. Předchozí práce z tohoto ranku pojednávaly o fyzikálních charakteristikách mraku na různých stupních přesnosti a komplexnosti, a poté tyto aproximace využívaly k zobrazení mraku. J. Blinn (Blinn, 1982) ukázal využití modelů hustoty – představil nízké *albedo*[6]. J. Kajiya a B. Herzen (Kajiya, et al., 1984) rozšířili tento postup o metody vysílání paprsku (*ray-tracing* - viz 2.4.3 Další užitečné metody) skrze objemová data při využití *single* i *multiple scatteringu*.

Nishita et al. ukazuje aproximace a vykreslovací techniky pro globální osvětlování mraků, beroucí v úvahu *multiple anisotropic scattering* a jas oblohy.

Jednou z nejsledovanějších a nejskloňovanějších prací v oblasti tvorby mraků pro herní aplikace je však práce M. Harrise (Harris, 2002). I my se na ní podíváme podrobněji. Jeho postup vykreslování odkazuje nejvíce na vykreslovací techniky prezentované Y. Dobashi et al. (Dobashi, et al., 2000). Metody stínování prezentované Y. Dobashi et al. využívají aproximací *isotropického single scatteringu*. M. Harris rozvíjí tuto metodu o *aproximaci multiple scatteringu* z pohledu slunce (tzv. *forward scattering*) a *anisotropického scatteringu* prvního stupně z pohledu pozorovatele. Jeden z výsledků práce M. Harrise vidíme na Obrázek 2.14.

Obecně však celkový postup vykreslování mraků uvedený v práci M. Harrise můžeme shrnout do několika bodů:

1) Výpočet *multiple forward scatteringu* v *pre-process* fázi (fáze přípravy modelů, výpočtů osvětlení apod., která se provádí pouze jednou, a to před samotným vykreslováním scény) – tento typ *scatteringu* vystínuje mrak.
2) Výpočet *single scatteringu* z pohledu kamery a uložení tohoto výsledku do textury. *Single scattering* se však již nestará o změnu

[6] Jedná se o single scattering aproximaci pro osvětlování v uniformním médiu.

pozice světla, ale pouze zajišťuje korektní zobrazení mraku při změně pozice kamery.

3) Namapování této textury na speciální billboard, zvaný imposter (viz 3.2.6 Impostery).
4) Výpočet bodu 2) a provedení bodu 3) pouze za předpokladu, že se pozorovací úhel kamery vůči mraku změnil o více než konstanta ε.

M. Harris ještě používá fintu, kdy při průletu letadla mrakem vykreslí dva impostery – jeden před a jeden za letadlem a docílí tak velmi pěkného efektu, kdy opravdu letadlo vypadá vnořené do mraku.

Obrázek 2.14: Ukázka výstupu programu M. Harrise. Jeho metoda využívá multiple forward scatteringu a anisotropického scatteringu (Harris, 2002).

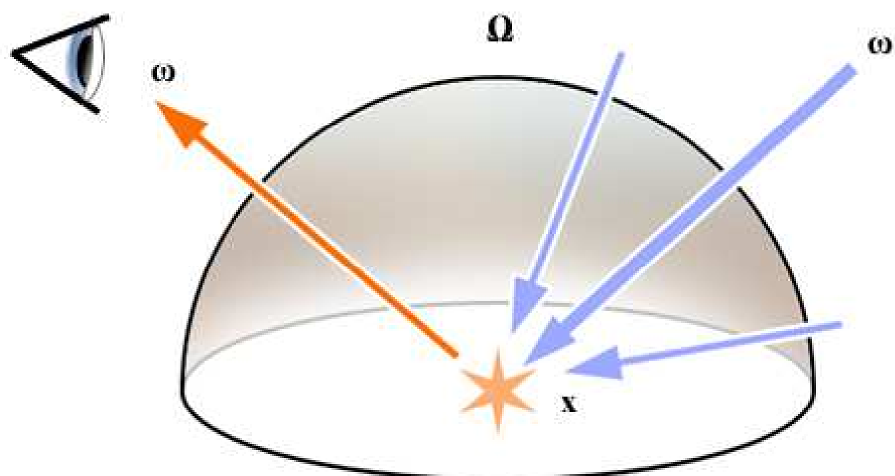
2.4.3 Další užitečné metody

Velmi zajímavou metodou, která není navržena přímo pro zobrazování objemových mraků, ale pro vykreslování objemových těles obecně, je přímé vykreslování objemových těles, konkrétně její mutace pro výpočet na procesoru grafické karty *GPU based – Direct volume rendering* (Purchart, 2009), které vychází z práce D. Weiskopfa *GPU-Based Ray Casting* (Weiskopf, 2004).

Je to v podstatě úprava klasické metody sledování paprsku objemem (*Volume ray casting*). Na Vzorec 2.5 a graficky také na Obrázek 2.15 vidíme zobrazovací rovnici, ze které sledování paprsku vychází.

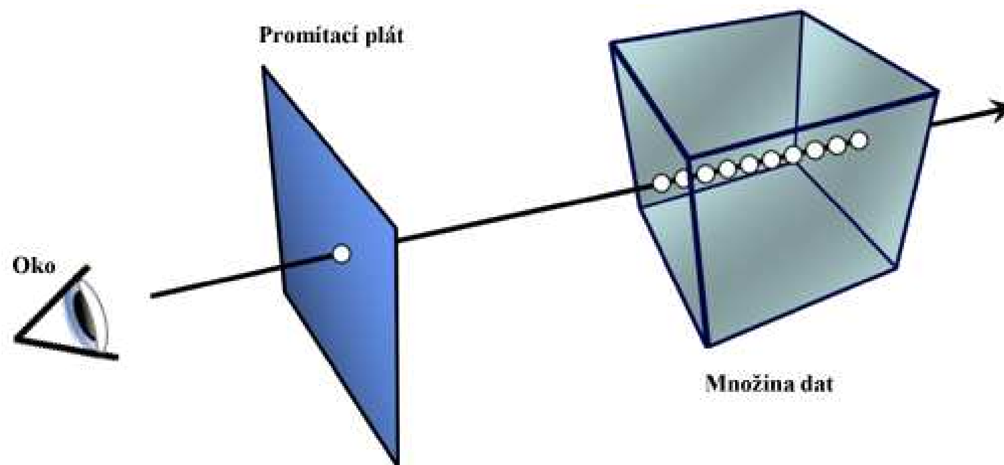
$$L_0(\boldsymbol{x},\omega,\lambda,t) = L_e(\boldsymbol{x},\omega,\lambda,t) + \int_{\Omega} f_r(\boldsymbol{x},\omega',\lambda,t)\, L_i(\boldsymbol{x},\omega',\lambda,t)(-\omega'\cdot\boldsymbol{n})d\omega'$$

Vzorec 2.5: Zobrazovací rovnice – základ pro metodu sledování paprsku objemem, přičemž λ je vlnová délka světla, t je čas, L_0 je výsledné světlo, L_e je vyzářené světlo, $\int_{\Omega}$ je integrál přes polokouli, f_r je odrazivost, L_i vstupující světlo a $-\omega\cdot n$ je útlum vstupujícího světla. (Justin, et al., 2009)



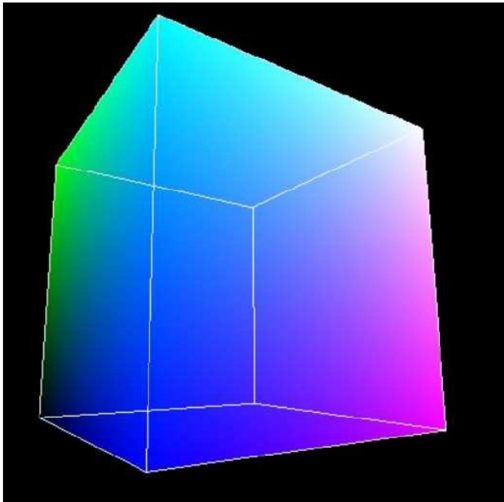
Obrázek 2.15: Grafické znázornění zobrazovací rovnice. Převzato a upraveno z (Justin, et al., 2009).

Na rozdíl od obecného vyjádření zobrazovací rovnice však sledování paprsku pracuje nad diskrétními daty, konkrétně jak nad pravidelnou mřížkou voxelů, tak i nestrukturovanou mřížkou (obecně množina dat - *Data set*). Mezi oko (*Eye*) a množinu dat vložíme promítací plát (*Image plane*), což je v důsledku výsledný 2D obraz, který bude zobrazovat náš výsledek. Barvu každého pixelu promítacího plátu spočteme poté tak, že vyšleme paprsek z oka skrz požadovaný pixel. Pokud mineme množinu dat, zůstane pixel beze změny (tzn. bílý, černý, průhledný apod.). Naopak jestliže daný paprsek projde objemem, naakumuluje barevnou informaci ze vzorků dat, kterými prošel (viz Obrázek 2.16). Počet paprsků, které z oka vysíláme, je tedy rovný počtu pixelů promítacího plátu.

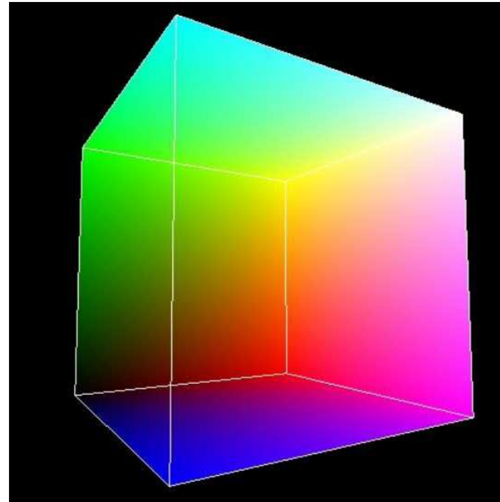


Obrázek 2.16: Ukázka ray-castingu neboli vrhání paprsku objemem. Paprsek spočítá pixel na promítacím plátu pomocí hodnot, které nasbírá při průchodu množinou dat. Převzato a upraveno z (Weiskopf, 2004).

Po objasnění fundamentálních pojmů se vraťme k metodě *GPU based – Direct volume rendering*. Tato metoda využívá principů uvedených v (Weiskopf, 2004) a celý výpočet je prováděn na GPU (popis postupu např. také v (Vaněček, et al., 2008)). Nejprve je nutné namapovat množinu dat do krychle o rozměrech $1 \times 1 \times 1$ (pokud množina dat není krychle, ale kvádr o různé délce stran, zachováme poměr stran původní množiny dat s tím, že nejdelší strana má rozměr 1) umístěné v počátku. Poté použijeme jako barvu vrcholů krychle námi normované souřadnice a vykreslíme nejprve přední, a poté zadní strany krychle. Oba tyto výstupy si uložíme do textury s tím, že první nám udává pozici vniknutí paprsku (*rayIn*) do objemu a druhá pozici, kde paprsek pro daný pixel objem opouští (*rayOut*). Jedná se tedy vlastně pouze o důmyslné využití barev, které nám v intervalu $\langle 0; 1 \rangle$ jednoznačně určují souřadnice (viz Obrázek 2.17 a Obrázek 2.18).



Obrázek 2.17: Čelní (přivrácené) strany jednotkové krychle, kde barva udává kudy paprsek do objemu vstupuje (*rayIn*). (Purchart, 2009)



Obrázek 2.18: Zadní strany jednotkové krychle, kde barva udává kudy paprsek z objemu vystupuje (*rayOut*). (Purchart, 2009)

Známe-li pozici vstupu i výstupu paprsku, spočítáme jednoduše směr ($direction$), kterým paprsek pro daný pixel objemem prochází (viz Vzorec 2.6).

$$direction = rayOut - rayIn$$

Vzorec 2.6: Směr průchodu paprsku pro daný pixel. RayOut a rayIn jsou pozice, prezentované pomocí barevné informace na intervalu $\langle 0; 1 \rangle$. (Purchart, 2009)

Tímto směrem poté postupujeme po ekvidistantních krocích, při kterých akumulujeme do výsledného vektoru (*dst*) barvu (*rgb*) a průhlednost (*a*) ze vzorku, na kterém se právě nacházíme (*src*). Akumulaci se provádí sofistikovaněji - viz Vzorec 2.7.

$$dst.rgb = dst.rgb + (1 - dst.a) \cdot src.a \cdot src.rgb$$
$$dst.a = dst.a + (1 - dst.a) \cdot src.a$$

Vzorec 2.7: Výpočet barvy a průhlednosti. Akumulace do výsledného vektoru v každém kroku průchodu množinou dat. (Purchart, 2009)

Pokud je během akumulace výsledný vektor neprůhledný ($dst.a \geq 1$), nebo pokud jsme již opustili množinu vzorků, akumulaci ukončíme a výsledek

zobrazíme na příslušný pixel (V. Purchart dále ve své práci výsledný vektor upravuje pomocí tzv. převodní funkce, která ovlivňuje barevnost a průhlednost výsledného vektoru, tato část je však již mimo spektrum využitelnosti pro vykreslování mraků).

2.5 Sumarizace poznatků

Přistupme nyní nejprve k zhodnocení vyzkoušených her. Bohužel je dopředu třeba říci, že ani jedna neodpovídá autorově představě o ideálním vzhledu mraků. Najdou se však i dílčí řešení, kterými se inspirovat lze. Pojďme si tedy postupně projít tabulku vlastností.

Stavba mraku z řady různorodě natočených billboardů je technologie, kterou používat nebudeme. Naše technika bude využívat spíše sofistikovanějších metod z oblasti vykreslování objemových dat (viz 2.4.3 Další užitečné metody) a výsledný obraz bude nanesen na jeden až dva billboardy otočené ke kameře. Nepůjde však v pravém slova smyslu o billboardy, jelikož se bude dynamicky měnit jejich textura (tzv. imposter viz 2.4.2 Zobrazování).

Vzhled mraku *cumulus* je, dle subjektivního názoru autora, nejpovedenější ve hře Tom Clancy's HAWX. Sice zřejmě není tak realistický, ale přesně splňuje očekávání hráče. Velké barevné rozdíly, stínování, hezký tvar. Tomuto vzhledu se budeme částečně tedy snažit přiblížit.

Osvětlení a stínování mraku budeme dělat vlastním postupem, který kombinuje předpočítané detailní vystínování s osvětlením v reálném čase a využívá předností reprezentace mraku voxely (viz kapitola 2.4.1 Modelování mraku), kterou budeme – na rozdíl od všech výše zmíněných her – používat.

Průlet mrakem není udělán dobře v žádné hře. Nejhoršího vizuálního dojmu dosahuje hra Air Conflicts, kde se billboardy většinou uhýbají před kamerou. Ve zbývajících hrách námi ovládané letadlo – přestože ho vidíme před kamerou – nikdy nevletí do mraku (ve smyslu, že ho mrak nezahaluje), což velmi snižuje realistický dojem. Jediné, čím se bude inspirovat, je zprůhledňování billboardu, ke kterému se blížíme.

Jako způsob mizení se jeví nejlepší zmenšovaní mraku až k horizontu (resp. hranici, kdy ho již tvoří jen pár pixelů), které vypadá velmi realisticky.

Pohyb mraku nemá žádná hra, popřípadě je velmi těžko pozorovatelný. My zavedeme vítr, který bude posouvat se všemi mraky shodně (čímž zajistíme stále stejné relativní vzdálenosti mraků a zamezíme tak jejich prolínání).

Jako rozmístění mraků po obloze zvolíme náhodné rozmístění (stejně si poté ve hře chce grafik či *level designer*[7] většinou umísťovat mraky dle sebe).

Pokud tedy máme určit hru, které se budou naše mraky nejvíce podobat, je to hra Tom Clancy's HAWX. Bude to však i tak podoba velmi vzdálená, jelikož náš přístup se v řadě důležitých věcí liší.

Potom, co jsme shrnuli vlastnosti vypozorované, pojďme analyzovat a sumarizovat, které postupy a dílčí řešení využijeme ze sekce vlastností vyčtených. Co se modelování mraků týká, můžeme rozdělit metody na modelování založené na fyzikální simulaci vzniku mraku na jedné a modelování založené na napodobení vizuálního vzhledu na druhé straně. První postup nám skýtá jako hlavní výhodu možnost animování změn v mraku, přeměny jeho tvaru apod., druhý poté představuje jednodušší implementaci a především daleko menší potřebný výpočetní čas. Z podstaty našeho zadání tedy zvolíme druhou možnost s tím, že budeme využívat voxelové reprezentace dat. Uložíme je do pravidelné mřížky – vytvoříme krychli, kterou rozdělíme ve všech osách na shodný počet dílků. Tato data poté zkusíme vyplnit pomocí Perlinova šumu a celé oříznout do kopulovitého tvaru s plochou základnou. Zde budeme provádět ještě další úpravy přímo při konstrukci mraku.

Vykreslovací techniku založíme na vrhání paprsku dle ořístupu D. Weiskopfa, který je určen pro vykreslování na GPU a je dostatečně obecný, aby po úpravě mohl zobrazovat i mraky. Podobně jako M. Harris uděláme

[7] Level designer je člen vývojářského týmu, který tvoří daný level neboli úroveň hry. Z dostupných grafických objektů a herních prvků většinou poskládává úroveň tak, aby byla graficky atraktivní a zároveň dobře hratelná.

dvojstupňové vykreslování, první náročnější v *pre-processingu*[8], druhé poté *real-time*. Osvětlovací model však upravíme tak, abychom mohli v reálném čase měnit polohu světla. To bude sice za cenu zjednodušení modelu, ale algoritmus bude mít širší využití (např. pro hry, kde hráč stráví v jednom levelu několik hodin herního času, a tudíž se zákonitě bude měnit poloha slunce). Po vzoru M. Harrise použijeme dále namapování výsledného obrazu na imposter i použití dvou imposterů při průletu skrze mrak.

[8] Anglickým slovem pre-processing (někdy též offline) se označuje fáze, která probíhá pouze jednou a to před začátkem samotného vykreslování. Do této fáze se umisťují náročné výpočty, které není třeba počítat v reálném čase.

3 Realizační část

V této kapitole si popíšeme postup vývoje mraků a důležité části si probereme „krok za krokem“.

Realizace by se dala rozdělit na dvě hlavní části. První z nich je tvorba podkladů, tzn. textur ve 2D grafickém editoru a modelů ve 3D grafickém editoru. Druhá je potom samotná programová realizace a věci s ní spojené. V té si nebudeme popisovat celý program. To by bylo zcela jistě na úkor zajímavosti i obecných záměrů této práce. Budeme se soustředit na fragmenty, které jsou důležité pro tvorbu a především vykreslování mraků. Obzvláště se poté zaměříme na složité části a také na algoritmy, které nebyly inspirovány předešlými pracemi, ale vznikly originálně pro tuto diplomovou práci.

3.1 Tvorba podkladů

Jako podklady bylo třeba vytvořit 3D model letadla, nafotit řadu mraků a vytvořit textury slunce a oblohy.

Jako 3D editor pro tvorbu letadla bylo využito *Autodesk 3ds Max 2009* (dále 3ds Max). Tento výkonný grafický editor byl zvolen proto, že je to stále nejvíce používaný editor tvůrci počítačových her, je pro tuto činnost přímo stavěn a modelování objektů je v něm velmi komfortní. Dále je také možné z něj bez problémů exportovat jakýkoli model s texturou do formátu *DirectX .X* a ten snadno načíst v *XNA Game Studiu 3.1*, které používáme (více v 3.2 Programová realizace). Mimoto má autor práce s tímto grafickým editorem dlouholeté zkušenosti.

Jako fotoaparát autor používal nejprve digitální fotoaparát *Olympus FE150*, který ovšem nenabízel v některých ohledech dostatečnou kvalitu, a proto byl nahrazen fotoaparátem *Casio Exilim EX-Z280*. Ten nabízí lepší optický zoom, stabilizaci obrazu, která je při focení mraků při přiblížení velmi užitečná a poskytuje rozlišení 12.1 mega pixelů, což bylo potřebné s výhledem na případnou

další úpravu fotografií. Fotografie se však nakonec jako textury nevyužily, avšak posloužily jako ilustrační materiál pro tento text.

Textury byly tvořeny v 2D editoru *Adobe Photoshop CS2*, který je pro upravování bitmapových obrázků (textur, fotografií, apod.) jedním z nejlepších a díky svým dlouholetým zkušenostem s tímto editorem je autor schopen dosáhnout adekvátních výsledků.

3.1.1 Model letadla

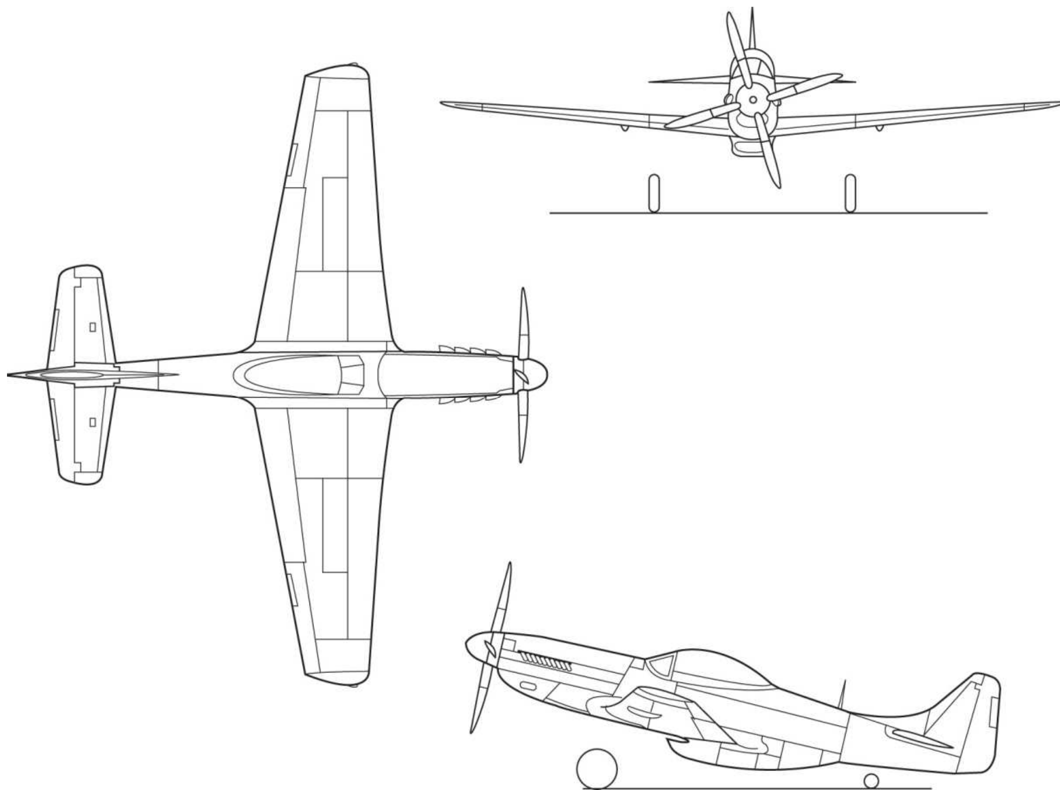
Jak již víme, model letadla byl tvořen v 3ds Max. Jedná se o tzv. *low-poly model*, neboli model, který má nízký počet polygonů[9] (obecně většinou se udává počet trojúhelníků), aby nezpomaloval vykreslování scény. Před cca 10 lety znamenal *low-poly* model s „pár set" trojúhelníky. Dnes je to již v řádech tisíců. Pro naše účely byl stanoven ideální počet okolo dvou tisíc. Další odlišností *low-poly* modelu je i samotné modelování. Nepoužíváme NURBS křivky apod., ale celý model většinou „vytahujeme" z krychle, nebo plochy (viz dále), jelikož tak máme právě kontrolu nad počtem trojúhelníků. Nejedná se sice o část, která je pro tuto práci esenciální, ale jelikož model vznikal pouze pro tento program a jeho tvorba je zajímavá, pojďme si ji v krátkosti a s doplňujícími obrázky popsat.

Nejprve je tedy nutné rozhodnout se, jaký model budeme tvořit a k němu poté vyhledat, popřípadě si nafotit či nakreslit čelní, boční a vrchní pohled (nárys, bokorys, půdorys). My jsme se rozhodli pro známý americký letoun *P-51D Mustang* a našli si jeho skicu (Obrázek 3.1).

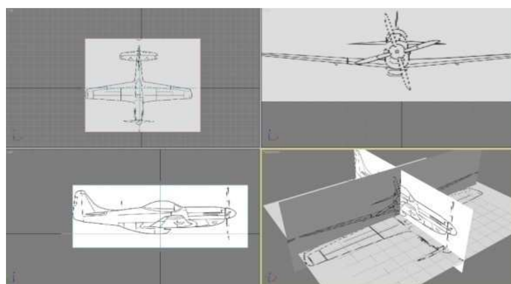
V 3ds Max si vytvoříme 3 navzájem kolmé osově zarovnané plochy (*planes*), na které si naneseme skicu vždy z příslušného pohledu (Obrázek 3.2). Dále si musíme vytvořit počátek – vybereme si část trupu, odkud začneme modelovat a položíme na ni plochu (*plane*). Na Obrázek 3.3 vidíme zvolený počátek v přední části letadla. Plochu poté tvarově upravíme, aby odpovídala skice. Toho dosahujeme pomocí vytahování hran (*edges*) v *Editable poly* módu.

[9] *Polygon*, česky též n-úhelník, nebo mnohoúhelník, je obecně rovina ohraničená třemi a více úsečkami.

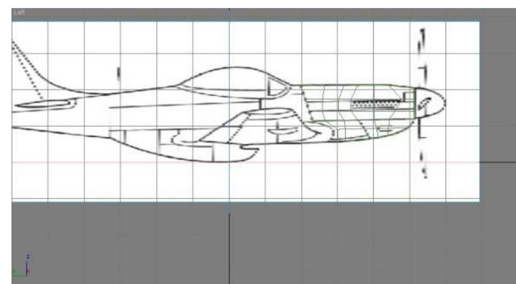
Poté ještě upravíme nově vzniklou plochu přesouváním jejích okrajových bodů (*vertex*). Tímto postupem vymodelujeme převážnou část letadla. Vytažená hrana nám vždy vytvoří nový polygon, u kterého upravíme *vertexy*. Je dobré vytahovat hrany tak, aby šel později model lehce rozdělit či zaoblit.



Obrázek 3.1: Jeden z nejznámějších stíhacích letounů 2. světové války 51D Mustang - nárys, bokorys, půdorys. (Owly, 2005)



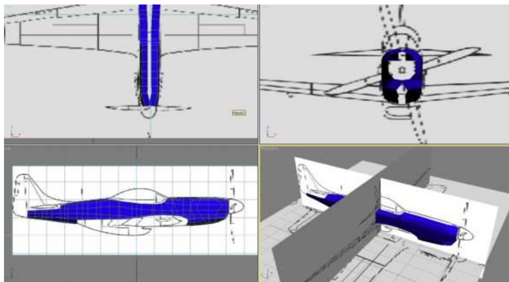
Obrázek 3.2: Modelování letadla, krok jedna. Vytvoření 3 osově zarovnaných ploch a nanesení skici.



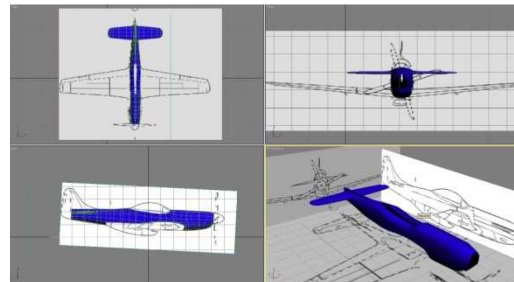
Obrázek 3.3: Modelování letadla, krok dva. Položení počáteční plochy na motor letadla a přizpůsobení jeho tvaru.

Jelikož je model symetrický, budeme modelovat pouze jeho levou polovinu (levou z čelního pohledu) a modifikátorem *Symetry* si automaticky

necháme zobrazovat i jeho pravou část. Na Obrázek 3.4 vidíme již vymodelovanou základní část trupu letadla. Dále budeme tvořit ocasní křídlo – musíme si tedy připravit uzavřený otvor v trupu ve tvaru křídla z bokorysu. Tu poté jednoduše „vytáhneme“ z hrany na okraji otvoru. Výsledek je vidět na Obrázek 3.5, kde je také model již zaoblený. Toho se dosáhne rozdáním zaoblovacích skupin (*Smoothing groups*).

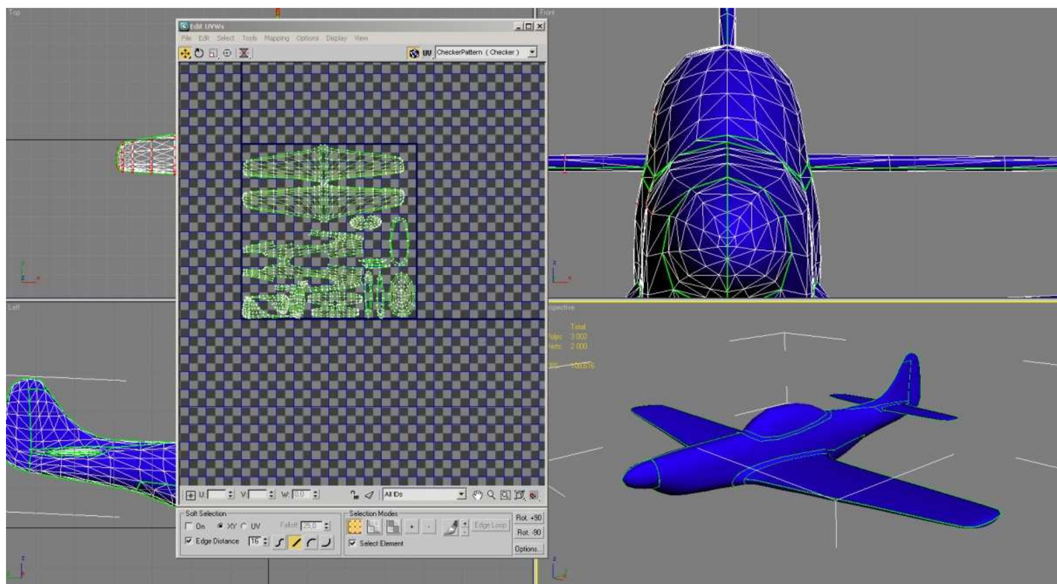


Obrázek 3.4: Modelování letadla, krok tři. Vymodelování základní části levé poloviny trupu a jeho zrcadlení modifikátorem Symetry.



Obrázek 3.5: Modelování letadla, krok čtyři. Přidání zadního křídla.

Výše popsanými technikami vymodelujeme zbylou část levé poloviny trupu. Pokud jsme s výsledkem naší práce spokojeni, sloučíme modifikátor *Symetry* s modifikátorem *Editable Poly* – tím nám vznikne celý model letadla a již nepůjde upravovat symetricky jedna část.



Obrázek 3.6: Modelování letadla, krok pět. Finální model, na kterém vidíme jednotlivé trojúhelníky. V levé části je poté modifikátorem UVW Unwrap ukázáno mapování textury na náš model.

Na daný model aplikujeme modifikátor *Editable Mesh*, kterým si můžeme zobrazit počet trojúhelníků. Náš model jich má 1996, tudíž jsme se perfektně vešli do našeho záměru. Poslední modifikátor, který aplikujeme, bude *UVW Unwrap*. Ten nám otevře samostatné okno, ve kterém umožní rozložit model do roviny (resp. jeho texturovací souřadnice). Poté na něj můžeme přesně namapovat 2D texturu, viz Obrázek 3.6.

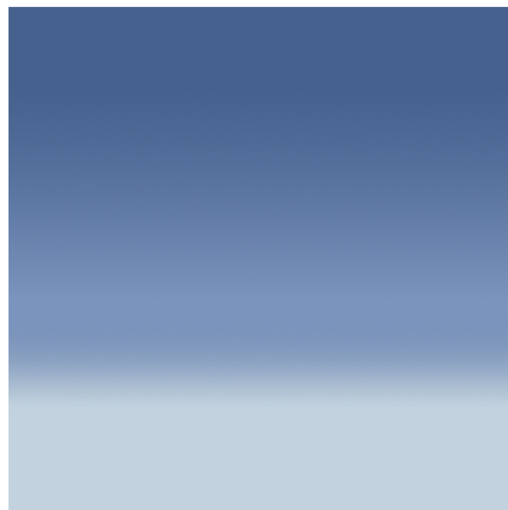
3.1.2 SkyBox a slunce

Důležitým prvkem velmi se podílejícím na výsledném vizuálním dojmu, je pozadí mraku a celková scéna. Oblak zobrazený na jednobarevné modré ploše působí nepřirozeně a jaksi vytrženě z kontextu. Proto bylo třeba na pozadí umístit tzv. *SkyBox*, neboli otexturovanou krychli, která obaluje celou scénu. Jelikož tvorba takové krychle není zcela triviální, popíšeme si zde celý postup.

Nejprve je tedy nutné vytvořit speciální texturu pro každou stranu krychle. Právě správná textura je na celém procesu výroby S*kyBoxu* asi to nejsložitější a také velice ovlivní vzhled celé aplikace. Ukažme si tedy podrobně jak takovou texturu jedné strany (viz Obrázek 3.11) vytvořit.

V programu *Adobe Photoshop* vytvoříme nový soubor o velikosti 512 × 512 pixelů s průhledným pozadím. Poté vybereme *Nástroj přechod*, zvolíme lineární přechod mezi neprůhlednými barvami R:70, G:97, B:142 a R:146, G:171, B:210 a nakreslíme jej přes celou plochu (Obrázek 3.7).

Obrázek 3.7: Tvorba textury SkyBoxu. Krok jedna, vyplnění oblohy přechodem.



Obrázek 3.8: Tvorba textury SkyBoxu. Krok dva, přidání oparu.

U tohoto nástroje ještě zůstaneme. První barvu ponecháme, pouze jí krytí nastavíme na 0% a druhou barvu změníme na R:194 G:210 B:222 a nakreslíme druhý přechod ve spodní části (Obrázek 3.8), který nám bude reprezentovat opar. Poté vybereme *Nástroj rozmazání*, v nastavení *Stopa* ji nastavíme *Hlavní průměr:* 60 ob a *Tvrdost:* 0% a dále nastavíme *Síla:* 25%. Nyní rozmažeme opar (viz Obrázek 3.9), tak aby vznikly nepravidelnosti. Pozor - opar bude základem pro textury všech stran krychle (pouze se na něm vyrobí jiné nepravidelnosti), nerozmazáváme tedy části oparu těsně u krajů, jinak textura nebude navazovat.

Dále u stejného nástroje změníme tyto hodnoty: *Hlavní průměr:* 20 ob a *Síla:* 40% a rozmažeme vršky nepravidelností (Obrázek 3.10). Jelikož se textura nanese na krychli, která bude obklopovat hráče, vypadá nepřirozeně, pokud necháme texturu takto rovnou. Potřebujeme ji zaoblit a dosáhnout tak efektu zaoblení zeměkoule. Zvolíme tedy *Filtr - Deformace - Zaoblení* a nastavíme hodnotu *Míra:* -100%. Výsledek vidíme na Obrázek 3.11.

Obrázek 3.9: Tvorba textury SkyBoxu. Krok tři, nepravidelný opar.

Obrázek 3.10: Tvorba textury SkyBoxu. Krok čtyři, rozmazání vršků oparu.

Obrázek 3.11: Tvorba textury SkyBoxu. Krok pět, zaoblení.

Obrázek 3.12: Tvorba textury SkyBoxu. Krok šest, kontrola návazností okrajů.

Tímto jsme vytvořili texturu, kterou lze použít pro všechny čtyři strany nebo ji můžeme použít pouze jako např. texturu pro přední stranu a vyrobit stejným postupem (až k oparu použijeme stejný obrázek a začneme až od jeho rozmazávání) další 3 textury (zadní, levá, pravá). To má za výhodu přirozenější různorodější vzhled. Pokud bychom ale přeci jen chtěli využít čtyřikrát tuto texturu, ukážeme si fintu, jak se přesvědčit, že na sebe bude navazovat. Mělo by

se toho sice dosáhnout tím, že jsme neupravovali boky oparu, ale pro jistotu zvolíme *Filtr - Jiné - Posun* a *Vodorovně* posuneme obrázek o polovinu jeho vzdálenosti (256), *Svisle:* 0 a vybereme možnost *Přetočit dokola*. Okraje se poté přesunou doprostřed (viz Obrázek 3.12) a vidíme, zda navazují. Pokud ne, vybereme opět *Nástroj rozmazání* a ostrý přechod rozmažeme. Poté opět stejným postupem posuneme o dalších 256, a tím vrátíme texturu do původního tvaru. Nyní máme 100% jistotu, že bude navazovat (pokud používáme 4 textury, je třeba si položit nakonec všechny 4 vedle sebe a opět se přesvědčit, zda navazují).

Poté co si nakreslíme všech šest textur, vytvoříme soubor *textures.txt*, do kterého názvy textur napíšeme (viz Výpis 3.1).

```
right.bmp
left.bmp
up.bmp
down.bmp
front.bmp
back.bmp
```

Výpis 3.1: Obsah textového souboru textures.txt. (Rstralberg, 2008)

Nyní si vytvoříme ve stejném adresáři soubor *create_sky.cmd* a vložíme do něj text z Výpis 3.2. Dále je nutné z (Legacy DDS Utilities, 2007) stáhnout a nainstalovat program *DDS Utilities Installer*. Poté spustíme soubor *create_sky.cmd*, který nám vytvoří finální *sky.dds*, což je *MIP Map* textura, jenž obsahuje všech šest textur pro otexturování celé krychle – tu budeme načítat pomocí *TextureCube Class* v *XNA Game Studiu 3.1* (viz (TextureCube Class, 2010)). Více o implementaci do programu v části 3.2.9 SkyBox, slunce a vítr.

```
if exist sky.dds del sky.dds
"C:\Program Files\NVIDIA Corporation\DDS Utilities\nvdxt.exe" -
cubeMap -list "textures.txt" -output "sky.dds"
```

Výpis 3.2: Obsah textového souboru create_sky.cmd. (Rstralberg, 2008)

Textura slunce je vidět na Obrázek 3.13. Ve skutečnosti nemá modré pozadí a je tvořena pouze bílými přechody, avšak pro tisk bylo pozadí přidáno. Zkráceně tvorba textury probíhá tak, že ve *Photoshopu* vytvoříme nový soubor 512x512 pixelů s průhledným pozadím. Vybereme *Nástroj přechod* a nastavíme přechod z bílé s *krytím* 0% do bílé barvy s *krytím* 100%, zvolíme *Kruhový přechod* a vytvoříme přechod se středem ve středu obrázku a poloměrem 256 ob. Poté vytvoříme další menší přechod uvnitř a celý efekt doplníme několika bílými čárami vedoucími ze středu do různých směrů (*Nástroj čára*, *tloušťka* 1 ob*, Režim: Závoj, Krytí:* 6% a *se zvoleným Vyhlazením*).

Obrázek 3.13: Textura slunce. V reálu není v textuře pozadí modré, ale průhledné. To by ale jaksi nebylo na papíře vidět.

3.2 Programová realizace

Z předchozích zkušeností z oblasti vývoje počítačových her zvolil autor pro tuto práci jako hlavní vývojářský nástroj *XNA Game Studio*. Pro tuto volbu hovoří mimo skutečnosti, že má s tímto produktem od společnosti Microsoft (dále MS) již dlouholeté zkušenosti, především fakt, že se jedná o jedinečný prostředek určený cíleně pro tvůrce 3D počítačových her. Navazuje na tzv. *Managed DirectX*, což bylo *DirectX* (platforma pro vývoj hardwarově akcelerovaných aplikací) ve spojení s jazykem C# a je jakýmsi velmi odlehčeným a obecným herním *enginem*. Umožňuje nám využít nástrojů, předpřipravených konstrukcí, matematických funkcí apod. Oficiálně podporuje pouze námi využívaný jazyk C# - jazyk od MS, při jehož vývoji se brala inspirace z C++ a Javy. Jazyk C++ je sice stále o něco rychlejší, avšak C# nabízí větší efektivitu a rychlost vývoje, je robustnější a odpovídá moderním trendům. XNA je tedy ideální pro naše účely, jelikož nám ušetří čas potřebný na programování základní funkčnosti a budeme se moci rovnou soustředit na tvorbu a vykreslování mraků.

Samotné vykreslování je poté programováno především přímo na grafické kartě pomocí tzv. *High Level Shader Language* (dále HLSL). HLSL je proprietární vysokoúrovňový jazyk vyvíjený společností Microsoft, určený pro programování GPU. Při programování v XNA (a v podstatě tedy nepřímo v *DirectX*) se jedná o nejvíce využívaný a podporovaný standart (podobně jako v *OpenGL* se využívá GLSL).

Organizačně je poté třeba upozornit, že kódy ve výpisech jsou hojně komentovány, a proto je v samotném textu vysvětlíme obecně, ale nebudeme procházet řádek po řádku, jelikož by se jednalo o duplikování informace.

3.2.1 Programovatelná pipeline

Jak jsme uvedli v úvodu této kapitoly, je vykreslování mraku převážně realizované pomocí jazyka HLSL na grafické kartě. Jelikož bude tvořit toto programování základ naší metody, pojďme se nejprve na tento jazyk podrobněji

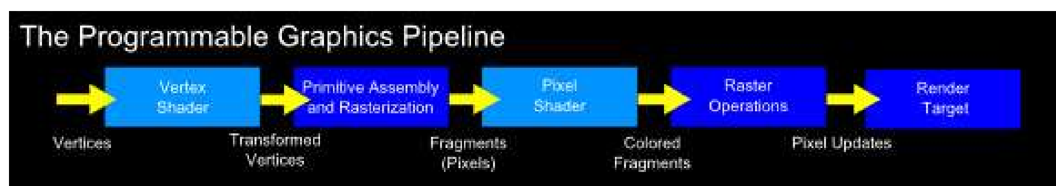
podívat. Pomocí jazyka HLSL se vytváří tzv. *shader effects* – soubory s příponou fx, ve kterých programujeme grafickou *pipeline*. Grafická *pipeline* je v podstatě posloupnost procesů na grafické kartě, které mají na starost vykreslení množiny vrcholů (3D objekt) se správnými transformacemi, texturami, nasvícením atp. Pomocí HLSL lze ovlivnit 3 části tohoto procesu: *Vertex shader*, *Geometry shader* a *Pixel shader*.

Vertex shader (dále VS) se vykonává pro každý vrchol námi vykreslovaného objektu a využívá se především pro transformace (typicky převedení vrcholu z jeho vnitřního souřadného systému do obecného) a výpočet texturovacích souřadnic. Z určitého počtu vrcholů (typicky tří), které projdou VS, se poté interpoluje oblast mezi nimi (proces nazýváme rasterizace) a dělením této oblasti vzniknou nejmenší možné části – tzv. pixely.

Pixel shader (dále PS) se vykonává pro každý pixel a stará se o výpočet barvy, která již bude pro daný pixel zobrazena na monitoru.

Geometry shader (dále GS) stojí poté mezi VS a PS a vykonává se pro každé primitivum (materiál, hrana, trojúhelník apod.). GS se dá využít na grafickém hardware podporující *Direct3D 10* a výše a my ho v našem programu využívat nebudeme.

Celý proces programovatelné *pipeline* a využití VS a PS vidíme na Obrázek 3.14.



Obrázek 3.14: Programovatelná pipeline. Vstupem Vertex Shaderu jsou vrcholy (Vertices), výstupem jsou transformované vrcholy, které se poté rasterizují a výsledné pixely (Pixels) putují do Pixel Shaderu, kde se pro každý spočítá jeho barva. Světle modře jsou na obrázku vyznačeny části pipeline, které budeme přímo programovat. (Whitaker, 2009)

3.2.2 Generování mraku

Hlavní gró této práce spočívá v rychlosti vykreslování, a tudíž je většina pozornosti věnována vykreslovací části. Ta však samozřejmě musí mít co vykreslovat a nelze tedy opomenout i samotné generování dat mraku. Z pohledu nižší výpočetní náročnosti (hráči mohou čekat před začátkem hry v řádu sekund, rozhodně však ne v řádu desítek minut) jsme se rozhodli generovat data bez fyzikální simulace procesů – budeme se pouze snažit o co nejvěrnější zachycení podoby mraku *cumulus*. Ten má v podstatě většinou tvar polokoule s plochou základnou a roztřesenými okraji a tento tvar se pokusíme nasimulovat.

Podívejme se tedy na Výpis 3.3. První podstatná věc je definice 3D textury. Tu budeme předávat později GPU a z ní budeme vykreslovat data. Co to vlastně 3D textura je? Nejjednodušeji si ji lze představit jako *n*-rozměrné pole klasických 2D textur, přičemž právě *n* je hloubka – tedy třetí rozměr.

Po definici proměnných přichází na řadu trojnásobný cyklus. Ten je v kódu především pro přehlednost, jelikož všechna data ukládáme do jednorozměrného pole *data*. Souřadnice *x, y* a *z* nám udávají, kde se reálně v krychli nacházíme. Jedná se o namapování krychle o obecné délce hrany *size* na krychli o délce hrany 1. Navíc se vždy v každé souřadnici ještě posouváme o polovinu délky hrany jednoho dílku, čímž zajišťujeme, že se nacházíme vždy ve středu dílku. Poté si zadefinujeme kouli s poloměrem $\frac{1}{2}$, což odpovídá půlce hrany krychle. Data, která jsou mimo kouli, obarvíme na černo (*colValue = 0*), stejně jako data, která se nacházejí pod $y > 0.4$. Tím zajistíme uříznutí spodní části koule a zploštění základny. Abychom zajistili porušení hladkého povrchu koule, je ještě v té samé podmínce přidán šum *((float)(rand.NextDouble()) * 0.1)*, který odečítáme od povrchu kulovité části plochy (tzn. číslo v intervalu $\langle 0; 1 \rangle$) a o něco menší šum, který odečítáme od podstavy.

Pokud tedy voxel splní podmínky a leží uvnitř mraku, uložíme do všech jeho složek hodnotu 255, což odpovídá bílé barvě. Ostatní voxely budou mít ve všech složkách černou. Volitelnou možností je použít na alfa kanál Perlinův šum,

který vytvoří náhodný šum uvnitř celého mraku (ne tedy pouze na jeho obvodu) a z mraku *cumulus* vytvoří mrak typu *altocumulus* (lidově beránky).

```
w = h = d = size; // Šířka, výška i hloubka rovné velikosti mraku.
data = new byte[w * h * d * 4]; // Pole dat o mraku (rgba).
int offset = 0;   // Pozice v poli.
a = b = c = 1 / 2f;     // Koeficienty pro tvorbu koule.
r = (1f / 2.5f);  // Poloměr koule.
float x, y, z, colValue; // Souřadnice v krychli a barva/alfa.

// Vyplnění kulového objemu bílou (a volitlně perlin noise alfy).
for (int i = 0; i < w; i++)
    for (int j = 0; j < h; j++)
        for (int k = 0; k < d; k++)
        {
            // Přemapování souřadnic na interval <0;1>.
            //(i je 0..size / size = 0..1) + (1/2size = 1/128)
            x = (float)i / size + 1f / (size * 2);
            y = (float)j / size + 1f / (size * 2);
            z = (float)k / size + 1f / (size * 2);

            // Uděláme kouli. Uvnitř bílá barva, vně černá.
            Zaneseny chyby na okrajích a plochá základna.
            float globe = (x - a)² + (y - b)² + (z - c)²;
            if (globe < (r² - (random * 0.1)) && (y > 0.4 -
            (random * 0.04)))
                colValue = 255;
            else
                colValue = 0;

            // Do dat přidáme r,g,b + (perlin noise) alfu.
            data[offset++] = (byte)colValue;
            data[offset++] = (byte)colValue;
            data[offset++] = (byte)colValue;
            if (altocumulus)
                 data[offset++] = (byte)((perlinNoise.cloudAbs(x,
                 y, z, 5, random, random)) * colValue);
            else
                data[offset++] = (byte)colValue;
        }
```

Výpis 3.3: Příprava dat mraku. Vytváříme polokouli, kterou po obvodu narušíme náhodným šumem. To je v podstatě základní tvar mraku cumulus. (Pro větší přehlednost klasický zápis mocnin.)

Tento postup nám tedy generuje pouze bílý mrak – viz Obrázek 3.15 – tedy základní tvar, který ovšem potřebuje ještě osvětlit atd., aby vypadal realisticky.

Obrázek 3.15: Neosvětlený mrak, který pouze ukazuje základní tvar, vygenerovaný naším programem.

3.2.3 Vykreslování mraku

Celá následující část třetí kapitoly se zabývá vykreslováním mraku. V této podkapitole si nejprve souhrnně vysvětlíme celý proces a v dalších se budeme zabývat jeho obtížnými a zajímavými částmi.

Jaký je tedy celkový průběh vykreslení jednoho mraku? Body 1 – 4 probíhají na CPU, další již využívají i GPU.

1) Vytvoříme 3D texturu mraku a naplníme ji odpovídajícími daty (viz 3.2.2 Generování mraku) – tzn. do barevné informace uložíme barvu jednotlivých voxelů vzhledem k poloze slunce a tomu, jestli tvoří či netvoří mrak a do alfa kanálu informace o průhlednosti.
2) Zjistíme, zdali se poloha mraku vůči pozorovateli změnila o více než nějaký malý úhel a pokud ano, zavoláme vykreslování mraku.
3) Vytvoříme si imposter z aktuálního pohledu tak, aby zakrýval celý mrak, byl k němu co nejblíže a samozřejmě aby svým normálovým vektorem směřoval ke kameře (viz 3.2.6 Impostery).

4) Pokud svírá normálový vektor imposteru s vektorem kamery takový úhel, že je mrak viditelný (tzn., nachází se v pohledovém jehlanu – viz 3.2.7 Zjišťování viditelnosti mraku), vypočítáme, zdali je dostatečně velký na obrazovce (viz 3.2.11 Řízení úrovně detailů).
5) Pokud je mrak dostatečně velký (tzn. bude viditelný), začneme ho v adekvátní velikosti (velikost promítacího plátu, určená jako nejbližší mocnina dvou) vykreslovat na GPU pomocí upravené *ray-casting* metody (viz 3.2.4 Vykreslování mraku).
6) V té se nejprve vypočítá *real-time* osvětlení a poté vykreslí samotný objem.
7) Výslednou texturu (která má velikost hrany rovnou 2^n, maximálně však 64) naneseme na obdélník, který je přichycen přímo na obrazovku, a pomocí GPU texturu roztáhneme. Výsledek nevykreslíme, ale uložíme do textury.
8) Texturu naneseme na výše zmiňovaný imposter a opět na GPU roztáhneme a vyhladíme – tím vlastně necháme mrak projít 2x hardwarově akcelerovaným vyhlazováním a získáme z relativně malé textury velkou texturu s hladkými rozmazanými hranami (což v případě mraku chceme).

3.2.4 Vykreslování mraku na GPU

V podkapitole 3.2.1 Programovatelná pipeline jsme poměrně obsáhle seznamovali čtenáře s pojmem GPU a programovatelné *pipeline*. Důvod byl prozaický – na GPU vykonáváme přes 600 řádek HLSL kódu, hlavní *shader* má téměř 500 řádek a obsahuje nejdůležitější vykreslovací a osvětlovací algoritmus. Proto je třeba s ním čtenáře podrobně seznámit.

Náš program obsahuje 4 *shader effect* soubory – *Skybox.fx*, *Sun.fx*, *Box.fx* a *Cloud.fx*. První dva jmenované si vysvětlíme v 3.2.9 SkyBox, slunce a vítr, zde se pojďme podívat na poslední dva jmenované.

Box.fx je jednoduchý *shader*, který pouze vykresluje souřadnice krychle normované na interval ⟨0; 1⟩ jako barvu. Na VS přijde pozice vrcholu a ten ji bez aplikace jakýchkoli transformací předá PS jako texturovací souřadnici *wPos*. PS pak tuto souřadnici vykreslí jako barvu (viz Výpis 3.4).

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
      //Vykreslení souřadnice jako barvy.
      return float4(input.wPos.xyz, 1);
}
```

Výpis 3.4: Pixel shader vykreslující souřadnici krychle o velikosti hrany 1 jako barvu.

Cloud.fx je poté daleko sofistikovanější shader a obsahuje mj. VS metody *VertexShaderFunctionHR*, *VertexShaderFunctionLR* a PS metody *PixelShaderFunctionHR*, *PixelShaderFunctionLR*. Metody s příponou *HR* (*high-resolution*) jsou určeny pro vykreslení textury, kterou jsme nanesli na obrazovku a uložili na imposter. V podstatě jediná funkčnost *shaderu* je, že nastavíme *sampler* pro tuto texturu. Ten je propojen s texturou a říká, jakým způsobem bude textura zpracována. My nastavíme *MipFilter*, *MinFilter* a *MagFilter* na hodnotu *LINEAR*, což nám zajistí rozmazání a vyhlazení textury.

Metody s příponou *LR* (*low-resolution*) zajišťují *ray-casting*, osvětlení a zprůhledňování mraku. *VertexShaderFunctionLR* pouze předává pozice vrcholů PS. V *PixelShaderFunctionLR* se vykonává *ray-casting*, osvětlení v reálném čase (to nalezneme detailně popsané v 3.2.8 Osvětlení mraku) a zprůhledňování (podrobně v 3.2.10 Alfa průhlednost). *Ray-casting* vychází z metody popsané v 2.4.3 Další užitečné metody a jelikož je to nejdůležitější část této práce, uvádíme ho kompletní v kapitole Příloha B - Hlavní pixel shader s očíslovanými řádky pro snažší odkazování z textu. V řádcích 3-29 počítáme souřadnice objemové krychle, které později využíváme k osvětlování v reálném čase. Mezi řádky 32-46 počítáme *alphaMisting*, což je proměnná, kterou zvětšujeme průhlednost mraku, ke kterému se blížíme. Poté až po řádek 71 zavádíme proměnné potřebné pro samotný *ray-casting* v mraku. Na následujících řádcích 81-85 zajišťujeme

prolétávání mrakem. Dále si definujeme proměnné do kterých akumulujeme hodnoty apod. Na řádku 102 nám poté začíná hlavní cyklus, který končí až s koncem metody na řádku 177.

V tomto cyklu procházíme pevným počtem kroků objemovou krychli mraku. Ta je normovaná na interval $\langle 0; 1 \rangle$ a proto vždy kontrolujeme, jestli jsme stále uvnitř intervalu. Pokud ano, načteme hodnotu a vykonáme krok ve směru paprsku o ekvidistantní délce. Jako hodnotu načítáme na řádku 115 z 3D textury poslední 2 složky – b a a, jelikož složky *r g b* mají v našem případě vždy shodnou hodnotu. Poté v řádcích 117 až 146 vykonáváme příkazy pro výpočet *real-time* osvětlení. Toto osvětlení poté upraví hodnotu *b*, kterou budeme akumulovat. Dokud neprojdeme počet kroků, popřípadě není splněna jiná podmínka (např. výsledný voxel již dosaduje téměř neprůhledného stavu apod.) procházíme cyklus. Po jeho ukončení spočítáme výslednou barvu voxelu (viz řádky 159 - 172).

3.2.5 Průlet mrakem

Jedním ze základních požadavků na tuto práci byl fakt, že se má dát mrakem prolétávat. Bylo třeba s tím počítat již v počátečním návrhu metody a vyžádalo si to několik specifických úprav. První překážka, kterou bylo třeba překonat, byla samotná podstata *ray-casting* metody využívající triku s vykreslením přivrácené a odvrácené strany krychle. Ta vyžaduje, aby vždy byla krychle vykreslena celá, jinak přestane fungovat regulérně. To jsme vyřešili zavedením dvou kamer – zadní (*back*) a přední (*front*). Přední kamera je reálná, kterou vidí uživatel. Zadní je vždy posunuta o *cameraBackOffset* (viz Vzorec 3.1) směrem za kameru *front* a z pohledu této kamery se vykresluje objemová krychle.

$$Camera\ Back\ Offset = \sqrt{3} \cdot Max\ Cloud\ Size + \varepsilon$$

Vzorec 3.1: Posun zadní kamery o úhlopříčku maximálně velkého mraku plus nějaké malé navýšení, pro zamezení aritmetických chyb.

Nikdy se nám tedy nemůže stát, že by kamera neviděla celou krychli objemu, ve kterém se právě nacházíme. Posun můžem být z podstaty algoritmu

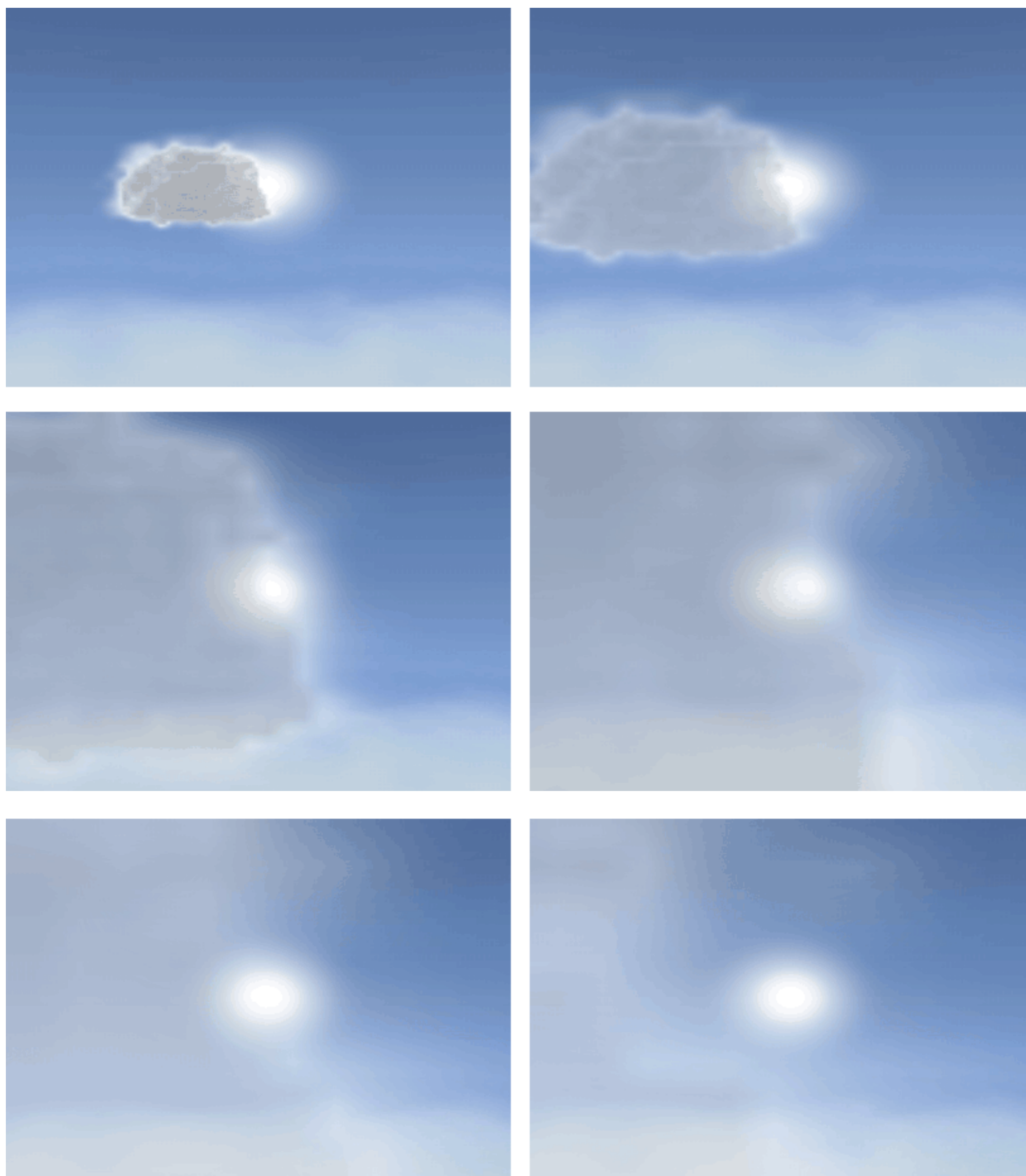
jakýkoli (např. 1000000) a jde pouze o to, aby byl vždy mimo krychli, kterou prolétáváme. Algoritmus se poté sám ke krychli „přimkne“.

Další věc, kterou bylo třeba řešit je posun imposteru. Ten se totiž vždy nachází tak, že pro zadní kameru zakrývá celý objem mraku. Pokud jsme však my již někde v objemu, nemůžeme tento imposter (viz 3.2.6 Impostery) vidět. Proto se musí vždy, když je uvnitř mraku naše přední kamera (tj. kamera, kterou reálně sledujeme scénu na monitoru), nebo letadlo (viz dále), posouvat imposter s námi tak, abychom ho viděli.

Kdyby se s námi pouze posouval, viděli bychom celý průlet skrze mrak stejnou texturu, jakou jsme viděli zvenčí. My ovšem chceme vidět aktuální texturu – v podstatě průřez mrakem. Tomu na míru jsme upravili naši *ray-casting* metodu v *shaderech*. Předáváme tedy *shaderům* pozici kamery násobenou inverzní maticí mraku. Takto upravená pozice se v našem *shaderu* jmenuje *PlayerinCloud* a jedná se o převod pozice hráče do takového souřadného systému, kde se objemová krychle mraku vždy nachází v počátku a je jednotková. Pokud je tedy hráč v objemu mraku, začneme kumulovat barvu každého voxelu od místa vstupu paprsku do objemu posunutého o vzdálenost, které je mezi tímto místem a pozicí kamery v mraku. O to se nám stará jednoduchá podmínka na řádcích 80 až 85 v Příloha B - Hlavní pixel shader. Jinak řečeno to tedy znamená, že budeme voxely nacházející se za hráčem ignorovat. Tato jednoduchá metoda tedy zajistí velmi hezky vypadající prolétávání mrakem, jak dokládá Obrázek 3.16.

Jelikož je tento program zaměřen obecně, postihuje dva základní typy pohledů v počítačových hrách. Jedná se o tzv. pohled z první osoby (*first person look*), kdy se imposter posouvá spolu s kamerou a pohled ze třetí osoby (*third person look*), kdy se přimkne imposter k předku letadla. V první zmíněné metodě tedy nejprve prolétne letadlo mrakem a teprve až když se v mraku ocitne kamera, začne se imposter posouvat. Druhá zmíněná metoda je o něco sofistikovanější – pokud letadlo dorazí do mraku, nejprve do něj zaletí, imposter se přimkne k ocasu letadla, utvoří se další imposter a ten se přimkne k čumáku letadla. V našem ukázkové programu je však ukázáno pouze zaletění a poté je vypnut imposter za

letadlem, aby se lépe demonstroval průlet (a to co „vidí“ pilot). Více o volbách těchto nastavení v Příloha C - Uživatelská příručka.



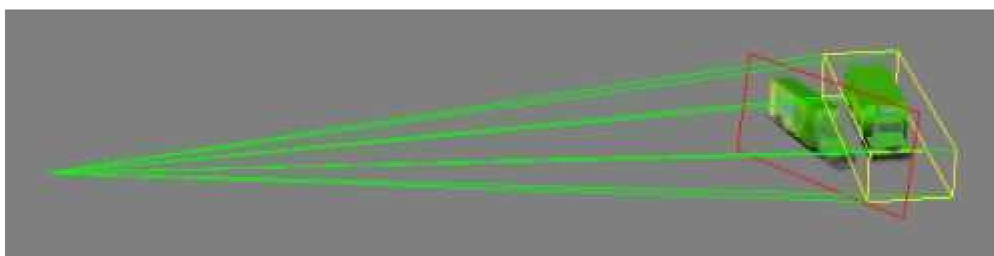
Obrázek 3.16: Ukázka průletu mrakem. Textura se dymanicky mění tak, jak prolétáváme objemem.

3.2.6 Impostery

V této práci nezřídka zmiňovaný imposter si vysvětleme detailněji. Jedná se o 2D náhradu 3D objektu, kterou vykreslujeme na místě tohoto objektu. Postup tvorby se dá shrnout do tří kroků:

1) Kolem objektu vytvoříme *bounding box*[10] a z jeho vrcholů odvodíme vrcholy imposteru, na který se bude vykreslený objekt nanášet. Docílíme tím správného natočení i velikosti – v podstatě vytvoříme billboard, který nám poté plně zakrývá 3D objekt viz Obrázek 3.17.
2) Vykreslíme 3D objekt, výsledný obrázek uložíme do textury, a tu naneseme na imposter.
3) Při změně polohy kamery vůči objektu vždy zkontrolujeme úhel svírající vektory kamery a normály imposteru. Pokud je tento úhel větší než konstanta ε, znovu vykreslíme objekt a texturu naneseme na imposter. V opačném případě vykreslujeme stále stejnou texturu.

Tímto postupem docílíme, že se daný objekt nevykresluje zdaleka tak často a u vzdálených objektů, popř. u objektů neurčitých tvarů jakými jsou mraky, není fakt, že využíváme několik snímků stále stejný obrázek, patrný.



Obrázek 3.17: Ukázka imposteru (červeně ohraničený billboard), vykresleného před 3D objektem automobilu (žlutě ohraničený). Zeleně jsou ukázány spojnice mezi kamerou a vrcholy bounding boxu. (Davis, 2006)

[10] Minimální možný kvádr, který kompletně ohraničuje daný objekt.

Pokud si uvedeme k jednotlivým bodům konkrétní implementaci, tak první bod - vytvoření imposteru vidíme na Výpis 3.5.

```
Vector3[] screenVertices = new Vector3[8]; // Rohy b. boxu.

// Projekce všech rohů do screen space
for (int i = 0; i < screenVertices.Length; i++)
{
    screenVertices[i] = device.Viewport.Project(corners[i],
     projection, view, Matrix.Identity);
}

// Určení maxima a minima os X a Y a minima osy Z - tam stačí
minimum, protože chceme nejbližší body obrazovce.
float minX, maxX, minY, maxY, minZ;
minX = maxX = minY = maxY = minZ = screenVertices[0].Z;
for (int i = 1; i < screenVertices.Length; ++i)
{
    minX = Math.Min(screenVertices[i].X, minX);
    maxX = Math.Max(screenVertices[i].X, maxX);
    minY = Math.Min(screenVertices[i].Y, minY);
    maxY = Math.Max(screenVertices[i].Y, maxY);
    minZ = Math.Min(screenVertices[i].Z, minZ);
}

// Pole, ve kterém jsou uloženy rohy billboardu.
Vector3[] screenSpaceVerts = new Vector3[4];
screenSpaceVerts[0] = new Vector3(minX, minY, minZ);
screenSpaceVerts[1] = new Vector3(maxX, minY, minZ);
screenSpaceVerts[2] = new Vector3(maxX, maxY, minZ);
screenSpaceVerts[3] = new Vector3(minX, maxY, minZ);

// Rohy billboardu převedeme zpět do jejich reálných pozic.
billboardVetices = new Vector3[4];
Vector3[] mImpostorVerts = new Vector3[4];
for (int i = 0; i < 4; i++)
{
    mImpostorVerts[i] =
    device.Viewport.Unproject(screenSpaceVerts[i], projection,
    view, Matrix.Identity);
}
```

Výpis 3.5: Zjištění souřadnic imposteru (resp. billboardu, na ketrý se později nanese dynamická textura), pomocí projekce bounding boxu do obrazovkového souřadného systému, vybrání správných bodů, tak, aby zakrývaly celý bounding box a vrácení těchto výsledných hodnot zpět do původního souřadného systému. (Hayward, 2008)

Druhý bod – vykreslení objektu do textury a její nanesení na imposter se v našem řešení ještě větví (nejprve vykreslíme texturu bez rozmazávání na čtverec

rovný její velikosti, výstup opět uložíme do textury a tu teprve nanášíme na imposter a vyhlazujeme pomocí lineární interpolace – viz 3.2.3 Vykreslování mraku), ale jelikož se v podstatě jedná o velmi podobný postup, ukažme si pouze první část – vykreslení objektu do textury na Výpis 3.6.

```
// Nastavíme jako cíl vykreslování texturu, která odpovídá
velikostí nejbližší mocnině 2 reálné velikosti mraku na obrazovce.
RenderTarget2D target = new RenderTarget2D(device,
GameConstants.width, GameConstants.height, 1,
device.PresentationParameters.BackBufferFormat);
device.SetRenderTarget(0, target);

// Vyčistíme pozadí bílou (aby mrak nezískal barevný nádech).
device.Clear(new Color(new Vector4(255, 255, 255, 0)));

// Vykreslíme objekt.
DrawLowRes(device, box, rectangle, world, viewProj, playerinCloud,
playerPosition);

// Nastavíme zpět cíl vykreslování a uložíme texturu do lowRes.
device.SetRenderTarget(0, null);
lowRes = target.GetTexture();
```

Výpis 3.6: Vykreslení 3D objektu do textury lowRes.

Poslední bod – kontrola úhlu normálového vektoru imposteru vůči vektoru z kamery do středu 3D objektu je v našem případě také lehce modifikován. V našem případě totiž může pozorovatel zůstat v neměnném úhlu vůči objektu, a přesto potřebujeme změnit texturu imposteru. Toto nastává při pohybu slunce. Abychom ovšem nevykreslovali při každém pohybu slunce všechny mraky, máme zavedenou kontrolu, jestli se i slunce pohnulo o více, než nějakou minimální mez.

```
// Určuje, zdali potřebuje imposter předaný jako parametr
updatovat, nebo můžeme použít již vygenerovanou texturu.
bool doesImposterNeedUpdate(Cloud imposter)
{
    // Vektor od kamery na čele letadla ke středu imposteru.
    Vector3 newCameraVec = cameraFront.Position -
    imposter.mImpostorCenter;
    newCameraVec.Normalize();

    // Pokud je skalární součin dvou normalizovaných vektorů roven
    jedné, jsou vektory rovnoběžné a stejně orientované.
    if (Vector3.Dot(newCameraVec, imposter.mImpostorCameraVec) <
    0.99999f)
    {
        return true;
    }
    else
    {
        // Pokud se hýbe slunce
        if (sunMoving)
        {
```

```
            if (alphaSunMove - imposter.OldAlphaSunMove > 0.05f)
            {
                imposter.OldAlphaSunMove = (float)alphaSunMove;
                return true;
            }
            else
            {
                return false;
            }
        }
        else
            // Pokud jsme blízko mraku
            if ((cameraFront.Position -
            imposter.meshCenter).Length() <
            GameConstants.maxCloudSize * 2)
            {
                return true;
            }
            else
            {
                return false;
            }
    }
}
```

Výpis 3.7: Metoda, která na základě normálového vektoru imposteru a vektoru mezi kamerou a středem imposteru určí, jestli se poloha pozorovatele změnila natolik, že se musí vykreslit objekt znovu, anebo můžeme používat stále již vygenerovanou texturu.

3.2.7 Zjišťování viditelnosti mraku

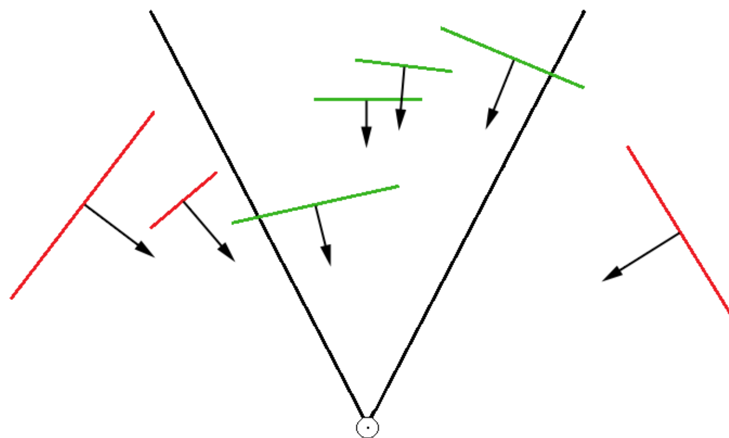
Impostery zmiňované v předchozí podkapitole mají ještě jednu výhodu a to pro zjišťování viditelnosti mraku. Celý mechanismus funguje tak, že si pro každý mrak nejprve spočteme jeho imposter (viz Výpis 3.5). Dále si najdeme střed tohoto imposteru a vektor směřující od středu imposteru ke kameře (viz Výpis 3.8). Poté si spočítáme úhel mezi tímto vektorem a vektorem, který určuje směr pohledu kamery. Pokud je tento úhel mimo pozorovací úhel kamery (a pokud se nenacházíme uvnitř mraku), skončíme a přeskočíme celý další proces vykreslování.

```
// Spočítáme si střed imposteru.
impostorCenter = Vector3.Zero;
impostorCenter = mImpostorVerts[0] + mImpostorVerts[1] +
mImpostorVerts[2] + mImpostorVerts[3];
```

```
impostorCenter *= .25f;

// Vektor mezi imposterem a kamerou.
impostorCameraVec = cameraFront.Position - impostorCenter;
impostorCameraVec.Normalize();

// Vektor určující pohled kamery v přímém směru.
Vector3 normalLookAtCamera = cameraFront.Position -
cameraFront.LookAt;
normalLookAtCamera.Normalize();

// Úhel svírající vektor pohledu kamery a vektor jdoucí k
imposteru.
float scalarIL = Vector3.Dot(normalLookAtCamera,
impostorCameraVec);// normalImprostorCamera);
float radiansTmp = (float)Math.Cos(scalarIL);

// Pokud máme mrak v zorném úhlu, popřípadě pokud jsme uvnitř
mraku, vykresluj dál.
if (radiansTmp < GameConstants.fieldOfView || distanceCameraCloud
< 0)
      {
      // Vykresluj
      }
```

Výpis 3.8: Testování úhlu svírajícího vektor od kamery k imposteru s pohledovým vektorem kamery.

Takovýto jednoduchý test nám tedy zajistí odřezání všech mraků, které nejsou v pohledovém jehlanu – pro ilustraci na Obrázek 3.18 vidíme situaci se sedmi mraky ve scéně, kdy vykreslujeme pouze ty vyznačené zeleně. Červeně vyznačené potom zahazujeme.



Obrázek 3.18: Zjišťování viditelnosti sedmi mraků. Tři červené jsou mimo pozorovací úhel a nevykreslují se, čtyři zelené se vykreslují.

Po tomto testu, následuje ještě další, který zjišťuje, jak je velký mrak při vykreslování (počet jeho pixelů) a pokud zjistí, že je příliš malý, opět přeskočí celý následný proces vykreslování. O tomto dalším testu se dočteme podrobně v kapitole 3.2.11 Řízení úrovně detailů.

3.2.8 Osvětlení mraku

Osvětlení mraku je v řadě detailů zcela originální a vymyšlené přímo pro tuto práci. Podobně jako osvětlení, které uvádí M. Harris je dvoufázové - první fáze se odehrává v *pre-processing* fázi na CPU a druhé v reálném čase na GPU, zde však podobnosti končí.

Většina existujících řešení problémů neuvažuje změnu polohy světla v reálném čase. I když tedy stíny a osvětlení vypadají reálně, nelze je aplikovat na velkou množinu počítačových her, ve které je třeba měnit polohu slunce a tudíž i osvětlení v reálném čase. Náš cíl byl tedy opačný - nesnažit se o ultra-realistické osvětlení, ale vymyslet princip, který bude použitelný v co nejširším spektru počítačových her.

Máme tedy statickou proměnnou *GameConstants.sunPosition*, která udává pro celý program polohu slunce. Poté nejprve v *pre-processing* fázi programu vypočítáme barvu všech voxelů v každém mraku a tento výsledek uložíme do příslušné 3D textury. Tento výpočet svým principem vychází z *ray-casting* metody (viz 2.4.3 Další užitečné metody), avšak namísto vysílání paprsku z oka skrze promítací plát, prochází celý objem mraku, z každého voxelu vyšle paprsek směrem do slunce, cestou naakumuluje barvu z dalších voxelů, kterými projde a výslednou barvu přiřadí do toho, z kterého byl vyslán. Metodu pro výpočet barvy jednoho voxelu vidíme na Výpis 3.9.

```
private float computeVoxelColour(float x, float y, float z)
{
    float color = 0.0f;       // Zde se bude akumulovat barva.
    int colorCounter = 0;     // Počer prošlých voxelů.
    Vector3 step = new Vector3(1.0/size, 1/size, 1/size); // Krok.
    bool insideBox = true;
    Vector3 startVoxelPosition = new Vector3(x, y, z); // Počátek.
    Vector3 rayStart = Vector3.Transform(startVoxelPosition,
    localWorld);              // Voxel V ve World souřadnicích.
    Vector3 actRayPosition = rayStart;
    Vector3 rayDirection = sunPosition - rayStart; // Směr.
    rayDirection.Normalize();

    // Dokud jsme uvnitř objemu, akumuluj barvu.
    while (insideBox)
    {
        // Skoč vždy o step ve směru paprsku.
        actRayPosition += step * rayDirection;

        // VoxelIndex - přepneme se do krychle <0; 1>.
        Vector3 voxelIndex = Vector3.Transform(actRayPosition,
        Matrix.Invert(localWorld));

        // Pokud jsme mimo objem – skonči, jinak pokračuj.
        if (isVoxelOutOfUnitCube(voxelIndex))
            insideBox = false;
        else
        {
            // Přepočítej index do 1D pole.
            int index = indexIn1DArray(voxelIndex, size);
            // Naakumuluj barvu, která se tam nachází.
            color += (data[index] * GameConstants.lighter);
            colorCounter++;
            // Pokud je již barva hodně tmavá, skonči.
            if (color > 250)
            {
                insideBox = false;
```

```
                colorCounter = (int)(colorCounter *
                 GameConstants.darkLight);
            }
        }
    }

    // Výsledek, jako naakumulovanou barvu / počet prošlých barev
    color = color / colorCounter;
    if (color < 100)
        return 255;
    else
        return 255 - color;
}
```

Výpis 3.9: Osvětlovací metoda v pre-processingu. Pro každý voxel V vyšleme paprsek směrem ke slunci a pomocí této metody spočítáme jeho barvu.

U metody uvedené výše nehraje výpočetní náročnost prim, jelikož se provede pouze jednou. Mrak se vystínuje a toto osvětlení je dostatečné (viz Obrázek 1.1, který obsahuje právě statické osvětlení) pro aplikace s neměnnou pozicí slunce. My ovšem potřebujeme ještě jednu metodu, která bude co nejrychlejší, aby se mohla vykonávat v reálném čase. Tato podmínka znamená, že metoda nebude moci poskytovat komplexní osvětlovací model, ale bude pouze dotvářet iluzi. Vyvinuli jsme tedy velmi rychlou metodu, která se provádí na GPU přímo v hlavní smyčce pro výpočet barvy a průhlednosti *voxelu* při *ray-castingu*. Tato smyčka prochází objem po ekvidistantních krocích a vždy naakumuluje hodnoty voxelu, kde se právě nachází. My v tu chvíli ještě vypočítáme změnu hodnoty barvy - zavoláme funkci *dynamicLight* v *shaderu Cloud.fx* a jako parametr jí dáme polohu tohoto voxelu. Nejprve se tedy podívejme na část hlavní smyčky *ray-castingu* – viz Příloha B - Hlavní pixel shader, číslo řádku 116 až 143.

Metoda *dynamicLight* (řádek 126) nám vypočítá vzdálenost voxelu V (což je aktuální voxel ve kterém se nacházíme) a průsečíku T_{min}. To je průsečík paprsku, jdoucího mezi voxelem V a sluncem, s objemovou krychlí (tzn. v podstatě *bounding boxem* kolem všech voxelů jednoho mraku). Podívejme se na samotné tělo metody ve Výpis 3.10.

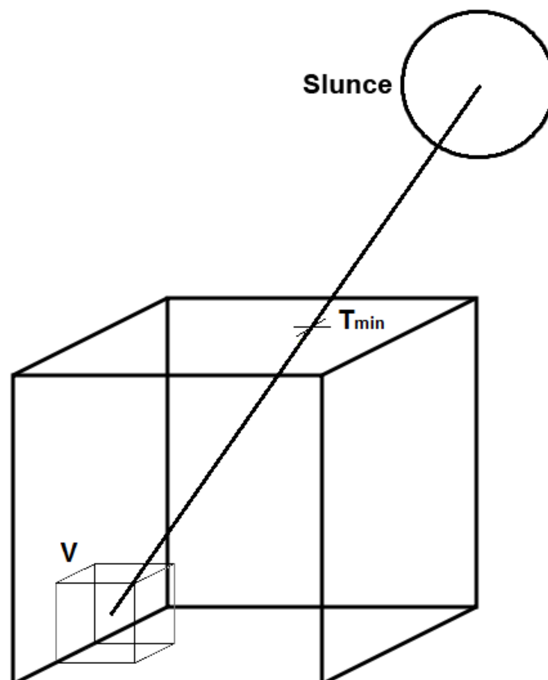
```
float dynamicLight(float3 posV)
{
    // Vydělíme homogenní souřadnicí.
    float4 pos = mul(float4(posV.xyz, 1), World);
    pos.xyz /= pos.w;
    pos.w = 1;

    // Vypočteme parametry průsečíků přímky, proložené voxelem a
   sluncem se všemi rovinami, které ohraničují náš mrak.
    float tl1 = tIntersect(A, B, C, pos);
    float tl2 = tIntersect(B, C, G, pos);
    float tl3 = tIntersect(D, C, G, pos);
    float tl4 = tIntersect(A, D, E, pos);
    float tl5 = tIntersect(A, B, F, pos);
    float tl6 = tIntersect(E, H, G, pos);

      // Nalezmene nejbližší kladný parametr
    float tMin = 1;
    if (tl1 > 0 && tl1 < tMin)
        tMin = tl1;
    if (tl2 > 0 && tl2 < tMin)
        tMin = tl2;
    if (tl3 > 0 && tl3 < tMin)
        tMin = tl3;
    if (tl4 > 0 && tl4 < tMin)
        tMin = tl4;
    if (tl5 > 0 && tl5 < tMin)
        tMin = tl5;
    if (tl6 > 0 && tl6 < tMin)
        tMin = tl6;

    // Vzdálenost mezi voxelem a sluncem vynásobíme parametrem
   tMin a získáme tak vzdálenost mezi voxelem a průsečíkem.
    return length(pos - lightPosition) * tMin;
}
```

Výpis 3.10: Metoda počítá vzdálenost voxelu V na pozici posV s průsečíkem tMin na hraně krychle mraku. Průsečík je na úsečce mezi voxelem a sluncem.

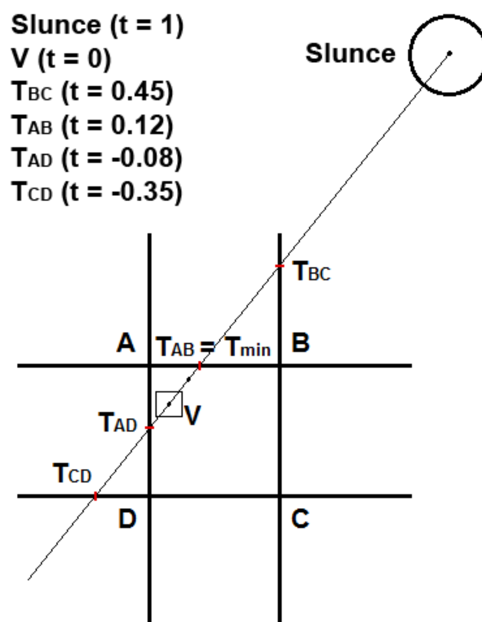
Abychom si ilustrovali tuto metodu, pojďme si ji ukázat graficky. Na Obrázek 3.19 vidíme voxel V, ve kterém se nacházíme a slunce, mezi nimiž je proložená úsečka. Na této úsečce poté najdeme průsečík T_{min} s objemovou krychlí.

Naše metoda funguje tak, že nalezne pouze parametry průsečíků přímky p (přímka, na které leží voxel V a slunce) s 6 plochami L_1 až L_6 (na každé leží jedna stěna krychle). Parametry (t_1 až t_6) fungují tak, že voxel V má hodnotu parametru 0, slunce 1, a tudíž hledáme jen v intervalu $\langle 0; 1 \rangle$, a to takový parametr, který je

nejblíže nule. Pro lepší představu ještě jedna ukázka převedená do 2D – Obrázek 3.20.



Obrázek 3.19: Úsečka proložená mezi voxelem V a sluncem. T_{min} je poté průsečík této úsečky s objemovou krychlí.



Obrázek 3.20: Převedení našeho případu do 2D, sloužící pro snažší ilustraci. Průsečíky přímky proložené sluncem a pixelem V se všemi 4 přímkami ohraničujícími čtverec.

Výhoda tedy spočívá v tom, že my nehledáme konkrétní body a nepočítáme vzdálenosti od voxelu z nich. Zajímá nás pouze vzdálenost, ne konkrétní bod, a tu si můžeme zjistit pouze z parametru, čímž ušetříme řadu operací. Celá naše metoda má tedy složitost pro každý voxel $O(1)$, a tudíž téměř nezpomaluje výpočet. Jedná se sice o aproximaci osvětlení založenou na poloze voxelu v objemu, ale funguje překvapivě dobře, a tudíž jde pro hry o vhodné řešení. Tím, že metoda zesvětluje v jiném poměru světlé a tmavé části, potřebuje pro lepší výsledky „počáteční nástřel“. Ten jí právě poskytuje offline metoda, která je daleko přesnější a výpočetně náročnější. Proto je ideální kombinovat tyto dvě osvětlovací metody.

3.2.9 SkyBox, slunce a vítr

Tvorbu textury SkyBoxu a slunce jsme si podrobně popsali v kapitole 3.1.2 SkyBox a slunce, nyní si ještě ve zkratce vysvětlíme jejich programovou realizaci.

Skybox.cs je jednoduchá třída, která pouze nadefinuje krychli o velikosti $\langle -1; 1\rangle$, načte texturu SkyBoxu typu *TextureCube*, kterou jsme vytvářeli a oboje předá shaderu *Skybox.cs*. Shader vždy posune krychli na pozici kamery, přetransformuje ji do projekčního prostoru, namapuje texturu (vnitřně se jedná o soubor 6 textur) na krychli, vypne *z-buffer* a krychli vykreslí. Tím, že vypneme *z-buffer*, nekolidují s krychlí žádné další objekty apod. a spolu s tím, že se posouvá s námi, vytváří dojem, že je nekonečně velká.

Poloha slunce je reprezentována proměnnou *GameConstants.sunPosition*. Tato proměnná se používá ve všech výpočtech osvětlování v programu a je defaultně v poloze $X = -10000;\ Y = 0;\ Z = 0$, tj. na horizontu. Pokud spustíme demostrační rotaci, začne se poloha měnit v osách X a Y viz Výpis 3.11. Na poloze slunce je umístěn billboard stále otočený k hráčovi, který má jako texturu Obrázek 3.13.

```
if (sunMoving)
{
      alphaSunMove += 0.01f;

      //Pohyb slunce po kružnici
      GameConstants.sunPosition.X = (float)(10000 *
      Math.Cos(alphaSunMove));
      GameConstants.sunPosition.Y = (float)(10000 *
      Math.Sin(alphaSunMove));
}
```

Výpis 3.11: Pokud je spuštěný pohyb, při každém zavolání této funkce posuneme slunce o kousek po kružnici.

Celá tato jednoduchá realizace obsahuje ještě *shader Sun.fx*, který se pouze stará o správné namapování textury na billboard slunce a vypnutí *z-bufferu* před jeho vykreslením.

Celou trojici jednoduchých metod uzavírá metoda větru. Jelikož má metoda vykreslování mraku jako parametr posun, stačí nám jednoduše z hlavní metody vždy zadávat nějaké malé číslo, které se před vykreslením přičte k pozici mraku a posune ho. Číslo lze zadávat konstantní, popř. hodnotu na sinusoidě, která zajistí nerovnoměrný pohyb.

3.2.10 Alfa průhlednost

Jelikož jsou mraky částečně průhledné a celé vykreslování funguje na principu nanášení textur těchto mraků na impostery, je třeba věnovat zvýšenou pozornost alfa průhlednosti.

Alfa kanál definuje průhlednost textury. Jedná se v podstatě o další obrázek, který je ve stupních šedi. Tam, kde je černý, je textura, ke které se alfa kanál vztahuje, zcela průhledná. Tam, kde je bílý, je zcela neprůhledná. Na hodnotách barvy mezi těmito dvěma ohraničujícími barvami je textura potom více či méně průhledná.

My si zde vysvětlíme dvě zásadní témata týkající se průhlednosti. Zaprvé průhlednost a její řízení pro každý jednotlivý mrak a zadruhé korektní zobrazení řady imposterů s alfa texturou za sebou.

Před vykreslováním každého mraku je vždy třeba zapnout průhlednost *device.RenderState.AlphaBlendEnable = true*. Každý mrak je poté více či méně průhledný – to vychází z vykreslovacího algoritmu na GPU. Ovšem pro realističtější pocit přidáme ještě efekt, který nám zajistí, že při přibližování se k mraku začne mrak zprůhledňovat. Podívejme se na konkrétní část z programu na GPU - v části Příloha B - Hlavní pixel shader na řádcích 32 až 46. Zjistíme si vzdálenost mezi středem mraku a pozorovatelem, a pokud je menší rovna dvojnásobku velikosti mraku, bude hodnota *alphaMisting* < *1*. Řádky 169-172 poté ukazují jak *alphaMisting* ovlivňuje finální průhlednost (*result.a*). Efekt je tedy takový, že se mrak při příletu pozorovatele částečně zprůhlední a více budí dojem mlhoviny.

Pojďme si nyní vysvětlit druhou část, a to řešení viditelnosti z důvodu použití *z-bufferu*[11]. Jelikož chceme vykreslovat více poloprůhledných objektů přes sebe, musíme využít nějakého algoritmu. My použijeme princip malířova algoritmu, neboli budeme vykreslovat nejprve nejvzdálenější objekty, přes ně ty bližší atd. až k těm nejbližším. Tento jednoduchý algoritmus si můžeme dovolit používat proto, že máme jistotu, že se žádné dva mraky neprolínají. V takových případech totiž tento algoritmus nefunguje.

Máme tedy pole mraků *arrayOfClouds[]*. Vytvoříme si další pole *cloudsDistance[]*, do kterého budeme ukládat vždy vzdálenost mraku s příslušným indexem od kamery (viz Výpis 3.12).

```
for (int i = 0; i < cloudsDistance.Length; i++)
{
        cloudsDistance[i] = (int)(cameraFront.Position -
        arrayOfClouds[i].meshCenter).Length();
}
```

[11] Z-buffer je paměťový zásobník, obsahující Z-ové souřadnice pixelů v prostoru obrazovky. Tyto souřadnice nám udávají jak daleko od pozorovatele je daný předmět a využívají se právě pro kreslení ve správném pořadí.

```
Sort(cloudsDistance);
```

Výpis 3.12: Uložení do pole vzdáleností každého mraku od kamery a následné srovnání tohoto pole.

Pole *cloudsDistance[]* poté seřadíme a vykreslíme od nejvzdálenějšího k nejbližšímu (viz Výpis 3.13). Poté máme vždy zobrazeny mraky ve správném pořadí, což je dobře vidět na Obrázek 3.21.

```
// Zde vykreslíme všechny mraky v opačném pořadí, než je jejich
vzdálenost ke kameře
// (nejprve nejvzdálenější, nejpozději nejbližší).
for (int i = 0; i < GameConstants.noOfClouds; i++)
{
    // Od nejvzdálenějšího
    int tmp = cloudsDistance[GameConstants.noOfClouds - i - 1];

    // Poprvé každý mrak vykreslíme plnou metodou.
    if (firstPass < GameConstants.noOfClouds)
    {
        firstPass++;
        arrayOfClouds[tmp].DrawFull(graphics, gameTime,
        cameraFront, cameraBack, Vector3.UnitZ,
        player1.LocalWorld);
    }
    else
    {
        // Pokud potřebuje imposter updatovat, vykreslíme znovu
        celý mrak. Jinak použijeme již vygenerovanou texturu a
        vykreslíme ji znovu.
        if (doesImposterNeedUpdate(arrayOfClouds[tmp]))
            arrayOfClouds[tmp].DrawFull(graphics, gameTime,
            cameraFront, cameraBack, Vector3.UnitZ,
            player1.LocalWorld);
        else
            arrayOfClouds[tmp].DrawAgain(graphics, gameTime,
            cameraFront);
    }
}
```

Výpis 3.13: Vykreslování mraků od nejvzdálenějšího k nejbližšímu.

Tato procedura tedy spočítá v podstatě pouze v seřazení pole celých čísel, a proto její rychlost dramaticky ovlivňuje použitý algoritmus řazení. Využíváme tzv. *Fast Bucket Sort*, což je v současné době jeden z nejefektivnějších algoritmů řazení (viz Výpis 3.14). Vychází principiálně z algoritmu *quicksort*, ale

inteligentně si volí pivot jako hodnotu uprostřed intervalu hodnot. Je to tedy na rozdíl od *quicksort* algoritmus stabilní a má náročnost $O(n)$.

```
public static void Sort(int[] integers)
{
    // Nelezní maximálního a minimálního prvku
    int maxValue = integers[0];
    int minValue = integers[0];

    // Začínáme prohledávat od 2. prvku
    for (int i = 1; i < integers.Length; i++)
    {
        if (integers[i] > maxValue)
            maxValue = integers[i];
        if (integers[i] < minValue)
            minValue = integers[i];
    }

    // Každá hodnota bude uložena s korespondujícím indexem.
    // Např. 34 => index 34 bude minValue
    LinkedList<int>[] bucket = new LinkedList<int>[maxValue -
     minValue + 1];

    // Přesun do „vědra"
    for (int i = 0; i < integers.Length; i++)
    {
        if (bucket[integers[i] - minValue] == null)
            bucket[integers[i] - minValue] = new
            LinkedList<int>();
        bucket[integers[i] - minValue].AddLast(i);
    }

    // Přesun z vědra zpět do originálního pole.
    int k = 0;
    for (int i = 0; i < bucket.Length; i++)
    {
        if (bucket[i] != null)
        {
            LinkedListNode<int> node = bucket[i].First;
            while (node != null)
            {
                integers[k] = node.Value;
                node = node.Next;
                k++;
            }
        }
    }
}
```

Výpis 3.14: Fast Bucked Sort - neboli urychlený „vědrový" algoritmus. Jeden z nejefektivnějších algoritmů řazení. Převzato z (Vckicks, 2008).

3.2.11 Řízení úrovně detailů

Jelikož jsme si metodu vykreslování navrhovali sami, uvažovali jsme již od počátku se zahrnutím co nejefektnějšího řízení úrovně detailu mraků. Využíváme pro to základního principu *ray-tracingu* (viz 2.4.3 Další užitečné metody), kde pro každý pixel promítacího plátu vyšleme paprsek z oka (tzn. od kamery), skrze daný pixel. Pro tento paprsek poté akumulujeme barvu a průhlednost, což je poměrně náročný výpočet. Náš algoritmus jde tedy cestou snižování rozlišení promítacího plátu, a tím i počtu paprsků.

Nejprve si musíme zjistit, jakou plochu zabírá mrak na obrazovce. To provádíme ve Výpis 3.5 – a tyto hodnoty máme uloženy v poli *screenSpaceVerts*. Těchto hodnot využijeme ke zjištění šířky (*screenW*) a výšky (*screenH*) mraku, kterou ještě znásobíme konstantou *lodLength* (viz Výpis 3.15), která je rovna Vzorec 3.2.

$$lodLength = \frac{3 \cdot maximální\ velikost\ textury\ mraku}{výška\ okna\ aplikace}$$

Vzorec 3.2: Konstanta zajišťující snížení nejvyššího rozlišení mraku.

a zajišťuje, aby mraky nepřecházely do maximálních povolených detailů moc daleko od hráče. Maximální povolené detaily (neboli maximální délka hrany promítacího plátu - *defaultSize*) byly zjištěny empiricky jako minimální velikost, při které mrak stále vypadá dobře, a jsou rovny hodnotě 64, tedy 2^6. Jelikož promítací plát je textura, kterou budeme dále mapovat apod., budeme se držet obecně známého doporučení, udržovat texturu čtvercovou, s délkou hrany odpovídající mocnině dvou. Proto i dále v našem Výpis 3.15 určujeme délku hrany promítacího pásu (*screenPow*) jako nejbližší mocninu dvou z minima *screenW* a *screenH*. Pokud je *screenPow* větší, než *defaultSize*, přiřadíme mu velikost *defaultSize*. Pokud je naopak menší, nebo roven minimální velikosti mraku, skončíme celý cyklus a mrak nevykreslujeme.

```
// Zjištění šířky a výšky billboardu na obrazovce a její zmenšení
(i mrak má maximální velikost textury malou)
float screenW = (screenSpaceVerts[1] -
screenSpaceVerts[0]).Length() * GameConstants.lodLength;
float screenH = (screenSpaceVerts[3] -
screenSpaceVerts[0]).Length() * GameConstants.lodLength;

// Výpočet nejbližší mocniny dvou z minima šířky a délky
int screenPow = NextPow2((int)Math.Min(screenW, screenH));

// Pokud je mrak větší, než defaultSize, bude velký defaultSize
if (screenPow > GameConstants.defaultSize)
    screenPow = GameConstants.defaultSize;

// Pokud je mrak moc malý, nezobrazovat; Jinak nastavíme width a
height mraku na tmp
if (screenPow <= GameConstants.invisibleSize)
    return;
else
{
    GameConstants.width = GameConstants.height = screenPow;
}
```

Výpis 3.15: Výpočet velikosti promítacího plátu mraku.

V tomto postupu je velmi důležitá výpočetní náročnost metody, pro hledání nejbližší mocniny dvou. Základní metoda založená na logaritmech je velmi výpočetně náročná – viz Výpis 3.16.

```
int NearestSuperiorPowerOf2(int n)
{
   return (int)Math.Pow(2, Math.Ceiling(Math.Log10(n)/
Math.Log10(2)));
}
```

Výpis 3.16: Původní pomalá funkce pro nalezení nejbližší mocniny dvou pomocí logaritmů v standardní matematické knihovně.

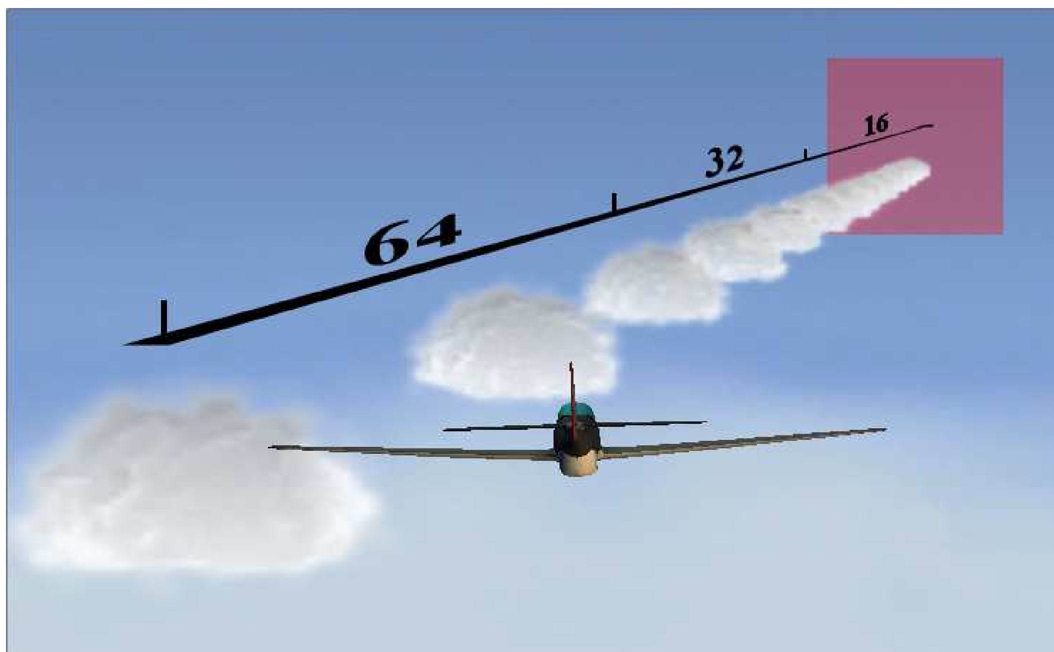
Provedli jsme tedy menší výzkum a našli jednu z nejrychlejších metod tzv. Saturaci - viz Výpis 3.17. Více pro porovnání v kapitole 4 Experimenty a výsledky.

```
int NextPow2(int x)
{
    --x;
```

```
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    return ++x;
}
```

Výpis 3.17: Jedna z nejrychlejších metod pro nalezení nejbližší mocniny dvou. Převzato z (Neoztar, 2004).

Jak vidíme, je metoda řízení detailů navržena tak, že velmi jednoduše a plynule koriguje detaily mraků. Na rozdíl od metod, které mají většinou 2-3 stupně detailnosti, nejsou zdaleka tak patrné předěly, kdy se mrak začne vykreslovat s jinými detaily. Celou metodu ještě shrnuje Obrázek 3.21, na kterém vidíme mraky, které jsou mimo vykreslovací oblast (naznačeno červeným čtvercem), a poté ty, které jsou mezi mantinely maximálních a minimálních detailů. Pokud se nesoustředíme, lze jen stěží rozpoznat, že mají vzdálenější mraky texturu s nižším rozlišením. Za cenu minimálního zhoršení vizuálních výsledků nám tedy metoda nabízí velké urychlení (viz 4.2 Výsledná měření našeho programu).



Obrázek 3.21: Ukázka z programu, demonstrující řízení úrovně detailů. Je zde velmi pěkně vidět, že textura u vzdálených mraků bohatě postačuje.

4 Experimenty a výsledky

Před závěrečnou sumarizací si uveďme několik měření, které ukáží do jaké míry je naše řešení využitelné v praxi a pomohou nám kvantifikovat míru úspěšnosti našeho přístupu.

Hardwarové (HW) i softwarové (SW) vybavení přenosného počítače, na kterém probíhalo vývoj a následná měření jsou uvedeny v Tabulka 4.1.

Laptop DELL INSPIRON 1720	
Operační systém	MS Windows Vista Home (Service Pack 1)
CPU typ	Intel Pentium Dual T2390
CPU frekvence	2 · 1862MHz
Operační paměť	3072 MB
Grafická karta	nVidia GeForce 8600M GT
Paměť grafické karty	256 MB
Shader model	4.0
Verze ovladačů grafické karty	195.62
DirectX verze	10.1

Tabulka 4.1: Hardwarové a softwarové vybavení testovacího počítače.

Jelikož se od sebe samozřejmě liší výkonově i počítače se stejným HW vybavením (odchylky způsobené množstvím souběžně spuštěných aplikací apod.), uvádíme v Tabulka 4.2 výsledky měření z testovacího programu, zaměřeného na výkon počítačů v herních aplikacích, *3DMark 2006*.

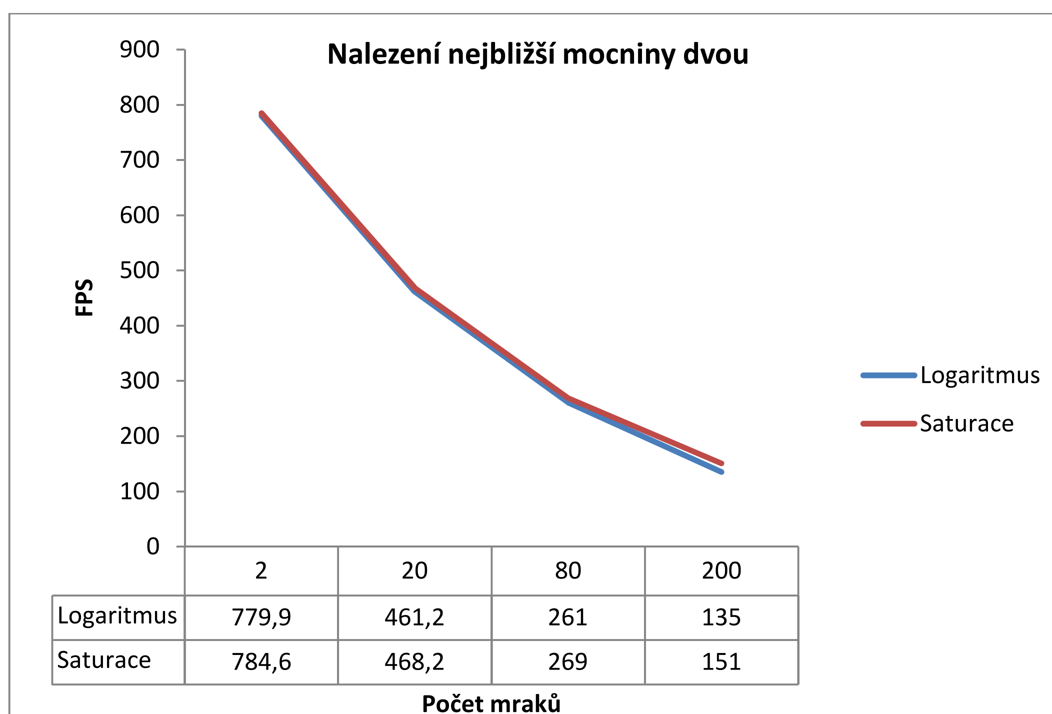
3DMARK 2006 verze 1.1.0	
Skóre („Default“, rozlišení 1280 × 800)	3231
HDR/Shader Model 3.0 skóre	1297
CPU skóre	1085

Tabulka 4.2: Výsledky měření v programu 3DMark 2006.

Dosažené skóre je podle aktuálních stránek výrobce testovacího programu(Futuremark Corporation, 2010) méně než polovina průměrného herního PC. Proto tedy lze s jistotou tvrdit, že pokud bude řešení dostatečně rychlé na tomto HW, bude použitelné pro širokou hráčskou obec.

4.1 Experimenty a výběr vhodného řešení

Naším prvním experimentem se pokusíme ukázat rozdíly mezi rychlostí programu s původní metodou pro hledání nejbližší mocniny dvou pomocí logaritmů a „finty" nazývané saturace (viz 3.2.11 Řízení úrovně detailů). Pokud bychom provedli měření samotných dvou metod, zjistíme, že saturace je několikanásobně rychlejší (viz fórum (Neoztar, 2004), kde například uživatel Thezensunni uvádí trvání saturace 1442 tiků procesoru ku 38376 tikům u logaritmické metody, tzn., saturace se vykoná cca 26,6 krát rychleji).



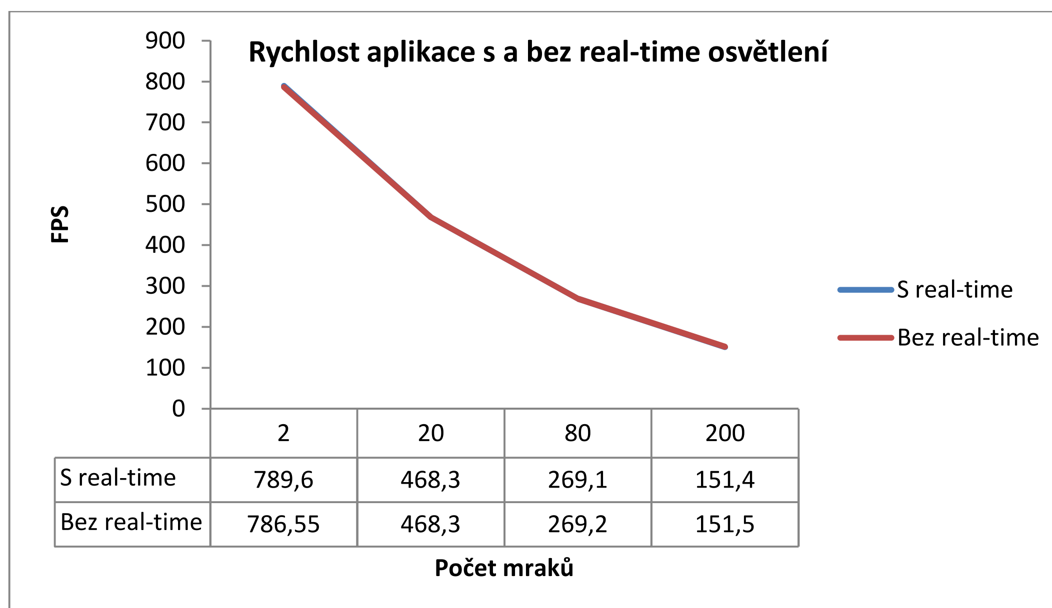
	2	20	80	200
Logaritmus	779,9	461,2	261	135
Saturace	784,6	468,2	269	151

Graf 4.1: Na tomto grafu je patrné urychlení naší aplikace, pokud použijeme v řízení úrovně detailů k namísto logaritmické metody saturaci.

My provádíme měření přímo v programu, kde samozřejmě vzhledem k řadě dalších výpočtů nebudou rozdíly tak propastné – viz Graf 4.1. Na grafu je patrné zrychlení, které se s počtem mraků (a tudíž i prováděním metod pro nalezení mocniny dvou) zvětšuje. Naše volba saturační metody se tedy ukázala jako správná a přínosná.

4.2 Výsledná měření našeho programu

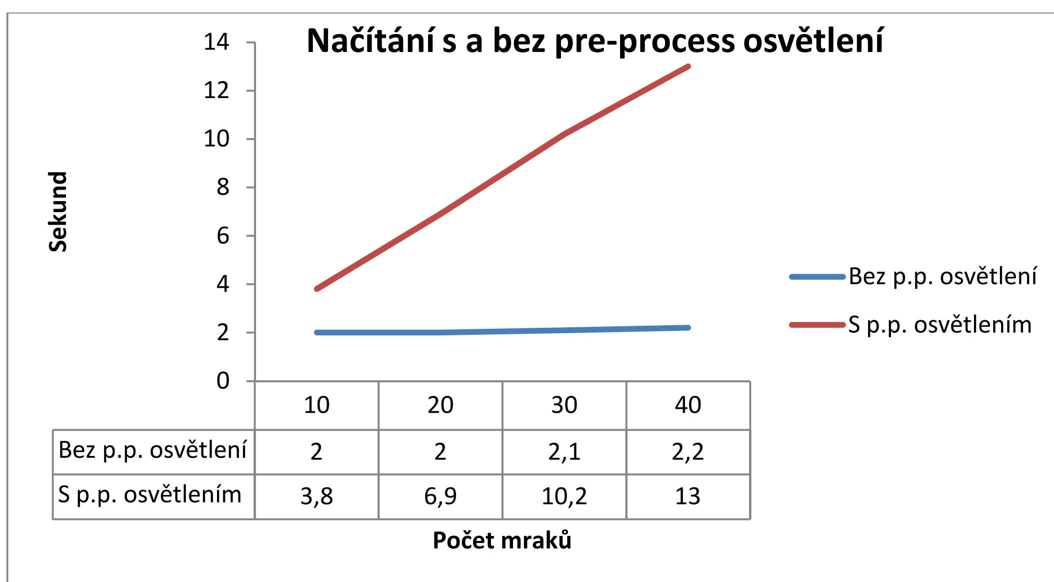
Pojďme si nyní změřit výkonnost našeho programu s ohledem na některé důležité aspekty. Graf 4.2 nám ukazuje rozdíl v rychlosti programu s vypnutým a zapnutým *real-time* osvětlováním. Je až zarážející, že výsledné hodnoty se vůbec nemění (dokonce drobné odchylky v měření způsobují, že je někdy se zapnutým osvětlováním program rychlejší) a pouze to poukazuje na skutečnost, že je tato metoda velice rychlá. Jedná se opravdu pouze o pár aritmetických operací, které v prizmatu pohledu celé aplikace nehrají žádnou roli v navýšení výpočetního výkonu.



	2	20	80	200
S real-time	789,6	468,3	269,1	151,4
Bez real-time	786,55	468,3	269,2	151,5

Graf 4.2: Rozdíl v rychlosti vykreslování s a bez osvětlování v reálném čase (real-time). Ukazuje se, že námi navržená metoda real-time osvětlování nezpomaluje výslednou aplikaci.

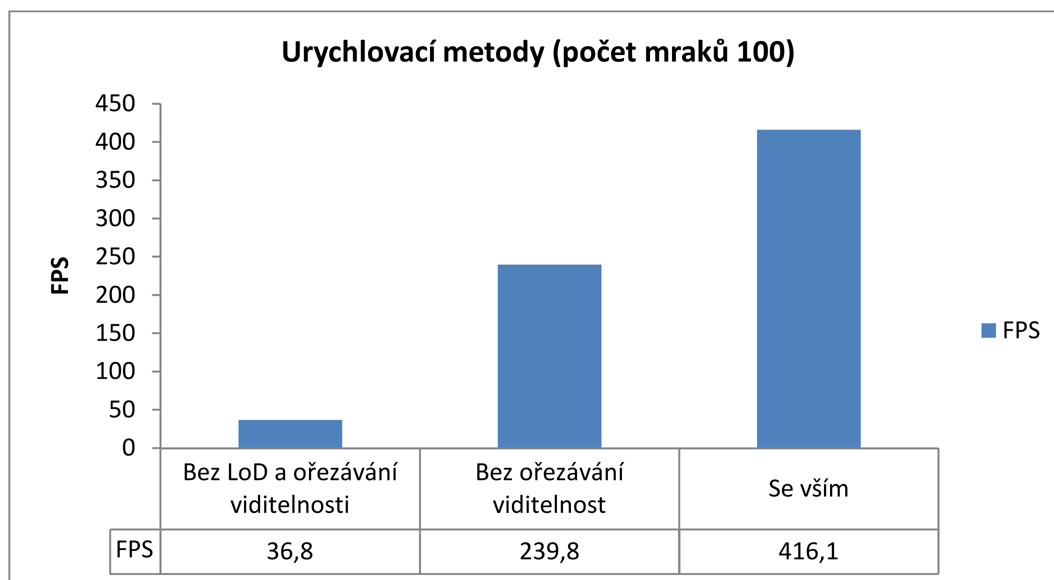
Nyní si ukažme další důležitý aspekt a tím je doba načítání aplikace s předpočítaným světlem (*pre-process* osvětlení). Při velmi nízkém počtu mraků to není tak znatelné, ale dle grafu a tabulky v něm uvedené je zřejmé, že zatímco rychlost bez *p.p.* osvětlení je s přibývajícími mraky téměř konstantní, tak rychlost s *p.p.* roste v lineárně a již při 40 mracích je to znatelné zpomalení. Ukazuje se tedy, že metoda s *pre-process* osvětlením (p.p.) je použitelná pouze do scén s řádově maximálně stovkami mraků.



	10	20	30	40
Bez p.p. osvětlení	2	2	2,1	2,2
S p.p. osvětlením	3,8	6,9	10,2	13

Graf 4.3: Čas potřebný pro spuštění aplikace s a bez pre-process osvětlení. Při vyšším počtu mraků je vidět markantní nárůst v čase s p.p. osvětlením.

Poslední graf, který si zde uvedeme, je Graf 4.4. Ten ukazuje scénu se 100 náhodně rozmístěnými mraky (v předchozích měřeních byly kvůli testování mraky vždy stejně velké a v řadě, proto je nyní FPS vyšší), ve které postupně není zapnuta žádná metoda urychlování, poté zapneme metodu řízení úrovně detailů (viz 3.2.11 Řízení úrovně detailů), které odřeže malé mraky a sníží velikost textur viditelným mrakům v dálce, a nakonec zapneme i metodu zjišťování viditelnosti (3.2.7 Zjišťování viditelnosti mraku), která ořeže mraky mimo náš pohledový jehlan.



Graf 4.4: Graf ukazující nárůst snímků za sekundu při zapnutí urychlovacích metod v programu. Narozdíl od předchozích grafů jsou mraky náhodně rozmístěny po scéně a nejsou všechny viditelné. LoD označuje Level of Detail – tzn. řízení úrovně detailů.

Z posledního grafu vidíme, že je rychlost vykreslování při scéně se 100 mraky (což je velmi blízko průměrné herní scéně) je se zapnutými urychlovacími metoda velmi dobrá a náš program může z hlediska rychlosti konkurovat komerčním aplikacím.

Rychlost ovšem není to jediné měřítko, kterým bychom měli hodnotit tuto aplikaci. Jelikož jde o vizualizační techniku pro hry, je také velmi důležitý vzhled a atraktivita pro hráče. Atraktivitu je samozřejmě velmi těžké kvantifikovat. Nechejme na každém čtenáři, aby si porovnal obrázky z hodnocených her v kapitole 2.3 Virtuální mraky – vypozorované vlastnosti s grafickými výstupy z naší aplikace uvedenými mj. v kapitole Příloha A - Grafické výstupy. Autor ze svého osobního úhlu pohledu považuje výstupy z této metody jako konkurenceschopné i hře Tom Clancy's H.A.W.X., ke které jsme se chtěli na počátku alespoň přiblížit.

5 Závěr

Cílem této diplomové práce bylo modelování a vykreslování objemových mraků pro herní aplikace. Jako priority jsme si stanovili rychlost vykreslování a atraktivitu pro hráče. Naše řešení nejprve vymodeluje a osvětlí všechny mraky v přípravné fázi, uloží je do 3D textury (datová reprezentace mraků jsou voxely), spočítá si pomocí *bounding boxu* imposter pro daný mrak, pokud tento imposter splní podmínky pro vykreslení, tak 3D texturu na GPU pomocí upravené *ray-casting* metody *real-time* osvětlí a vykreslí do 2D textury. Takto vykreslené mraky nanese dvoufázově na imposter a mrak finálně vykreslí.

Výše uvedené řešení se ukázalo jako dobře použitelné - mrak se vykreslí rychle, reálné osvětlení a vykreslovací technika ho dělá vizuálně atraktivním a samotný průlet vypadá velmi realisticky. V závěru by pak autor rád z této práce vyzdvihl především *real-time* osvětlovací techniku, která vznikla přímo pro tuto práci (dle dostupných informací není jinde publikovaná) a která dosahuje velmi dobrých výsledků. Dále stojí za povšimnutí samotný *ray-casting* s průletem a řízení úrovně detailů, které nastaví velikost textury v závislosti na velikosti na obrazovce, ale pouze do nějaké hranice a pomocí dvou průchodů *shadery* poté tuto texturu vyhladí a přizpůsobí reálné velikosti imposteru (i toto vylepšení, ani jeho mutaci nenašel autor jinde publikované).

Samozřejmě by bylo pokrytectví tvrdit, že je celý postup bez míst, která by nešla zlepšit. Jako tip na budoucí vylepšení tedy doporučujeme předpočítané osvětlení, jež by bylo dobré urychlit a zkrátit tak uživateli čekání při spouštění herní aplikace. Dalšími zajímavými náměty jsou do metody osvětlování v reálném čase zahrnout zastínění dalšími mraky, přidělat více typů mraků a rozdílný vítr ve výškových hladinách. Po určitých úpravách (obzvláště po urychlení *pre-process* osvětlování, které by se poté vždy jednou za čas přepočítalo), by také bylo možné simulovat dynamicky vývoj tvaru mraku v čase.

Autor věří, že řešení uvedená v této práci odpovídají moderním trendům a jsou použitelná v komerčních aplikacích, a že v této souvislosti tudíž byly splněny podmínky zadání.

Přehled zkratek

2D	– Dvou-rozměrné, rovina
3D	– Tří-rozměrné, prostor
3ds Max	– Autodesk 3ds Max, 3D grafický editor
AC	– Air Conflicts, počítačová hra
DVD	– Digital Versatile Disc, záznamové médium
CPU	– Central Processing Unit, centrální procesor počítače
FPS	– Frames Per Second, počet zobrazených snímků za sekundu
GLSL	– OpenGL Shading Language, jazyk pro programování na GPU
GPU	– Graphic Processing Unit, procesor grafické karty
GS	– Geometry shader, část programovatelné grafické pipeline
HLSL	– High-Level Shading Language, jazyk pro programování na GPU
HW	– Hardware, fyzické vybavení počítače
LoD	– Level of Detail, řízení úrovně detailů
MS	– Microsoft, firma produkující počítačový SW a HW
MSFS	– Microsoft Flight Simulator, počítačová hra
NURBS	– Non-Uniform Rational B-Spline, matematický model reprezentace křivek a ploch
p.p.	– Pre-Process, fáze probíhající pouze jednou před samotným vykreslováním scény
PS	– Pixel Shader, část programovatelné grafické pipeline
SW	– Software, programové vybavení počítače
TC HAWX	– Tom Clancy's HAWX, počítačová hra
VS	– Vertex Shader, část programovatelné grafické pipeline

Přehled obrázků

Obrázek 1.1: Ukázka výstupu našeho programu – mrak typu Cumulus. 10
Obrázek 2.1: Mrak typu cumulus nad městskou zástavbou. Fotografie pořízená 9. května 2010, 17:50 hod, Bratislava, Slovenská Republika..................................17
Obrázek 2.2: subLOGIC Flight Simulator 1, 1979. Pravděpodobně první legendární simulátor. (Havlík, 2010)..20
Obrázek 2.3: MS Flight Simulator 3.0, 1988. Jedno z prvních zobrazení mraků v počítačových hrách. (Havlík, 2010)...20
Obrázek 2.4: F-19 Stealth Figther, 1988. Průkopník vektorových mraků.............21
Obrázek 2.5: F-15 Strike Eagle II, 1989. Další zvýšení realističnosti oblohy, přidáním mlžného oparu. 21
Obrázek 2.6: MS Flight Simulator 4.0, 1991. Zobrazení mraků jako šestiúhelníků s různými stupni šedi. 21
Obrázek 2.7: MS Flight Simulator 5.0, 1993. Pro zobrazení mraků již použity textury. (Havlík, 2010)..21
Obrázek 2.8: Crimson Skies, 2000. Vysoko hodnocená hra s – na svou dobu – velmi dobrou grafikou. Mraky již tvořeny jako billboardy s nanesenou texturou. Bohužel se billboardy uhýbají kameře, takže jimi nelze proletět. 22
Obrázek 2.9: Ukázka zpracování mraků cumulus ve hře MS Flight Simulator X.24
Obrázek 2.10: Ukázka zpracování mraků cumulus ve hře Tom Clancy's HAWX. 24
Obrázek 2.11: Ukázka zpracování mraků cumulus ve hře Air Conflicts. 26
Obrázek 2.12: Obrázek ukazuje kombinaci textur 4 velikostí do výsledné textury. Šum na texturách je generován pomocí Perlinovy šumové funkce. Převzato z (Elias, et al., 1998). 29
Obrázek 2.13: Ukázka výstupu práce, kde je vykreslování mraků maximálně zjednodušeno - v podstatě se jedná jen o vykreslování několika plochých textur nad sebou. Převzato z (Roden, et al., 2005)..31
Obrázek 2.14: Ukázka výstupu programu M. Harrise. Jeho metoda využívá multiple forward scatteringu a anisotropického scatteringu (Harris, 2002). 33
Obrázek 2.15: Grafické znázornění zobrazovací rovnice. Převzato a upraveno z (Justin, et al., 2009)..34
Obrázek 2.16: Ukázka ray-castingu neboli vrhání paprsku objemem. Paprsek spočítá pixel na promítacím plátu pomocí hodnot, které nasbírá při průchodu množinou dat. Převzato a upraveno z (Weiskopf, 2004). 35
Obrázek 2.17: Čelní (přivrácené) strany jednotkové krychle, kde barva udává kudy paprsek do objemu vstupuje (*rayIn*). (Purchart, 2009)................................36
Obrázek 2.18: Zadní strany jednotkové krychle, kde barva udává kudy paprsek z objemu vystupuje (*rayOut*). (Purchart, 2009)...36
Obrázek 3.1: Jeden z nejznámějších stíhacích letounů 2. světové války 51D Mustang - nárys, bokorys, půdorys. (Owly, 2005) 42
Obrázek 3.2: Modelování letadla, krok jedna. Vytvoření 3 osově zarovnaných ploch a nanesení skici. 42

Obrázek 3.3: Modelování letadla, krok dva. Položení počáteční plochy na motor letadla a přizpůsobení jeho tvaru. 42
Obrázek 3.4: Modelování letadla, krok tři. Vymodelování základní části levé poloviny trupu a jeho zrcadlení modifikátorem Symetry. 43
Obrázek 3.5: Modelování letadla, krok čtyři. Přidání zadního křídla....................43
Obrázek 3.6: Modelování letadla, krok pět. Finální model, na kterém vidíme jednotlivé trojúhelníky. V levé části je poté modifikátorem UVW Unwrap ukázáno mapování textury na náš model. 43
Obrázek 3.7: Tvorba textury SkyBoxu. Krok jedna, vyplnění oblohy přechodem. 45
Obrázek 3.8: Tvorba textury SkyBoxu. Krok dva, přidání oparu..........................45
Obrázek 3.9: Tvorba textury SkyBoxu. Krok tři, nepravidelný opar. 46
Obrázek 3.10: Tvorba textury SkyBoxu. Krok čtyři, rozmazání vršků oparu.......46
Obrázek 3.11: Tvorba textury SkyBoxu. Krok pět, zaoblení. 46
Obrázek 3.12: Tvorba textury SkyBoxu. Krok šest, kontrola návazností okrajů. .46
Obrázek 3.13: Textura slunce. V reálu není v textuře pozadí modré, ale průhledné. To by ale jaksi nebylo na papíře vidět. 48
Obrázek 3.14: Programovatelná pipeline. Vstupem Vertex Shaderu jsou vrcholy (Vertices), výstupem jsou transformované vrcholy, které se poté rasterizují a výsledné pixely (Pixels) putují do Pixel Shaderu, kde se pro každý spočítá jeho barva. Světle modře jsou na obrázku vyznačeny části pipeline, které budeme přímo programovat. (Whitaker, 2009)..50
Obrázek 3.15: Neosvětlený mrak, který pouze ukazuje základní tvar, vygenerovaný naším programem...53
Obrázek 3.16: Ukázka průletu mrakem. Textura se dymanicky mění tak, jak prolétáváme objemem...58
Obrázek 3.17: Ukázka imposteru (červeně ohraničený billboard), vykresleného před 3D objektem automobilu (žlutě ohraničený). Zeleně jsou ukázány spojnice mezi kamerou a vrcholy bounding boxu. (Davis, 2006).......................................59
Obrázek 3.18: Zjišťování viditelnosti sedmi mraků. Tři červené jsou mimo pozorovací úhel a nevykreslují se, čtyři zelené se vykreslují. 65
Obrázek 3.19: Úsečka proložená mezi voxelem V a sluncem. T_{min} je poté průsečík této úsečky s objemovou krychlí. 69
Obrázek 3.20: Převedení našeho případu do 2D, sloužící pro snažší ilustraci. Průsečíky přímky proložené sluncem a pixelem V se všemi 4 přímkami ohraničujícími čtverec..69
Obrázek 3.21: Ukázka z programu, demonstrující řízení úrovně detailů. Je zde velmi pěkně vidět, že textura u vzdálených mraků bohatě postačuje. 77
Obrázek Příloha C. 1: Ukázka instalátoru našeho programu. Pomocí instrukcí provede uživatele instalací jak samotného programu, tak i všech dalších potřebných knihoven a aplikací...100
Obrázek Příloha C. 2: Ovládání našeho programu..101

Přehled tabulek

Tabulka 2.1: Druhy mraků, jejich latinský i český název, zkratka a výška, ve které se v mírném podnebném pásu vyskytují. (Fillinger, et al., 2010)..........................16
Tabulka 2.2: Vypozorované vlastnosti oblaků a oblohy ve hrách Microsoft Flight Simulator X, Tom Clancy's HAWX a Air Conflicts...25
Tabulka 4.1: Hardwarové a softwarové vybavení testovacího počítače................78
Tabulka 4.2: Výsledky měření v programu 3DMark 2006....................................78
Tabulka Příloha C.1: Možné nastavení argumentů pro pokročilého uživatele....102

Přehled grafů

Graf 4.1: Na tomto grafu je patrné urychlení naší aplikace, pokud použijeme v řízení úrovně detailů k namísto logaritmické metody saturaci. 79
Graf 4.2: Rozdíl v rychlosti vykreslování s a bez osvětlování v reálném čase (real-time). Ukazuje se, že námi navržená metoda real-time osvětlování nezpomaluje výslednou aplikaci. 80
Graf 4.3: Čas potřebný pro spuštění aplikace s a bez pre-process osvětlení. Při vyšším počtu mraků je vidět markantní nárůst v čase s p.p. osvětlením. 81
Graf 4.4: Graf ukazující nárůst snímků za sekundu při zapnutí urychlovacích metod v programu. Narozdíl od předchozích grafů jsou mraky náhodně rozmístěny po scéně a nejsou všechny viditelné. LoD označuje Level of Detail – tzn. řízení úrovně detailů. 82

Přehled výpisů programů

Výpis 3.1: Obsah textového souboru textures.txt. (Rstralberg, 2008)...................47
Výpis 3.2: Obsah textového souboru create_sky.cmd. (Rstralberg, 2008)............47
Výpis 3.3: Příprava dat mraku. Vytváříme polokouli, kterou po obvodu narušíme náhodným šumem. To je v podstatě základní tvar mraku cumulus. (Pro větší přehlednost klasický zápis mocnin.)..52
Výpis 3.4: Pixel shader vykreslující souřadnici krychle o velikosti hrany 1 jako barvu. 55
Výpis 3.5: Zjištění souřadnic imposteru (resp. billboardu, na ketrý se později nanese dynamická textura), pomocí projekce bounding boxu do obrazovkového souřadného systému, vybrání správných bodů, tak, aby zakrývaly celý bounding box a vrácení těchto výsledných hodnot zpět do původního souřadného systému. (Hayward, 2008) 60
Výpis 3.6: Vykreslení 3D objektu do textury lowRes. 62
Výpis 3.7: Metoda, která na základě normálového vektoru imposteru a vektoru mezi kamerou a středem imposteru určí, jestli se poloha pozorovatele změnila natolik, že se musí vykreslit objekt znovu, anebo můžeme používat stále již vygenerovanou texturu. 63
Výpis 3.8: Testování úhlu svírajícího vektor od kamery k imposteru s pohledovým vektorem kamery. 64
Výpis 3.9: Osvětlovací metoda v pre-processingu. Pro každý voxel V vyšleme paprsek směrem ke slunci a pomocí této metody spočítáme jeho barvu. 67
Výpis 3.10: Metoda počítá vzdálenost voxelu V na pozici posV s průsečíkem tMin na hraně krychle mraku. Průsečík je na úsečce mezi voxelem a sluncem. ...68
Výpis 3.11: Pokud je spuštěný pohyb, při každém zavolání této funkce posuneme slunce o kousek po kružnici...71
Výpis 3.12: Uložení do pole vzdáleností každého mraku od kamery a následné srovnání tohoto pole...73
Výpis 3.13: Vykreslování mraků od nejvzdálenějšího k nejbližšímu. 73
Výpis 3.14: Fast Bucked Sort - neboli urychlený „vědrový“ algoritmus. Jeden z nejefektivnějších algoritmů řazení. Převzato z (Vckicks, 2008). 74
Výpis 3.15: Výpočet velikosti promítacího plátu mraku......................................76
Výpis 3.16: Původní pomalá funkce pro nalezení nejbližší mocniny dvou pomocí logaritmů v standardní matematické knihovně...76
Výpis 3.17: Jedna z nejrychlejších metod pro nalezení nejbližší mocniny dvou. Převzato z (Neoztar, 2004). 77

Přehled vzorců

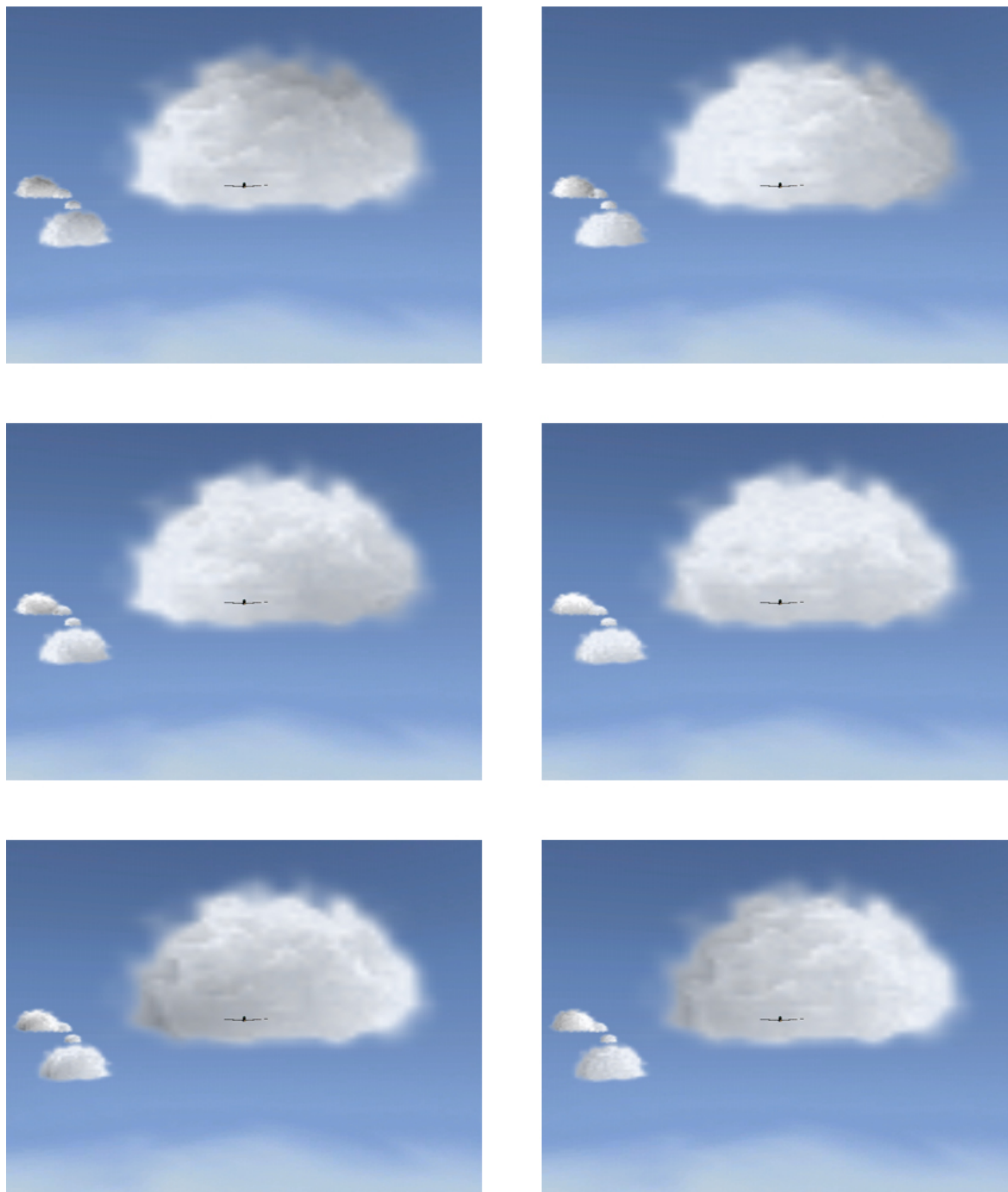
Vzorec 2.1: Albedo, nebo také koeficient rozptylu světla. 19
Vzorec 2.2: Rayleighova fázová funkce pro aproximaci průchodu světla částicí - α představuje úhel mezi zdrojem světla a pozorovatelem.(Harris, 2002) 19
Vzorec 2.3: Přechodová pravidla pro simulaci tvorby mraku. (Dobashi, et al., 2000) 30
Vzorec 2.4: Wyvillova kubická funkce. (Man, 2007) 30
Vzorec 2.5: Zobrazovací rovnice – základ pro metodu sledování paprsku objemem, přičemž λ je vlnová délka světla, t je čas, L_0 je výsledné světlo, L_e je vyzářené světlo, $\int_\Omega$ je integrál přes polokouli, f_r je odrazivost, L_i vstupující světlo a $-\omega \cdot n$ je útlum vstupujícího světla. (Justin, et al., 2009) 34
Vzorec 2.6: Směr průchodu paprsku pro daný pixel. RayOut a rayIn jsou pozice, prezentované pomocí barevné informace na intervalu $\langle \mathbf{0}; \mathbf{1} \rangle$. (Purchart, 2009) 36
Vzorec 2.7: Výpočet barvy a průhlednosti. Akumulace do výsledného vektoru v každém kroku průchodu množinou dat. (Purchart, 2009) 36
Vzorec 3.1: Posun zadní kamery o úhlopříčku maximálně velkého mraku plus nějaké malé navýšení, pro zamezení aritmetických chyb. 56
Vzorec 3.2: Konstanta zajišťující snížení nejvyššího rozlišení mraku. 75

Citovaná literatura

Anonym. 2010. DevMaster's Game and Graphics Engines Database. *DevMaster.* [Online] 2010. [Cited: 1 June 2010.] http://www.devmaster.net/engines/.
Blinn, James. 1982. *Light Reflection Functions for Simulation of Clouds and Dusty Surfaces.* s.l. : SIGGRAPH, 1982.
Davis, Ashley. 2006. Dynamic 2D Imposters: A Simple, Efficient DirectX 9 Implementation. *Gamasutra.* [Online] 5 January 2006. [Cited: 1 May 2010.] http://www.gamasutra.com/features/20060105/davis_01.shtml.
Dobashi, Yoshinori, et al. 2000. *A Simple, Efficient Method for Realistic Animation of Clouds.* New Orleans, Louisiana USA : ACM SIGGRAPH, 2000.
Elias, Hugo and Fairclough, Matt. 1998. Cloud Cover. [Online] 21 October 1998. [Cited: 18 June 2010.] http://freespace.virgin.net/hugo.elias/models/m_clouds.htm.
Fillinger, Zdeněk and Valový, Dalibor. 2010. Formování oblačnosti - typy mraků. *Euromarina.* [Online] 2010. [Cited: June 3, 2010.] http://www.euromarina.cz/pocasi/meteorologie/formovani-oblacnosti.htm.
Futuremark Corporation. 2010. 3DMark06. *PC Benchmarks.* [Online] 2010. [Cited: 14 July 2010.] http://service.futuremark.com/hardware/.
Geršl, Vladimír. 2008. *Game design, aneb jak navrhovat hry.* Plzeň : Západočeská univerzita v Plzni, 2008.
Harris, Mark. 2002. *Real-Time Cloud Rendering for Games.* s.l. : Game Developers Conference, 2002.
Havlík, Josef. 2010. History of Microsoft Flight Simulator. [Online] 2010. [Cited: 22 June 2010.] http://www.volny.cz/havlikjosef/historyenglish.htm.
Hayward, Kyle. 2008. Reflections with Billboard Impostors. *Graphics Runner.* [Online] 17 April 2008. [Cited: 29 April 2009.] http://graphicsrunner.blogspot.com/2008/04/reflections-with-billboard-impostors.html.
Jankovič, Jozef and al., et. 2006. Kumulus. *Wikipedia.org.* [Online] 19 June 2006. [Cited: 2 June 2010.] http://sk.wikipedia.org/wiki/Kumulus.
Justin, Talbot and Whiteman, D. 2009. Rendering equation. *Wikipedia.org.* [Online] December 27, 2009. [Cited: July 1, 2010.] http://en.wikipedia.org/wiki/Rendering_equation.
Kajiya, James and Herzen, Brian. 1984. *Ray tracing volume densities.* New York, NY, USA : ACM, 1984. 0097-8930.
Kučera, Vít. 2007. *Zobrazování mraků v reálném čase.* Brno : Vysoké učení technické, 2007.
Legacy DDS Utilities. 2007. [Online] nVidia, December 2007. [Cited: 29 June 2010.] http://developer.nvidia.com/object/dds_utilities_legacy.html.
Lička, Karel and al, et. 2010. Rosný bod. *Wikipedia.org.* [Online] 23 May 2010. [Cited: 2 June 2010.] http://cs.wikipedia.org/wiki/Rosn%C3%BD_bod.
Man, Petr. 2007. *Modelování a zobrazování mraků.* Praha : České vysoké učení technické, 2007.

McLinden, Chris. 1999. Mie Scattering. *The Web page of Chris McLinden.* [Online] Department of Earth System Science, University of California, Irvine, 22 July 1999. [Cited: 1 July 2010.] http://www.ess.uci.edu/~cmclinden/link/xx/node19.html.
Neoztar. 2004. Nearest power of 2. *GameDev.* [Online] 6 July 2004. [Cited: 3 June 2009.] http://www.gamedev.net/community/forums/topic.asp?topic_id=229831.
Ondřej, Jan. 2007. *Analytický model pro rozptyl světla v mlze.* Praha : České vysoké učení technické, 2007.
Owly, Ken. 2005. North_American_P-51D_Mustang_line_drawing.png. [Online] June 2005. [Cited: 12 July 2010.] http://www.everystockphoto.com/photo.php?imageId=1744193.
Perlin, Ken. 1999. Making Noise. [Online] 9 December 1999. [Cited: 18 July 2010.] http://www.noisemachine.com/talk1/index.html.
Prokop, Zděněk. 2007. *Modelování mraků pro VR aplikace.* Plzeň : Katedra informatiky a výpočetní techniky, Fakulta aplikovaných věd, Západočeská univerzita, 2007.
Purchart, Václav. 2009. *GPU based – Direct volume rendering.* Plzeň : Katedra informatiky a výpočetní techniky, Fakulta aplikovaných věd, Západočeská univerzita, 2009.
Roden, Timothy and Parberry, Ian. 2005. *Clouds and Stars: Efficient Real-Time Procedural Sky.* Valencia, Spain : ACE, 2005.
Rstralberg. 2008. Making a skybox using Terragen. *Wiki.com.* [Online] Leadwerks, July 2008. [Cited: 10 June 2010.] http://www.leadwerks.com/wiki/index.php?title=Making_a_skybox_using_Terragen.
TextureCube Class. 2010. *MSDN help.* [Online] Microsoft, 2010. [Cited: 14 July 2010.] http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.texturecube.aspx.
Vaněček, Petr and Hanák, Ivo. 2008. Přednášky z předmětu Grafická rozhraní a GPU. *Centre of Computer Graphics and Visualization.* [Online] Katedra informatiky a výpočetní techniky, Fakulta aplikovaných věd, Západočeská univerzita, October 7, 2008. [Cited: June 10, 2010.] http://herakles.zcu.cz/education/Grg.
Vckicks. 2008. Fast Bucket Sort. *Daniweb.* [Online] 17 December 2008. [Cited: 2 July 2010.] http://www.daniweb.com/code/snippet217213.html.
Weiskopf, Daniel. 2004. GPU-Based Ray Casting. *SIGGRAPH 2004 Course.* [Online] 2004. [Cited: June 28, 2010.] http://old.vrvis.at/via/resources/course-volgraphics-2004/slides/SIGGRAPH04_Part02.pdf.
Whitaker, R. 2009. Introduction to Shaders. *RB Whitaker's Wiki.* [Online] 19 March 2009. [Cited: 10 July 2010.] http://rbwhitaker.wikidot.com/intro-to-shaders.

Příloha A - Grafické výstupy



Obrázek Příloha A.1: Ukázka osvětlení mraků v reálném čase.

Obrázek Příloha A.2: Ukázka průletu letadla mrakem. Všimněte si, jak se mrak mění a řídne.

Příloha B - Hlavní pixel shader

```
1 float4 PixelShaderFunctionLR(VertexShaderOutput input) : COLOR0
2 {
3     // Vrcholy objemové krychle převedeme do world souřadnic.
4     A = mul(float4(0, 0, 0, 1), World);
5     B = mul(float4(1, 0, 0, 1), World);
6     C = mul(float4(1, 0, 1, 1), World);
7     D = mul(float4(0, 0, 1, 1), World);
8     E = mul(float4(0, 1, 0, 1), World);
9     F = mul(float4(1, 1, 0, 1), World);
10    G = mul(float4(1, 1, 1, 1), World);
11    H = mul(float4(0, 1, 1, 1), World);
12
13    // Nyní se přeneseme z homogenního do projektivního
      prostoru.
14    A.xyz /= A.w;
15    A.w = 1;
16    B.xyz /= B.w;
17    B.w = 1;
18    C.xyz /= C.w;
19    C.w = 1;
20    D.xyz /= D.w;
21    D.w = 1;
22    E.xyz /= E.w;
23    E.w = 1;
24    F.xyz /= F.w;
25    F.w = 1;
26    G.xyz /= G.w;
27    G.w = 1;
28    H.xyz /= H.w;
29    H.w = 1;
30
31    // Snižování alfy při přiblížení k mraku
32    float alphaMisting = 1;
33    // Dvojnásobek velikosti mraku
34    float MistingDistance = World[0][0] * 2;
35
36    // Pozice středu mraku
37    float3 cloudPos;
38    cloudPos.x = World[3][0] + MistingDistance / 2;
39    cloudPos.y = World[3][1] + MistingDistance / 2;
40    cloudPos.z = World[3][2] + MistingDistance / 2;
41
42    // Vzdálenost středu mraku a oka pozorovatele
43    float distVal = length(cloudPos.xyz - EyePosition);
44    // Pokud jsem uvnitř zony pro snižování průhlednosti
45    if(distVal <= MistingDistance)
46          alphaMisting = distVal / MistingDistance;
47
48    // Pozice nepronásobená World maticí - tzn., pozice v
      krychli <0;1>
49    float4 wPos = input.wPos;
```

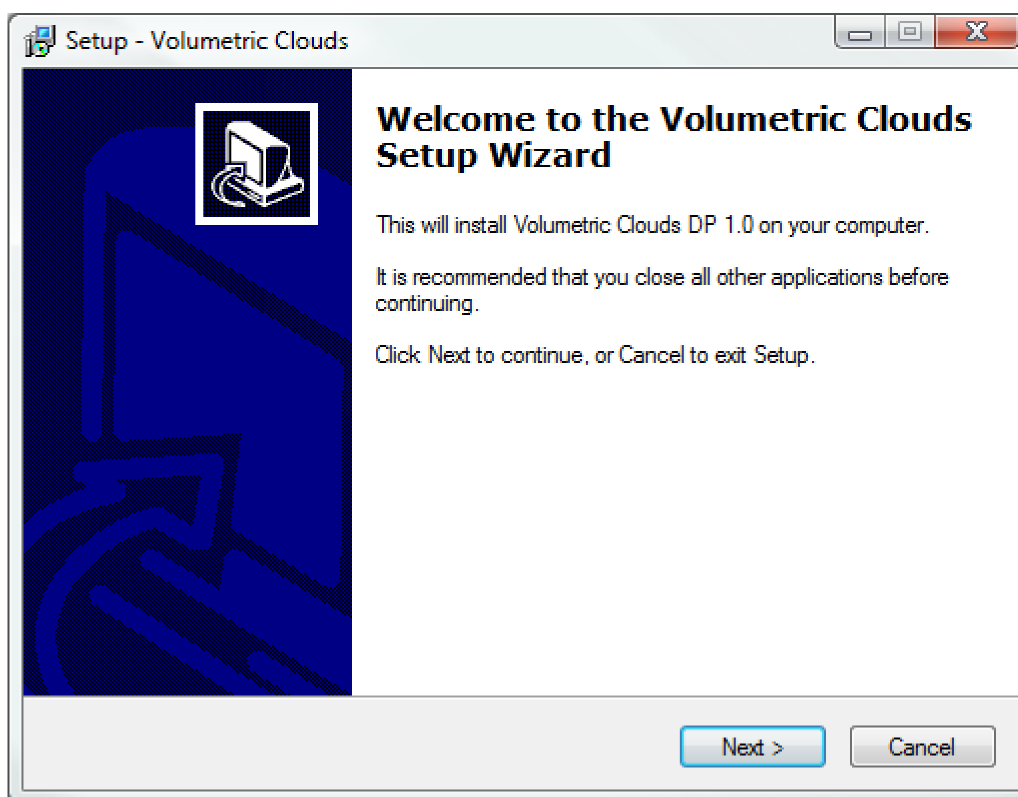
```
50     // Místo kde paprsek do objemu vstupuje.
51     // Načteme souřadnice z textury přivrácené strany.
52     float3 rayIn = tex2Dlod(sFirst, float4(wPos.x, -wPos.y, 0 ,
       0)).rgb;
53     // Místo kde paprsek z objemu vystupuje.
54     // Načteme souřadnice z textury odvrácené strany.
55     float3 rayOut = tex2Dlod(sSecond, float4(wPos.x, -wPos.y, 0,
       0)).rgb;
56     float3 dir = normalize(rayOut.xyz - rayIn.xyz);
57
58     int end = 0;
59
60     // Pevný počet kroků.
61     int stepCount = 100;
62     // Aktuální pozici na načátku nastavíme na místo vstupu
       paprsku do objemu.
63     float3 pos = rayIn;
64     // Empiricky zjištěná ideální velikost voxelu.
65     float  voxel = 1.54;
66     // Vektor pevné velikosti, o který se vždy posuneme.
67     float3 step = (voxel / stepCount) * dir;
68
69     // Hranice, za kterými již objem neprocházíme.
70     float min = 0.02;
71     float max = 0.98;
72
73     // Vzdálenost jakou je hráč zanořen v mraku,
74     float shift = 0;
75
76     // Vršek a spodek mraku - jelikož mrak nedosahuje k okrajům
       objemu.
77     float top = 1 - shiftCloudColision;
78     float bottom = 0 + shiftCloudColision;
79
80     // Pokud je hráč uvnitř objemu spočti jeho pozici a o tolik
       posuň počátek vykreslování.
81     if ((PlayerinCloud[3][0] < top && PlayerinCloud[3][0] >
       bottom) && (PlayerinCloud[3][1] < top && PlayerinCloud[3][1]
       > bottom) && (PlayerinCloud[3][2] < top &&
       PlayerinCloud[3][2] > bottom))
82     {
83           shift = length(rayIn - PlayerinCloud[3].xyz);
84           pos += shift * dir;
85     }
86
87     // Výsledná barva a alfa průhlednost (rgb je stejné, proto
       je barva zastoupena jednou složkou).
88     float2 value = 0;
89     // Zdroj.
90     float4 src = 0;
91
92     // Pokud začínáme, či končíme mimo objem, skonči.
93     if (rayIn.x < min && rayIn.y < min && rayIn.z < min)
       discard;
```

```
94    if (rayOut.x < min && rayOut.y < min && rayOut.z < min)
      discard;
95
96    float4 result = float4(0,0,0,0);
97    float lastIntensity = 0;
98    // Akumulační proměnná pro barvu a počítadlo projitých
      barev.
99    float tmpColor = 0;
100   int tmpColorCounter = 0;
101
102   for (int i=0; i<stepCount; i++) {
103   // Pokud je end == 0, pokračuj ve výpočtech. Jinak zastav
      cyklus.
104   if (end == 0)
105   {
106     // Posuneme se o krok po směru paprsku.
107     pos += step;
108     // Pokud jsme mimo objem skonči, jinak pokračuj.
109     if (pos.x < min || pos.y < min || pos.z < min);
110     else
111       if (pos.x > max || pos.y > max || pos.z > max);
112       else
113         {
114         // Načetme si barvu a průhlednost daného voxelu z 3D
            textury.
115         value.xy = tex3Dlod(sVolume, float4(pos, 0)).ba;
116         // Načteme si barvu a průhlednost předchozího voxelu.
117         float2 oldValue = tex3Dlod(sVolume, float4((pos -
            step), 0)).ba;
118         // Tato proměnná vyjadřuje, zdali se aktuální voxel na
            vrchu objemu.
119         float onTop = 1;
120
121         // Pokud byla o krok dříve černá barva (okraj mraku).
122         if (oldValue.y < 0.1)
123               onTop = 0.8f;
124
125         // Vypočítáme si vzdálenost aktuálního voxelu V a
            průsečíku paprsku s bounding boxem kolem mraku.
            Paprsek míří z V do slunce.
126         float length = dynamicLight(pos);
127         // Pokud je vzdálenost větší, než maximální velikost
            mraku, nastavíme ji na tuto maximální velikost.
128         int maxCloud = 70;
129         if (length > maxCloud)
130               length = maxCloud;
131
132         // Počáteční nástřel barvy.
133         float col = (1 - (length/maxCloud));
134         float epsCol = 0.1f;
135
136         // Úpravy výsledné barvy na základě nástřelu. Tmavá
            místa zesvětlujeme víc, než světlá.
137         if (col > value.x - epsCol)
138               value.x *= 1.5f + col;
```

```
139         else
140               value.x *= 0.3f + col;
141
142         // Ztmavíme ještě voxel, pokud je navrchu, abychom
            dodali plastičnost.
143         value.x *= onTop;
144
145         // Výsledná alfa vždy přičte
146         result.a += (value.y * 5) / (float)stepCount;
147
148         // K akumulátoru barvi přičti již osvětlenou hodnotu a
            navyš počítadlo.
149         tmpColor += value.x;
150         tmpColorCounter += 1;
151
152         // Pokud je alfa kanál již téměř neprůhledný, skonči.
            Další voxely již stejně skrz objem nejsou téměř vidět.
153         if (result.a > 0.90f)
154               end = 1;
155       }
156     }
157   }
158
159   // Pokud je naakumulovaná barva jiná než černá.
160   if (tmpColor > 0)
161   {
162         // Pokud není pixel zcela průhledný, pokračuj. Jinak
            zahoď.
163         if (result.a > 0)
164         {
165         // Výsledná barva = naakumulovaná barva / počet kroků,
            kdy jsme akumulovali.
166         float color = tmpColor / tmpColorCounter;
167         tmpColor = 0.0;
168         tmpColorCounter = 0;
169         if (alphaMisting < 0.8)
170               alphaMisting = 0.8;
171         // Alfa kanál nakonec ještě pronásobíme alphaMisting,
            která zprůhledňuje mrak, pokud je hráč blízko mraku.
172         return float4(color, color, color, result.a *
            alphaMisting);
173   }
174   else discard;
175   }
176   else discard;
177   }
```

Příloha C - Uživatelská příručka

Instalace probíhá běžným způsobem – po spuštění souboru *setup.exe* z přiloženého DVD se otevře průvodce instalací (viz Obrázek Příloha C.1), který nabídne mimo samotného programu (*Volumetric Clouds*) automaticky i instalaci všech balíčků a knihoven potřebných pro úspěšné spuštění.



Obrázek Příloha C.1: Ukázka instalátoru našeho programu. Pomocí instrukcí provede uživatele instalací jak samotného programu, tak i všech dalších potřebných knihoven a aplikací.

V nově vytvořeném adresáři s programem *Volumetric Clouds* nalezneme nyní spustitelný soubor *volumetric_clouds.exe* Pro běžného uživatele stačí základní ovládání, které je blíže popsáno na Obrázek Příloha C.2.

Obrázek Příloha C.2: Ovládání našeho programu.

Jak vidíme ovládání je intuitivní a připomíná arkádové počítačové hry typu Air Conflicts (viz 2.3 Virtuální mraky – vypozorované vlastnosti). Celý program je ovšem zaměřený na pozdější využití v (komerční) praxi, a je tedy modifikovatelný velkou řadou interních nastavení. Každý grafik či designer si bude moci zkorigovat vzhled mraků přímo podle grafických potřeb konkrétní hry. Program tedy nabízí možnost vyzkoušení změny některých z těchto modifikátorů a nastavení pro pokročilého uživatele. Jedná se ovšem stále o zlomek vnitřních nastavení, který je jednoduše modifikovatelný a nepotřebuje hlubokou znalost problému. Probíhá zadáváním tzv. argumentů pro spuštění, které nalezneme v Tabulka C.1. Argumentů je celkem šest a musí se zadávat postupně. Nelze tedy například zadat *Počet mraků*, bez zadání předchozích tří. Pokud některý z těchto argumentů zadáme v nesprávném tvaru, spustí se program v základním nastavení.

Číslo arg.	Název nastavení	Běžná hodnota	Možné hodnoty	Vysvětlení
1	Typ pohledu kamery	t	t	*Third person look* – Typ pohledu ze třetí osoby. Imposter se při průletu přichytí k čumáku letadla.
			f	*First person look* – Pohled z první osoby. Imposter se přichytí ke kameře.
2	Typ osvětlení	c	c	*Combinated* – Kombinované osvětlení v pre-process fázi a v reálním čase.
			r	*Real-time* – Osvětlení pouze v reálném čase. Slouží pouze pro testování!
3	Způsob výpisu FPS	s	s	*Synchronized* – Dosažená snímková frekvence synchronně korigovaná na max. 60 FPS.
			n	*Non-synchronized* – Nekorigovaná snímková frekvence (bez maxima).
4	Poč. mraků	30	Celé číslo > 1	Počet mraků na obloze.
5	Maximální velikost plátu	64	Celé číslo > 1	Maximální rozlišení textury promítacího plátu při přiblížení k mraku.
6	Minimální velikost	8	Celé číslo > 1	Min. velikost mraku v pixelech, který se vykreslí.

Tabulka Příloha C.1: Možné nastavení argumentů pro pokročilého uživatele.