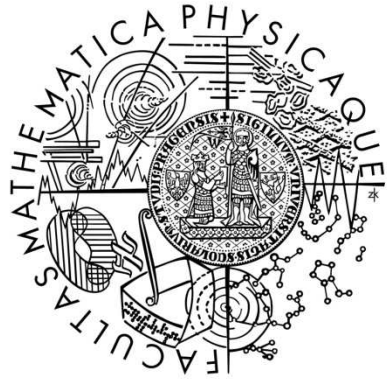
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



David Skupien

Experimentální systém pro vržené stíny

Katedra software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Petr Kmoch
Studijní program: Informatika, Obecná informatika

2009

Na tomto místě bych rád poděkoval vedoucímu práce Mgr. Petru Kmochovi za odborné rady a vstřícný přístup v celém průběhu jejího vzniku. Dále děkuji své přítelkyni a rodině za trpělivost, kterou se mnou měli i v těch nejvíce stresujících chvílích.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 14.7.2009 David Skupien

Obsah

1 Úvod **7**
- 1.1 Interaktivní vs. neinteraktivní zobrazování 7
- 1.2 Motivace – kdo potřebuje realistické stíny? 8
- 1.3 Cíle práce 8

2 Zobrazování stínů **9**
- 2.1 Světelné zdroje 9
- 2.2 Stíny 10
- 2.3 Přehled technik generování stínů 11

3 Stínová tělesa a plošný příjemce **14**
- 3.1 Metoda plošného příjemce 14
- 3.2 Metoda stínových těles 15

4 Implementace **20**
- 4.1 Prostředí a použité knihovny 20
- 4.2 Preprocesor 21
- 4.3 Testovací prostředí 21
- 4.4 Plošný příjemce 22
- 4.5 Stínová tělesa 22
- 4.5 Testy a výsledky. 25

5 Závěr **32**
- 5.1 Shrnutí 32
- 5.2 Splnění cílů 32
- 5.3 Diskuze a směr další práce 33

Literatura **34**

A Obsah CD **35**

B Uživatelská příručka **36**
B.1 Systémové požadavky 36
B.2 Instalace 36
B.3 Spuštění aplikace 37
B.4 Ovládání 37
B.3 Konfigurace pomocí vstupního XML souboru 37

C Přidávání stínových algoritmů **41**
C.1 Základní vlastnosti 41
C.2 Funkce Run() 42
C.3 Funkce Init() 42
C.4 Funkce SetupShaders() 43

Název práce: Experimentální systém pro vržené stíny
Autor: David Skupien
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Petr Kmoch
e-mail vedoucího: petr.kmoch@mff.cuni.cz

Abstrakt: Vykreslování vržených stínů patří již od zrodu grafiky jako takové k základním metodám navození atmosféry a dojmu realističnosti. Je to velmi těžká úloha, kterou v minulosti, ale i v současnosti řeší specializované stroje v řádech minut na jeden snímek. Díky rychlému rozvoji specializovaného hardwaru se stala zvládnutelnou i pro oblast *interaktivní grafiky*, a posunula se od jednoduchých geometrických náhražek na rovné podložce až k velmi přesvědčivým imitacím stínů. Cílem této práce je vytvořit grafické prostředí, ve kterém se dají vizuálně i výkonově porovnávat různé metody řešící zobrazování vržených stínů, a následně některé z nich implementovat. Vybrán byl jednoduchý algoritmus pro zobrazování stínů na *plošné příjemce* a pokročilý algoritmus *stínových těles*. Oba jsou implementovány v základní a rozšířené variantě, kde je jejich výkon v rozmezí desítek až tisíců snímků za sekundu.
Klíčová slova: programování GPU, fotorealistické zobrazování, stínová tělesa, vržené stíny

Title: Experimental System for Shadowing Algorithms
Author: David Skupien
Department: Department of software and computer science education
Supervisor: Mgr. Petr Kmoch
Supervisor's e-mail address: petr.kmoch@mff.cuni.cz

Abstract: From the advent of computer graphics is rendering of cast shadows one of the basic methods to evoke an atmosphere and give an observer a realistic impression. It has been a very difficult task solved by specialized machines in the order of minutes not only in the past but also in the present. However, thanks to rapid progress of dedicated graphics hardware, it is easy now to master it even for *interactive graphics*. It has moved from simple geometric excuses on planar receivers to convincing shadow imitations. The aim of this thesis is to develop a graphical environment capable of comparing visual and efficiency properties of different methods specialized for shadow casting and subsequently implement some of them. Two algorithms have been chosen. The first one is a simple algorithm specialized for rendering *shadows on planar receivers*, the second one is an advanced algorithm called *shadow volumes*. Both are implemented in a basic and enhanced version and capable of achieving rates from tens to thousands of FPS.
Keywords: GPU programming, photorealistic rendering, shadow volumes, cast shadows

Kapitola 1

Úvod

Jedním z hlavních důvodů vzniku počítačové 3D grafiky bylo přání co nejvěrněji modelovat svět, který nás obklopuje. Dnes se rozděluje do mnoha rozdílných odvětví, ale věrohodné (*fotorealistické)* zobrazování je stále tím hlavním. Základem všeho je světlo. Dalo by se říci, že pokud pomineme neméně důležité fyzikální vlastnosti, jako jsou pohyb a deformace, jediné a zásadní, co počítačová 3D grafika řeší, je chování světla. Zde se hovoří o reakci pevných materiálů, kapalin a plynů, které světlo různě odrážejí, propouští nebo pohlcují. Tato práce se zaměřuje na simulaci jedné z jeho vlastností – částečnou nebo úplnou nepřítomnost na nějakém místě scény.

1.1 Interaktivní vs. neinteraktivní zobrazování

Počítačová grafika je dnes nedílnou součástí filmového průmyslu. Proč je animace v dnešních filmech mnohem realističtější než u počítačových her? Odpověď je jednoduchá – čas. Film má dnes standardně 25 snímků za sekundu (FPS[1]). Pokud byl nějaký z nich obohacen o animaci, čas na jeho renderování není z pravidla tolik omezen. Pro každý snímek jsou k dispozici řádově minuty procesorového času na velmi výkonných strojích. Případně u fotografií nemusí být výjimkou i hodiny.

V *interaktivním světě* je situace zcela odlišná. Minimální hodnota FPS[1], kdy lidské oko není schopno rozeznat jednotlivé snímky, je zpravidla 25. Avšak u některých druhů počítačových her (například FPS[2]) je pro plynulost požadováno 60 snímků a více. Navíc se všechny musí kompletně zpracovat a zobrazit v reálném čase.

---

[1] **FPS** = **F**rames **P**er **S**econd (snímky za vteřinu)

[2] Zde jde o shodu zkratky, ale význam je **F**irst-**p**erson **S**hooter (podžánr akčních PC her)

1.2 Motivace – kdo potřebuje realistické stíny?

Proč se zabývat stíny? Stín je veličina, na kterou jsme podvědomě zvyklí a bez které pro nás nebude téměř žádná scéna vypadat realisticky – čím je stín kvalitnější, tím je kvalitnější i zážitek z pohledu na ni. Některé hry jsou pro svůj efekt na stínech více či méně založené. Za zmínku stojí například Doom III (obrázek 1.1) nebo F.E.A.R. Proto ve všech scénách, kde se kamera pohybuje blízko nějakých objektů, které jsou osvětleny alespoň jedním světelným zdrojem, by stíny v žádném případě neměly chybět už jen kvůli jejich nutnosti pro rozeznávání polohy objektů pozorovatelem.

Hlavním problémem veškerých *interaktivních* aplikací je potřeba jejich okamžité reakce na změnu libovolných parametrů scény – stíny se hýbou v závislosti na pohybu těles a světel. Jak uvidíme v následujících kapitolách, s kvalitou rostou rapidně i nároky na výkon počítače.

Obrázek 1.1: Ukázka stínů ze hry Doom III.

1.3 Cíle práce

Má práce se zaměřuje na splnění následujících cílů:

1. Prostudovat současné techniky pro zobrazování stínů v reálném čase.

2. Implementovat základní verze některých z těchto technik, následně je po vizuální stránce vylepšit.

3. Vytvořit testovací platformu, do které bude možnost algoritmy přidávat, vyvíjet je a testovat třetí stranou.

Kapitola 2

Zobrazování stínů

2.1 Světelné zdroje

Světla jsou velmi důležitou součástí každé uměle vytvořené scény. Ačkoli pro syntézu reálného obrazu by byl potřeba fyzikálně velmi věrohodný model, v počítačové grafice se používají zjednodušené varianty kvůli výpočetní složitosti. Stefan Brabec [1] uvádí následující světelné zdroje (zde jsou vybrány jen ty nejdůležitější).

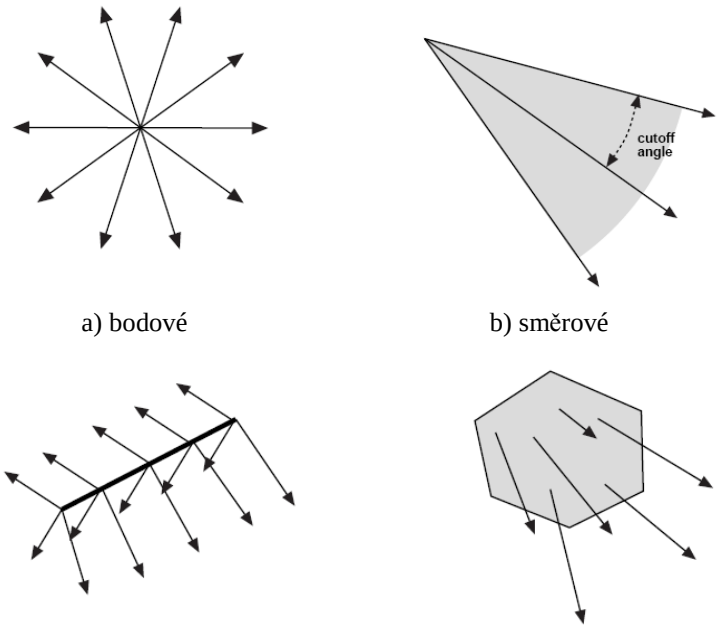
Základním světelným zdrojem je *bodové světlo* (angl. *point light*). Paprsky se šíří z jednoho bodu všemi směry. Je vhodné pro scény, kde se zdroj nachází v blízkosti objektů.

Jeho upravenou variantou je *směrové světlo* (angl. *spot light*), pod kterým si můžeme představit laser nebo reflektor, který vytváří světelné paprsky šířící se v kuželu definovaným *výřezovým úhlem* (angl. *cutoff angle*).

Další variantou jsou *lineární světla* (angl. *linear light*), pod kterými si za jistých okolností můžeme představit zářivky nebo neony.

Velmi často používaná *plošná světla* (angl. *area light*) jsou důležitá pro navození realističnosti při generování *měkkých stínů*. V reálném světě kromě plošné varianty jiné typy zdrojů světla prakticky neexistují, ale v závislosti na vzdálenosti od příjemce je můžeme zjednodušovat do předchozích variant[1] (obrázek 2.1). V počítačové grafice existují i komplexnější modely, ale nejčastěji se používají *bodová*, *plošná*, *směrová světla* a jejich kombinace.

[1]Například žárovka je považována za bodově světlo, ale ve vzdálenosti 10 cm od krabičky sirek tak rozhodně nepůsobí.



a) bodové b) směrové

c) lineární d) plošné

Obrázek 2.1: Světelné zdroje.

2.2 Stíny

Při vytváření realistických stínů je třeba kontrolovat dopad paprsků světla na všechny body zobrazené scény. Některá místa mohou být mimo kužel světla nebo mohou být zastíněna jinými tělesy. Cílem algoritmů pro generování realistických stínů je tato místa nalézt a správně zobrazit geometrii, která stín vytváří. Samozřejmě existují aproximativní metody navozující pouze dojem stínu, ale těmi se tato práce nezabývá.

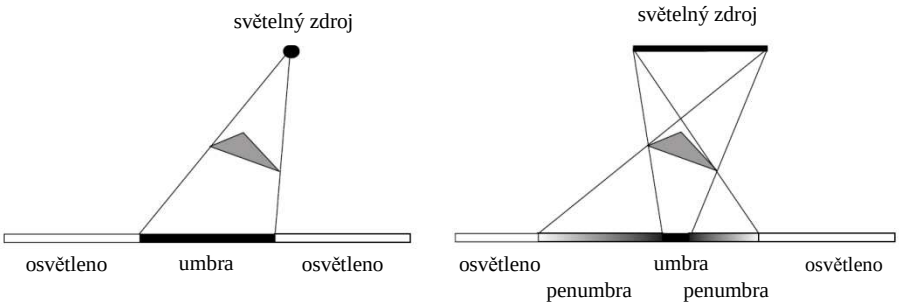
Dalším zajímavým aspektem jsou měkké vs. tvrdé stíny, které vznikají v závislosti na druhu zdroje světla. Tvrdé stíny jsou následkem bodových zdrojů světla. Jejich hrany jsou ostré po celém obvodu, tato část stínu se nazývá *umbra* (*Stín*). Naopak měkké stíny vznikají interakcí tělesa s plošným zdrojem světla. Tím dostáváme opravdu přirozené a věrohodné stíny (obrázek 2.2), kde je patrná absolutně zastíněná část – *umbra* a částečně zastíněná část – *penumbra* (*Polostín*). *Měkký stín* je ale v mnoha případech výpočetně náročnější, právě kvůli částečně zastíněné ploše. Podle [2] se vyzařování způsobené homogenním plošným světlem dá vyjádřit jako:

$$E = \int_A \frac{L \cos\theta_i \sin\theta_l}{\pi r^2} V dA \qquad (2.1)$$

kde θ_i je úhel, pod kterým dopadá paprsek světla na povrch tělesa, θ_l je úhel mezi paprskem světla a *normálou* světla, L je intenzita světla a V maska viditelnosti která je nastavena na 1 pro paprsky viditelné z daného bodu a 0 pro ostatní. Mnoho technik tento integrál zjednodušuje na:

$$ATT = \frac{1}{N}\sum_i^N V_i \qquad (2.2)$$

kde N je počet *vzorkování* pro jedno plošné světlo a *ATT činitel útlumu*.



a) tvrdý stín b) měkký stín

Obrázek 2.2

2.3 Přehled technik generování stínů

Existuje velké množství prací zaměřených na toto téma. Zde se zaměřím pouze na nejznámější a nejpoužívanější metody generování stínů pro *interaktivní* aplikace. Většina z nich využívá techniky pro odstraňování geometrií, které nejsou vidět z pohledu kamery, známé z OpenGL nebo DirectX. V tomto případě je pro část výpočtu považován zdroj světla za oko pozorovatele. Místa, která nejsou z jeho pozice vidět, se následně odstraní.

2.3.1 Paprsky stínů

Použitím metody *vrhání paprsků*, se zjišťuje *činitel útlumu* bodových nebo lineárních zdrojů světla. Pro tuto metodu se používá hrubá síla pro zjištění interakce všech paprsků s veškerými objekty ve scéně. Tento proces může být přesunut pomocí programovatelných *vertex* a *fragment shaderů* přímo na GPU[2], kde se scéna dá reprezentovat pomocí textur [3]. Následně se ve *fragment shaderu* spočítají interakce a promítnou stíny. Tato metoda je ale vhodná pouze pro středně komplexní scény. Na druhou stranu se s ní dají generovat *měkké stíny* reprezentací plošného zdroje světla například větším množstvím zdrojů bodových.

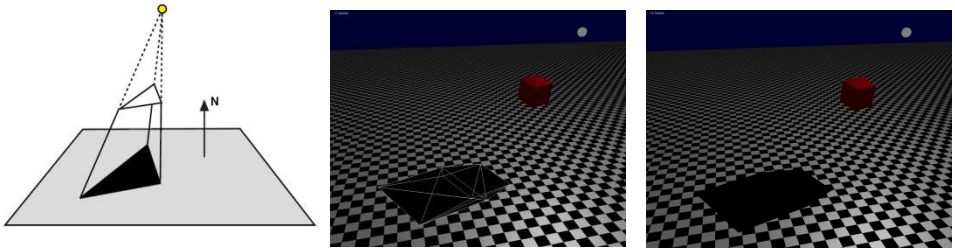
[2]Grafický procesor (z angl. **G**raphics **P**rocessing **U**nit)

2.3.2 Geometrická analýza

Zde jde o poměrně složitou metodu, kde se pomocí geometrické analýzy hledají místa ve scéně, ze kterých světlo, případně jen jeho část, není vidět. Vhodným postupem je projekce scény z pohledu světelného zdroje [4] a následné stínování každého bodu buďto analyticky nebo opětovným vzorkováním. Na první pohled je patrné, že tento postup není příliš vhodný pro *interaktivní* aplikace, ale existují implementace, které dokážou celý proces aproximovat [5].

2.3.3 Stíny na plošných příjemcích

Pro velmi specifické situace, kde se stín promítá na rovnou podložku, není třeba složitě generovat stín, ale stačí promítnout celé těleso. Tato metoda je vhodná z důvodu téměř nulové náročnosti – jen se každé těleso bez složitých úprav zobrazí podruhé (obrázek 2.3). Je třeba pouze provést projekci vyjádřitelnou pomocí matice o rozměrech 4x4 (podrobněji v kapitole 3).



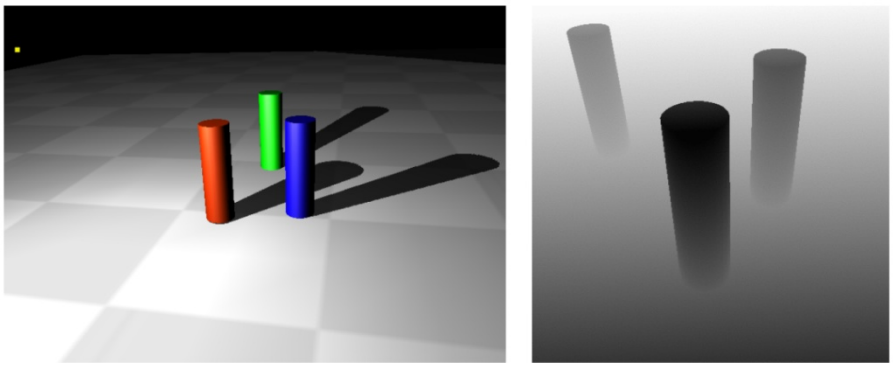
a) ilustrativní příklad b) jednotlivé stěny c) výsledný stín

Obrázek 2.3: Stín na plošném příjemci.

2.3.4 Stínové mapy

Stínové mapy (angl. *Shadow Maps*) patří v dnešní době mezi velmi oblíbené algoritmy. V roce 1978 předvedl ve svém algoritmu Lance Williams [6] základní myšlenku generování stínů pomocí zjednodušené reprezentace scény.

V prvním kroku je scéna zpracována z pohledu světelného zdroje. Do dvourozměrného pole (*mapy stínů*) se následně uloží hodnoty *depth bufferu* reprezentující nejbližší viditelná místa (obrázek 2.4). Ve druhém kroku je scéna opět zobrazena, tentokrát z pohledu kamery. Každý pixel je následně transformován do souřadnicového systému světelného zdroje pro porovnání jeho vzdálenosti s mapou stínů. Nevýhodou algoritmu jsou nekvalitní okraje stínů způsobené rozlišením stínové mapy. Většina úprav základního algoritmu se zabývá hlavně odstraněním této vady.



a) b)

Obrázek 2.4: Příklad scény (a) a náležící stínové mapy (b). Obrázky převzaty z [1].

2.3.5 Stínová tělesa

Stínová tělesa (angl. *Shadow Volumes*) se stala od svého uvedení Franklinem C. Crowem v roce 1977 [7] oblíbeným algoritmem obzvlášť pro svou vysokou kvalitu generovaných stínů.

Je rozdělen do 3. kroků. V prvním jsou všechna tělesa zpracována z pohledu světelného zdroje. Pro každý objekt a světlo je vytvořena *silueta* na hraně mezi jeho osvětlenou a neosvětlenou částí. Tato *silueta* je následně protažena do nekonečna. V místech, ze kterých není světelný zdroj vidět, vzniká *stínové těleso*. Druhý a třetí krok se mírně liší podle implementace, kde se s použitím *stencil bufferu* označí veškerá neosvětlená místa ve scéně – podrobněji v kapitole 3.

Kapitola 3

Stínová tělesa a plošný příjemce

V této kapitole bych chtěl podrobněji představit základní verze metod *plošného příjemce* a *stínových těles*. Zmínit jejich výhody, nevýhody a nedostatky. Hlavně u *stínových těles*, které jsem se ve své implementaci snažil vizuálně vylepšit. Obě metody jsou vizuálně, výkonově i obtížností jejich implementace zcela odlišné, proto byly vybrány a implementovány pro demonstraci těchto rozdílů.

3.1 Metoda plošného příjemce

Jak bylo zmíněno v předchozí kapitole, jedná se o velmi jednoduchý algoritmus vhodný pouze pro speciální případy, kde stín dopadá na rovnou podložku, a kde není třeba řešit jeho interakci s ostatními tělesy. V té chvíli je metoda *plošného příjemce* ideálním kandidátem pro požadovanou aplikaci. Není třeba složitě zjišťovat zakrytá místa, stačí pouze vytvořit *transformační matici*, která promítne osvětlené těleso na podložku.

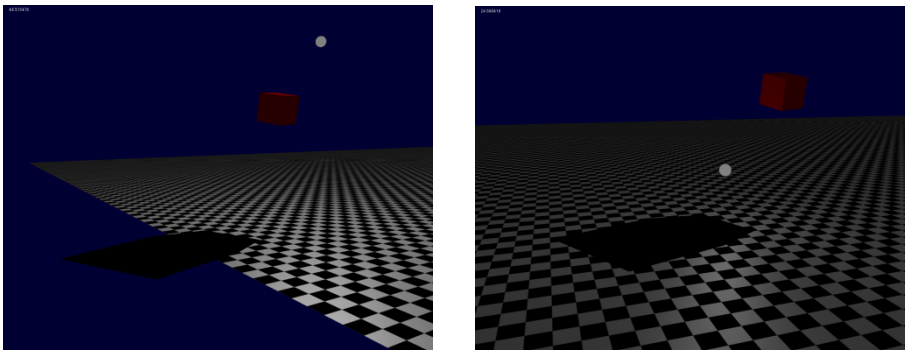
Pro jednoduchost předpokládáme polohu podložky ve výšce *y=0*. Projekci *p* bodu *v* na podložku podle světla *l* lze vyjádřit následovně:

$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y} \tag{3.1}$$

$$p_z = \frac{l_y v_z - l_z v_y}{l_y - v_y} \tag{3.2}$$

Algoritmus takto trpí dvěma nedostatky. V případě menšího plošného příjemce může stín přesahovat přes jeho hranu (obrázek 3.1 a). Tento

problém se ale dá vyřešit použitím *stencil bufferu*. Druhým nedostatkem je projekce stínu i v případě, že světelný zdroj je v poloze mezi objektem a příjemcem (obrázek 3.1 b). Řešení je opět jednoduché, stačí pouze otestovat polohu světla.



a) b)

Obrázek 3.1: Nedostatky plošného příjemce.

3.2 Metoda stínových těles

Myšlenka vrhání stínů pomocí *stínových těles* patří ke starším metodám, ale od prvního uvedení Franklinem C. Crowem [7] je stále hojně používaná. Dalo by se říci, že jde o základní a stěžejní metodu pro vrhání realistických stínů. Od uvedení původní verze se objevilo velké množství úprav a vylepšení algoritmu co se týče vlastností a místa zpracování (CPU, GPU a jejich kombinace).

3.2.1 Základní princip

V případě, že je algoritmus použit v dynamické scéně, kde se hýbou objekty nebo světla, je potřeba každý snímek vykonat následné kroky:

1. vytvořit aktuální *stínové těleso*
2. zobrazit přední stranu *stínového tělesa* a inkrementovat *stencil buffer* v místech viditelných z pohledu kamery (*depth test* projde). Vše s vypnutým zápisem do *frame* a *depth bufferu*.
3. zobrazit zadní stěnu *stínového tělesa* a dekrementovat *stencil buffer* v místech viditelných z pohledu kamery (*depth test* projde). Vše s vypnutým zápisem do *frame* a *depth bufferu*.

Tyto 3 kroky se týkají původní Crowovy *depth-pass* verze. Ostatní úpravy se více, či méně liší ve všech krocích. Největší rozdíl je ve tvaru *stínových těles* (první krok), kdy různé metody požadují uzavření horní

a spodní podstavy, případně protažení tělesa do nekonečna. Navíc pro statické scény stačí *stínové těleso* vytvořit pouze jednou při inicializaci. Druhý a třetí krok se liší hlavně v práci se *stencil bufferem*.

3.2.2 Vytváření stínového tělesa

Prvním krokem vytváření *stínových těles* je nalezení siluety objektu, tj. hranice (z pohledu světla) viditelné a zakryté části. Ta je tvořena hranami, buďto dvojice osvětlených a neosvětlených trojúhelníků, nebo hranami trojúhelníků bez sousedů. Zjistit, zda je trojúhelník osvětlen, se dá velmi snadno testem, jestli je světelný zdroj na přední straně jeho plochy. Tento krok musí být uskutečněn při každé změně polohy trojúhelníku nebo světla. V případě mé implementace se celé *stínové těleso* aktualizuje každý snímek.

Základní postup hledání siluety popisuje následující algoritmus v pseudokódu:

```
for všechny trojúhelníky
  if trojúhelník osvětlen
    zkontroluj jeho hrany jestli:  a) jsou sdíleny s neosvětleným
                                      trojúhelníkem
                                   b) jsou na okraji objektu
    if jedna z podmínek splněna, hrana je součástí siluety
  end
end
```

Je dobré zmínit, že tato operace se svou složitostí $O(n^2)$ patří ke dvěma výpočetně nejdražším částem celého algoritmu. Pokud ale trojúhelník z inicializace zná své sousedy, složitost se zmenší na $O(n)$, paměťové nároky však vzrostou.

Posledním úkolem je promítnout siluetu standardně do nekonečna. Dvojice bodů každé hrany siluety je promítnuta do předem definované vzdálenosti, případně do nekonečna použitím homogenního 4D souřadnicového systému[1]. Tím vznikne čtyřúhelník reprezentující jednu ze stěn *stínového tělesa*. Takto vznikne nekonečná geometrie, kterou je požadované *stínové těleso*. V *depth-fail* verzi je naopak třeba konečné *stínové těleso* s uzavřenou podstavou.

Pokročilejší algoritmy hledání siluety představili například Norman Chin a Steven Feiner [8], kteří upravili BSP[2] schéma. Pro každé světlo je

[1] (x, y, z, w), kde hodnota w je 1 pro body a 0 pro vektory

[2] BSP = **B**inary **S**pace **P**artitioning

vytvořen BSP strom, který reprezentuje celé *stínové těleso*. Poté následovalo ještě několik úprav pro zvýšení rychlosti zpracování, ale i tak BSP stromy nejsou pro dynamické scény ideální.

McCool [9] v roce 2000 představil algoritmus, který vytváří *stínové těleso* pomocí *hloubkové mapy*. V té se hledají nespojitosti, ze kterých se stanou ohraničující polygony *stínového tělesa*. Tato metoda ale trpí jejím nízkým rozlišením, vedoucím k nežádaným artefaktům na ploše příjemce.

Další možností je přesunout celý výpočet přímo na GPU a využít tak jeho masivního paralelismu, které přináší ještě další výhody v případě, že budeme objekty na grafické kartě deformovat. Tento postup s použitím *světového souřadného systému* a *4D textur* představil v roce 2003 Stefan Brabec [1]. Jako hlavní důvody pro přenos výpočtu uvádí nutnost synchronizace GPU a CPU v původní verzi. Aplikace navíc získá více procesorového času, který algoritmus prakticky úplně přestane využívat. Dalším důvodem jsou rozdíly v přesnosti výpočtů mezi CPU a GPU, kde například sqrt(x) může na obou jednotkách vést k významně rozdílným výsledkům vedoucím k chybám obrazu.

3.2.3 Depth-pass vs. depth-fail

Crowova *depth-pass* verze algoritmu má jedinou, ale zásadní nevýhodu. V případě, že se kamera dostane dovnitř *stínového tělesa*, algoritmus přestane fungovat – v druhém kroku se neinkrementuje *stencil buffer*, protože není vidět přední stěna *stínového tělesa*.

Řešení (ale zároveň i další problémy) přinesl John Carmack, Bill Bilodeau a Mike Songy v algoritmu známém jako *Carmack's reverse* [10]. Základní myšlenka se liší v 2. a 3. kroku:

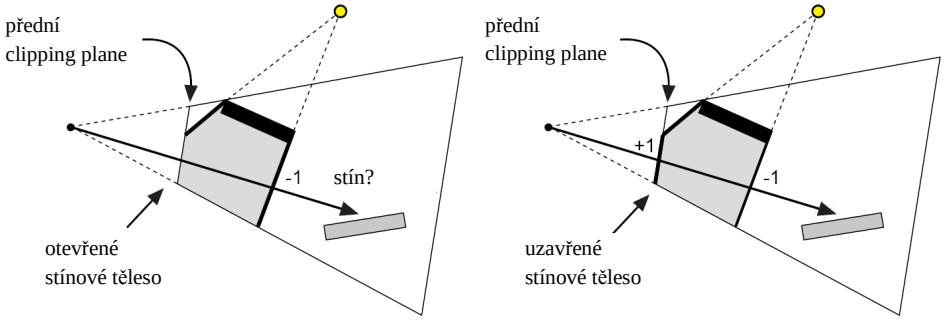
2. Zobraz zadní stěnu *stínového tělesa*. V případě, že selže *depth test* (stěna není vidět), inkrementuj *stencil buffer*. Opět s vypnutým zápisem do *frame* a *depth bufferu*.
3. Zobraz přední stěnu *stínového tělesa*. V případě, že selže *depth test*, dekrementuj *stencil buffer*. Vše s vypnutým zápisem do *frame* a *depth bufferu*.

Díky této úpravě algoritmus funguje i v případě, že kamera vstoupí do *stínového tělesa*. Problémy ale nastanou, jakmile *stínové těleso* narazí na zadní *clipping plane* – hranice, za kterou se již nic nezobrazuje. V tu chvíli se promítnou fragmenty inverzního stínu přímo na zastíněnou plochu. Tento problém se dá v některých případech snadno vyřešit uzavřením *stínového tělesa* ve vhodné vzdálenosti – vytvořením podstavy,

nebo ještě lépe jejím promítnutím přímo na *clipping plane*. Někdy je bohužel třeba pokročilejší metoda.

Další zajímavé a velmi elegantní řešení problému se vzdáleností zadní *clipping plane* je její posunutí do nekonečna. Tím *stínové těleso* prakticky nemá šanci *clipping plane* protnout – ale je stále třeba ho uzavřít, navíc se mírně zhorší přesnost *depth bufferu*. Úpravu projekční matice pro tuto situaci v OpenGL představil Eric Lengyel [11].

Depth-pass verze má ale podobný problém naopak s přední *clipping plane*. Je možné ho řešit projekcí stejně jako u *dept-fail* (Diefenbach [12], obrázek 3.2), ale v některých případech se objevují chyby v nepřesnosti výpočtů a vznikají tzv. *díry*, navíc jde o výpočetně složitou operaci. Triviální metoda řešící tento problém bohužel neexistuje, ale na druhou stranu se projevuje jen v malém počtu situací. Obrázek 3.1 ilustruje příklad tohoto problému.



a) b)

Obrázek 3.2: Problém s přední *clipping plane* u *depth-pass* verze algoritmu.

Ani jedna z těchto dvou metod není dokonalá (rychlost, robustnost), ale jejich kombinací, a vyhýbáním se velmi specifickým situacím, lze dosáhnout opravdu robustního generování kvalitních stínů.

Zde je jejich souborné srovnání:

	Depth-pass
výhody:	jednodušší na implementaci
	nevyžaduje uzavírání *stínových těles*
	je rychlejší
nevýhody:	nefunguje s kamerou uvnitř *stínového tělesa*
	problém s přední *clipping plane* nemá triviální řešení

	Depth-fail
výhody:	robustní metoda
nevýhody:	výpočetně i implementačně náročnější může vyžadovat *nekonečnou perspektivní projekci* vyžaduje uzavření obou podstav *stínového tělesa*

3.2.4 Měkké stíny

Stínová tělesa nejsou příliš vhodná pro generování měkkých stínů. Algoritmus naopak vytváří velmi přesně a ostře ohraničený okraj stínu, který se následně musí uměle změkčovat. První možností je několikanásobné vzorkování, což podstatně zvýší složitost algoritmu. Dále již delší dobu existuje algoritmus uvedený Tomoyukim Nishitou a kol. [13] v roce 1985. Ten konstruuje zvlášť *stínové těleso* pro měkkou i tvrdou část stínu.

Pro plošná světla byl v roce 2003 představen algoritmus nazvaný *penumbra wedges* [14], který generuje klíny místo jednoduchých stěn *stínového tělesa*. Výpočet těchto klínů je komplikovaný, ale přesto v jisté míře použitelný pro *interaktivní grafiku*.

Kapitola 4

Implementace

Cílem této části mé práce bylo vytvořit samostatnou aplikaci vhodnou pro testování a vyvíjení stínových algoritmů. Byl kladen důraz na snadné přidání nových algoritmů a jednoduchou konfiguraci testovací scény. V aplikaci lze použít libovolný objekt definovaný v standardně používaném OBJ formátu (*.obj*), použít libovolné textury v standardním grafickém formátu (*.bmp, .emf, .gif, .ico, .jpg, .wmf, .tga*) a vytvořit tak libovolnou scénu konfigurací vstupního souboru (*.xml*).

4.1 Prostředí a použité knihovny

Programovací jazyk zvolený pro implementaci prostředí i jednotlivých stínových algoritmů je C++. A to hlavně díky své rychlosti, rozšířenosti ve světě počítačové grafiky a počtu použitelných knihoven. Pro *shadery* implementovaných algoritmů byl použit jazyk CG. Aplikace byla vytvořena pro operační systém Windows XP nebo vyšší. Vývoj byl proveden na Windows XP SP3.

Při implementaci byly použity následující knihovny:

GLUT – Volně šiřitelná knihovna fungující jako nadstavba OpenGL, která obstarává vytvoření oken a menu, dále kontroluje reakci na myš a klávesnici.

expatpp – Načítání konfiguračních XML dokumentů.

TextureLoader – Plné zpracování textur různých vstupních formátů. Zde děkuji Chrisu Leathleymu za volnou distribuci jeho knihovny [15].

Licence jednotlivých knihoven jsou k dispozici v souboru **`licences.txt`** na přiloženém disku CD-ROM. Všechny jsou platformě nezávislé.

4.2 Preprocesor

Veškeré zpracování vstupních dat si aplikace provede sama při každém spuštění v závislosti na konfiguraci vstupního XML souboru. Toto předzpracování se týká hlavně spočtení chybějících normál objektů nebo zjištění sousednosti trojúhelníků, která je důležitá pro snížení výpočetní složitosti běhu algoritmu *stínových těles*.

4.3 Testovací prostředí

Testovací prostředí (nazvané **`Shadow Algorithms`**) je důležitou součástí této práce z důvodu jeho použitelnosti pro vývoj a testování algoritmů jiných autorů. Bylo navrženo a zdokumentováno tak, aby případný zájemce mohl velmi snadno přidat nové algoritmy a případně prostředí rozšířit nebo upravit. Je rozděleno do modulů obstarávající základní funkce vhodné pro stínové algoritmy jako například zobrazení objektů, transformace matic, vektorů, atd. Navíc, díky knihovně **expatpp**, je možné jednoduše přidávat další parametry konfiguračního XML souboru.

Program je sestaven tak, aby veškeré funkce byly intuitivně seskupeny podle svého účelu. Kromě stínových algoritmů, knihovny pro načítání XML a modulu pro zobrazování primitiv není objektově orientovaný z důvodu potřeby algoritmů rychle přistupovat ke globálním proměnným. Některé potřebují vidět prakticky na všechny proměnné, se kterými prostředí pracuje. V tu chvíli objektový návrh ztrácí smysl. Veškeré podrobné informace jsou v programátorské dokumentaci na přiloženém disku CD-ROM.

Z důvodu potřeby některých stínových algoritmů pracovat různě s testovací scénou, povinnou vlastností každého z nich je zobrazit všechny objekty buďto voláním funkcí, které nabízí prostředí, nebo jejich vlastní implementací, která se většinou liší pouze v detailech. Jediné, co prostředí ze scény zobrazí, jsou světla, čítač FPS a *clipping planes*.

Prostředí je velmi jednoduše ovladatelné pomocí klávesnice a myši. Za běhu programu je možné přepínat algoritmy, případně měnit jejich parametry, vypínat/zapínat *shadery* a světla. Kamera se může libovolně pohybovat a natáčet po celé scéně.

4.4 Plošný příjemce

Metoda plošného příjemce byla implementována ve 2 variantách. Nebyl na ni kladen takový důraz jako na stínová tělesa, proto obě verze kontrolují pouze polohu světla kvůli falešnému stínu, ale přesah přes hranu příjemce neřeší. První varianta promítá objekt v černé barvě 0.03 jednotky nad příjemce – z důvodu nepřesnosti ve výpočtech u čísel s plovoucí řádovou čárkou se při promítání přímo na podložku stín může s rostoucí vzdálenosti kamery ztrácet. Druhá varianta pracuje shodně, pouze oblast stínu místo začernění ztmaví.

4.5 Stínová tělesa

Stejně jako u plošného příjemce byl algoritmus implementován ve dvou variantách. První z nich je naprosto standardní verze popsaná v kapitole 3. Vzhledem k vizuálním nedostatkům ale vznikla ještě druhá varianta, která tyto neduhy originálního algoritmu zcela odstranila. Obě varianty jsou typu *depth-pass*.

Oba algoritmy jsou shodné svými požadavky. Při jejich inicializaci je potřeba najít, popřípadě načíst, sousedy každého trojúhelníku – testovací aplikace ve své inicializaci převádí veškeré konvexní mnohoúhelníky na trojúhelníky, takže jiné geometrie není třeba řešit. Hledání sousedů je operace s kvadratickou složitostí, proto je pro detailnější objekty možné zpracovanou sousednost uložit do souboru ve formátu *.txt* (podrobněji v uživatelské příručce). Z kapitoly 3 společně s potřebou inicializovat sousednost je patrné, že *stínové těleso* je vytvořeno na CPU. Vzhledem k velkému rozmachu vícejádrových procesorů jsem nepovažoval za nutné příliš šetřit procesorový čas. Paměťové nároky tak vzrostly o 3*32*n bitů, kde n je počet trojúhelníku, ale odměnou je zrychlené hledání siluety osvětleného tělesa se složitostí O(n).

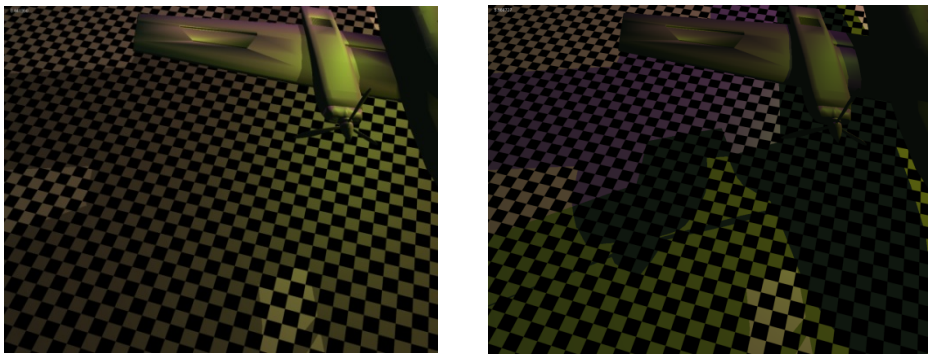
Za účelem co nejvíce oddělit stínové algoritmy od testovacího programu mají obě implementace *stínových těles* pro svou potřebu upravené kopie všech objektů ve scéně, ale zároveň využívají originální objekty a ostatní globální proměnné, ke kterým mají neomezený přístup.

Neuzavřené *stínové těleso* je složeno průměrně z méně jak polovičního počtu trojúhelníků původního tělesa. U velmi jednoduchých objektů jich může být více, ale i tak jsou jeho paměťové nároky zanedbatelné. Po vytvoření *stínového tělesa* se začínají verze lišit. První se drží popsaného

algoritmu a pro každé světlo provede všechny 3 kroky. Do *stencil bufferu* se tak uložilo každé místo scény, které není osvětleno alespoň jedním světlem. Tyto oblasti jsou následně ztmaveny a algoritmus končí. Na první pohled má tento postup několik nedostatků:

1. Všechny stíny od různě intenzivních světelných zdrojů jsou stejně tmavé, navíc jejich překřížením nevzniká tmavší zastíněná oblast.
2. Stíny nereagují na barevnost světelných zdrojů.
3. Zastíněná oblast reaguje na *ambientní* i *spekulární složku*.
4. Na zaoblených objektech vzniká nepřirozená hranice mezi osvětlenou a zastíněnou částí, což je vizuálně nejnepříjemnější vlastnost algoritmu, způsobená jednolitým ztmavením všech míst dopadu stínu včetně odvrácené strany objektu, bez ohledu na jakékoliv stínování.

Druhá verze přistupuje k problému odlišným způsobem. V první fázi se do *akumulačního bufferu* zobrazí celá scéna pouze s ambientním osvětlením. Dále se zvlášť pro každé světlo připíše osvětlená část scény, neosvětlená část se ztmaví – bylo to nutné, protože jinak vznikaly inverzní stíny. Tímto postupem scéna se shodnými parametry působí tmavším dojmem, ale odstraňuje všechny zmíněné nedostatky originálního algoritmu. Zobrazí se tak velmi pěkné a barevně propracované stíny. Obrázek 4.2 demonstruje vizuální rozdíly u 1. a 2. nedostatku původní verze (a) a jejich eliminace u druhé verze (b), při použití dvou různobarevných světelných zdrojů.

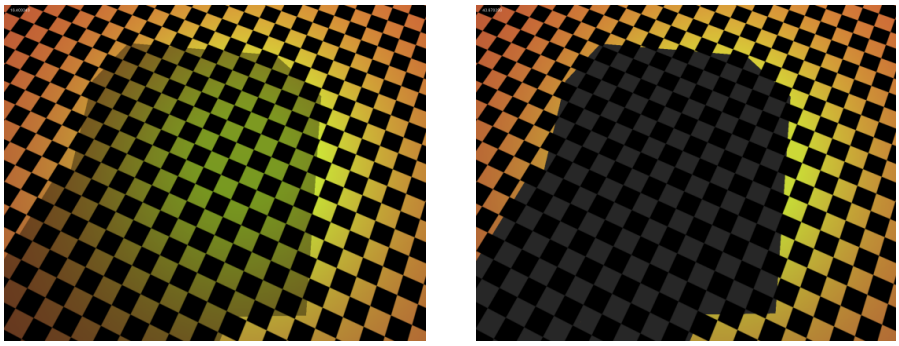


a) b)

Obrázek 4.2: Reakce na dva různobarevné světelné zdroje obou verzí algoritmu.

Díky opětovnému vykreslování nezastíněných částí scény pro každý světelný zdroj se v oblasti stínu neprojeví *difuzní* a *spekulární složka*. Obrázek 4.3 demonstruje scénu, s jedním světelným zdrojem s šedou *ambientní*, červenou *difuzní* a zelenou *spekulární složkou*, první (a) a druhé

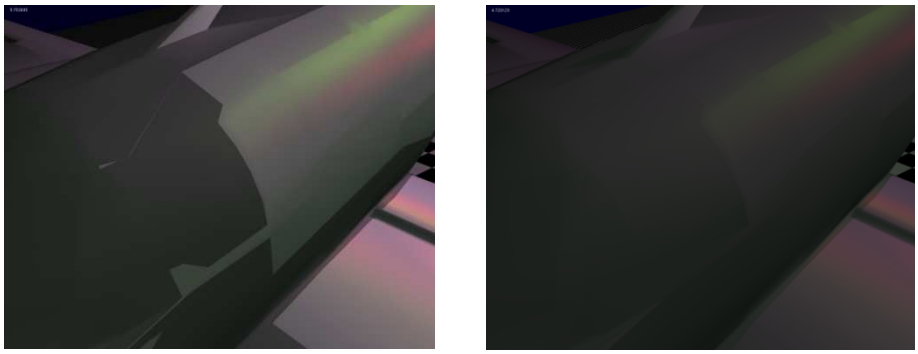
(b) verze algoritmu. Druhá verze tento problém eliminuje i při použití dvou a více světelných zdrojů.



a) b)

Obrázek 4.3: Problém reakce na ambientní i spekulární složku světla uvnitř stínu. Původní verze (a), opravená verze (b).

Posledním eliminovaným nedostatkem původního algoritmu jsou ostré hrany přechodu mezi zastíněnou a osvětlenou částí objektu. U obou algoritmů je použito *Phongovo stínování*, které uměle zaobluje hrany mezi trojúhelníky. Pozorovatel díky tomu vidí objekt jako jednolitou plochu, na které jsou zamaskovány plošky jednotlivých trojúhelníků. Oproti tomu pro stínová tělesa je bohužel relevantní pouze hranice mezi trojúhelníky viditelná na obrázku 4.4 (a) původního algoritmu, kde jsou použity 3 světelné zdroje. U více světelných zdrojů má použití *akumulačního bufferu* (u druhé verze algoritmu) za následek zjemnění rozdílů mezi těmito oblastmi, které demonstruje obrázek 4.4 (b).



a) b)

Obrázek 4.4: Přechody mezi osvětlenou a zastíněnou částí objektu.

4.6 Testy a výsledky

Nejprve, než se dostaneme k samotným testům, bych rád podrobněji popsal testovací aplikaci a samotné algoritmy. Výsledky poté budou srozumitelnější.

Poměrně často se setkávám se situací, kdy na internetu naleznu pěkný objekt ve formátu *.obj*, který bohužel ve své definici nemá přiloženy *normálové vektory*, které jsou kriticky nutné pro použité *Phongovo stínování*. V konfiguračním XML souboru je možnost *normály* objektu po zpracování uložit nebo načíst. V případě, že objekt je nemá definované a není zadán soubor pro jejich načtení, program je nejprve vytvoří kolmé na plochu svého trojúhelníku, a následně změní jejich směr podle sousedů. To má hned dva následky. Hranaté objekty bez normál se budou jevit zaobleně a pro složitější objekty bude výpočet trvat delší dobu. Totéž platí i pro hledání sousednosti pro *stínová tělesa*, kde je již pro 7000 trojúhelníků výpočet opravdu dlouhý. Tato příprava není stěžejní částí mé práce, proto nebyla optimalizována. Navíc, při dalším spuštění tento problém odpadá.

Všechny algoritmy je možné používat s vypnutými i zapnutými *shadery*, ale nastavení světelných zdrojů testovacích scén bylo optimalizováno pro jejich použití – s vypnutými *shadery* se scéna jeví „přepáleně“. Testovací program disponuje předpřipraveným *Phongovým stínováním*, které používají všechny algoritmy kromě rozšířené verze *stínových těles*. Ta má svou upravenou verzi pracující pro každé světlo zvlášť. Vzhledem k potřebě *Phongova stínování* normalizovat *normálové vektory* přímo ve fragment *shaderu* GPU, se jedná o jediný zásadně zpomalující prvek grafického jádra.

Plošný příjemce je jednoduchý algoritmus, který pro svou jedinou operaci (vytvoření *stínové matice*) využívá CPU. Obě varianty *stínových těles* vyžadují synchronizaci CPU a GPU, kdy optimalizovaná verze vytváření *stínového tělesa* zatěžuje jedno jádro CPU standardně na 100%. GPU obstarává stínování a *stencil buffer* spolu s *akumulačním bufferem* u druhé varianty. *Phongovo stínování* a vytváření *stínového tělesa* jsou dvě nejnáročnější operace, což se razantně projevuje na výkonu. Teď k samotnému testování.

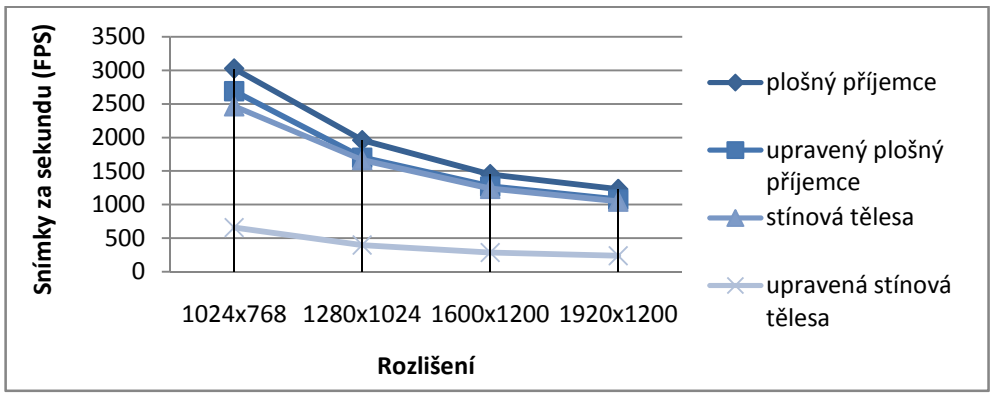
Testy probíhaly na pracovní stanici s procesorem Intel Core 2 Duo 2,8 GHz s operační pamětí 2 GB DDR2 800MHz a grafickou kartou Nvidia GeForce 9600 GT s 512MB GDDR3 VRAM v rozlišeních 1024x768, 1280x1024, 1600x1200 a 1920x1200.

Pro test byly použity 2 scény. První s 5 objekty o celkovém součtu 74 trojúhelníků (Obrázek 4.5 a), druhá s jediným objektem a 7446 trojúhelníky (Obrázek 4.5 b). Celkem jsem provedl 8 testů nazvaných T_1-T_8, každý ve všech uvedených rozlišeních:

Test	# Světelných zdrojů	Shadery	Scéna
T_1	1	vypnuté	č.1
T_2	1	zapnuté	č.1
T_3	3	vypnuté	č.1
T_4	3	zapnuté	č.1
T_5	1	vypnuté	č.2
T_6	1	zapnuté	č.2
T_7	3	vypnuté	č.2
T_8	3	zapnuté	č.2

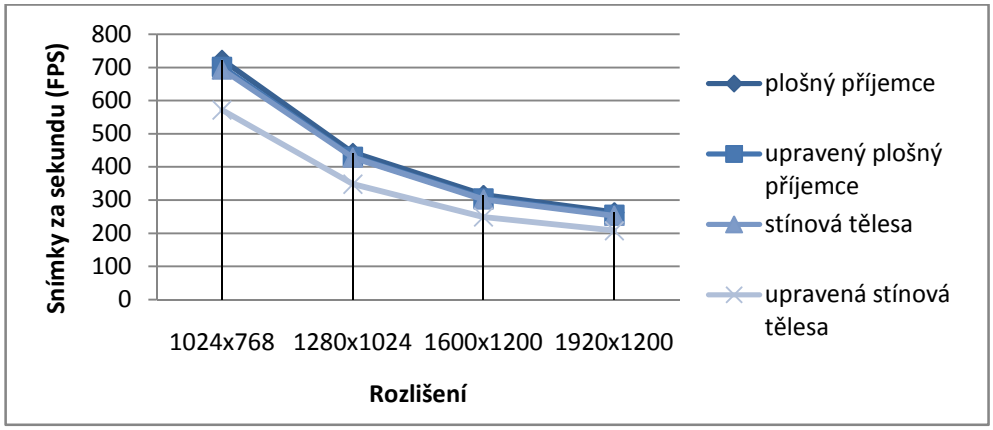
Rozdíl testů je v počtu světelných zdrojů, použití *shaderů* a použité scéně. Vzhledem ke koncepci programu, kdy se *stínová tělesa* a *stínové matice* počítají i v případě, kdy objekt není vidět, pohyb kamery neměl na měřené výsledky téměř žádný vliv. Proto všechny následující hodnoty jsou zprůměrovány a zaokrouhleny směrem dolů za 10s běhu se statickou kamerou. V žádné z testovacích scén není vidět zadní *clipping plane*, právě proto byl každý druhý test proveden s vypnutými *shadery* pro ilustraci složitosti *Phongova stínování*.

Obrázek 4.6 ilustruje výkon algoritmů u první scény s vypnutými *shadery* (test T_1). První verze *stínových těles* je hned v závěsu za oběma variantami *plošných příjemců*. U vylepšené verze je naopak patrný vliv *akumulačního bufferu*, který výkon omezuje.



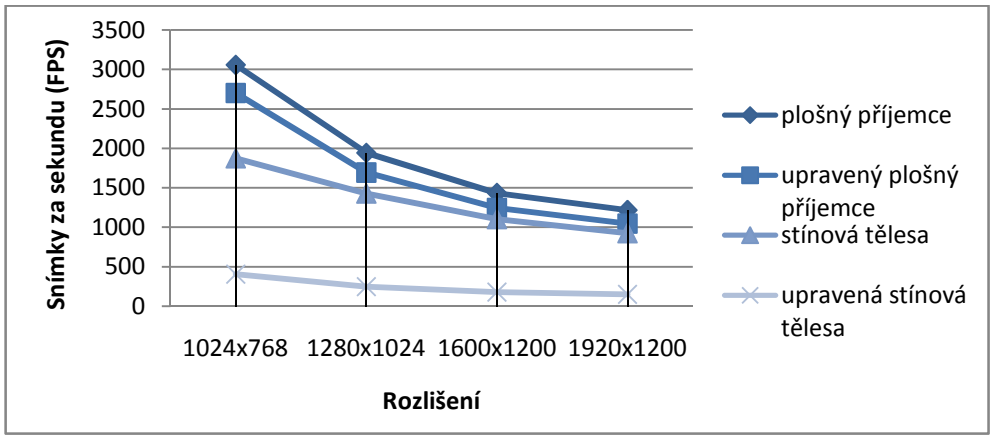
Obrázek 4.6: Test T_1

Obrázek 4.7 ilustruje shodnou scénu se zapnutými *shadery*. Zde se právě ony stávají omezujícím článkem, a všechny algoritmy mají velmi podobné výsledky.



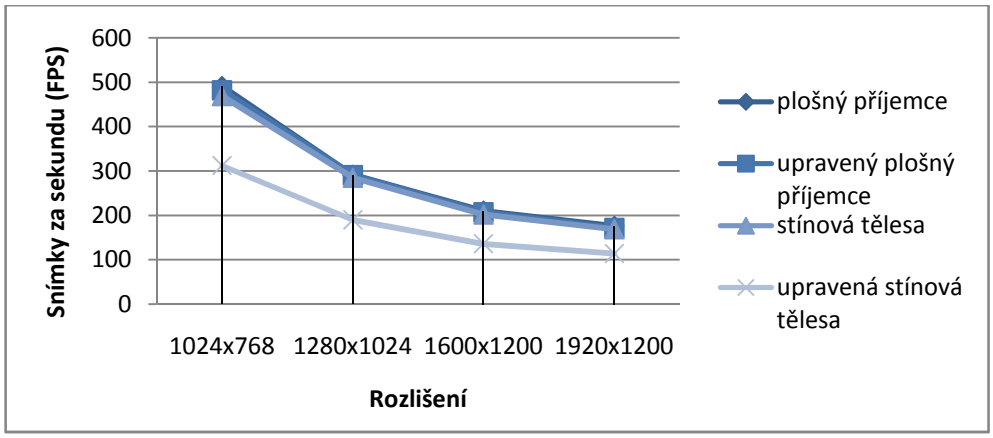
Obrázek 4.7: Test T_2

Přidání dvou světelných zdrojů s vypnutými *shadery* nemá na *plošného příjemce* prakticky žádný efekt. Zatímco u obou variant *stínových těles* je pozorovatelný propad výkonu až o 40% (viz obrázek 4.8):



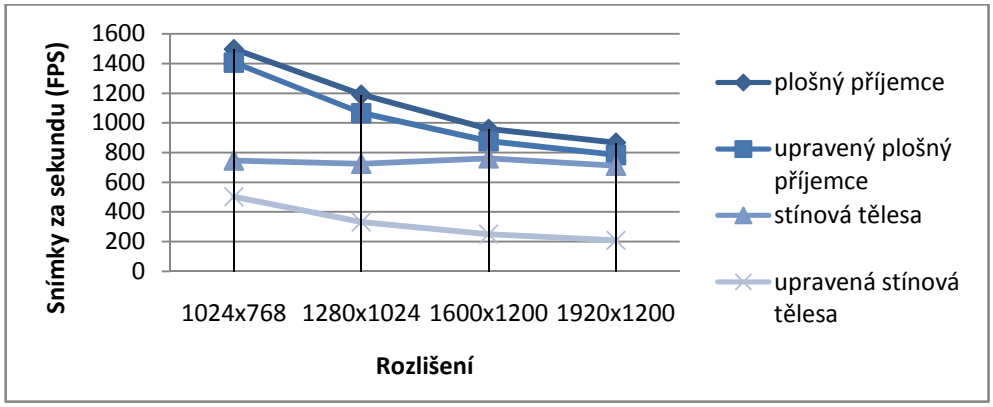
Obrázek 4.8: Test T_3

Opětovnou aktivací *shaderů* při použití 3 světelných zdrojů jsou všechny algoritmy výkonnostně podobné (obrázek 4.9). Pouze u upravených *stínových těles* se opět projevuje zpomalení zaviněné několikanásobným zobrazováním scény do *akumulačního bufferu*.



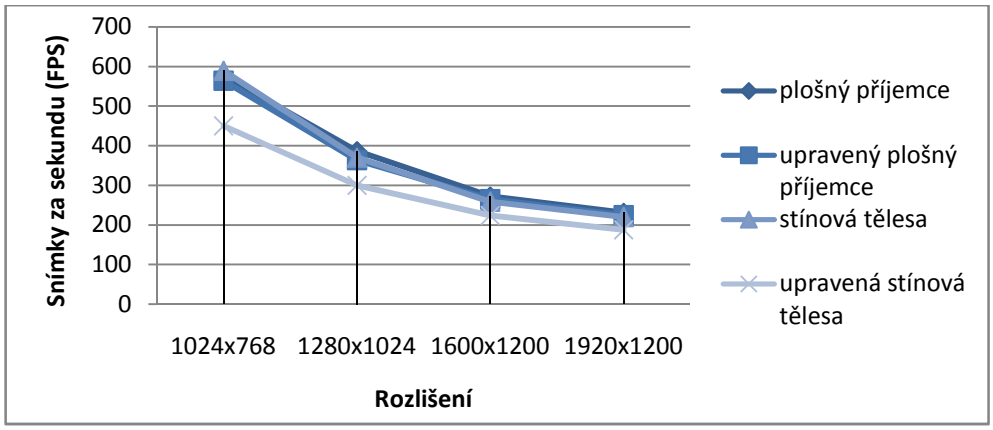
Obrázek 4.9: Test T_4

U druhé scény je situace zcela rozdílná. Při použití jediného světelného zdroje s vypnutými *shadery* je výkon prvního algoritmu *stínových těles* zcela zřetelně omezen CPU (obrázek 4.10). Tento fakt je prokazatelný ze stabilní hodnoty FPS ve všech rozlišeních. Graf také ukazuje, že optimální rozlišení této scény pro vyváženou spolupráci CPU a GPU u *stínových těles* je 1920x1200.



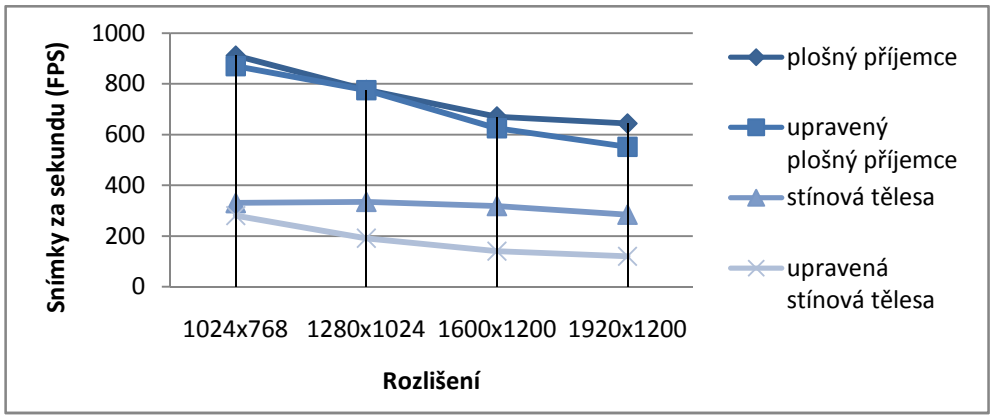
Obrázek 4.10: Test T_5

Po zapnutí *shaderů* se *upravená stínová tělesa* drží na téměř totožných hodnotách, tentokrát je rovnoměrné rozložení sil CPU a GPU na jejich straně. Ostatní algoritmy ztrácí vinou *shaderů* (viz obrázek 4.11).



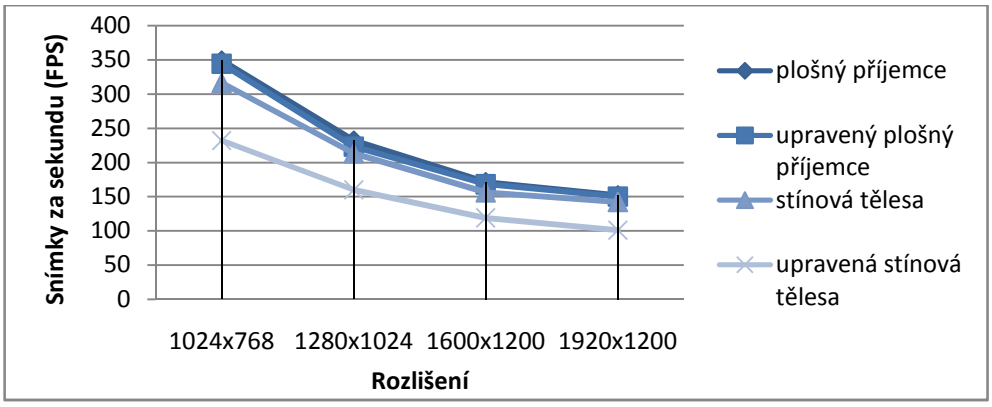
Obrázek 4.11: Test T_6

Přidáním dalších dvou světelných zdrojů je na výkonu *stínových těles* stále patrnější vliv CPU a již obě varianty začínají na *plošného příjemce* ztrácet (viz obrázek 4.12).



Obrázek 4.12: Test T_7

Aktivace *shaderů* opět sráží výkon obou *plošných příjemců* na úroveň *stínových těles*. *Upravená stínová tělesa* si ale drží stabilní výkon (obrázek 4.13).



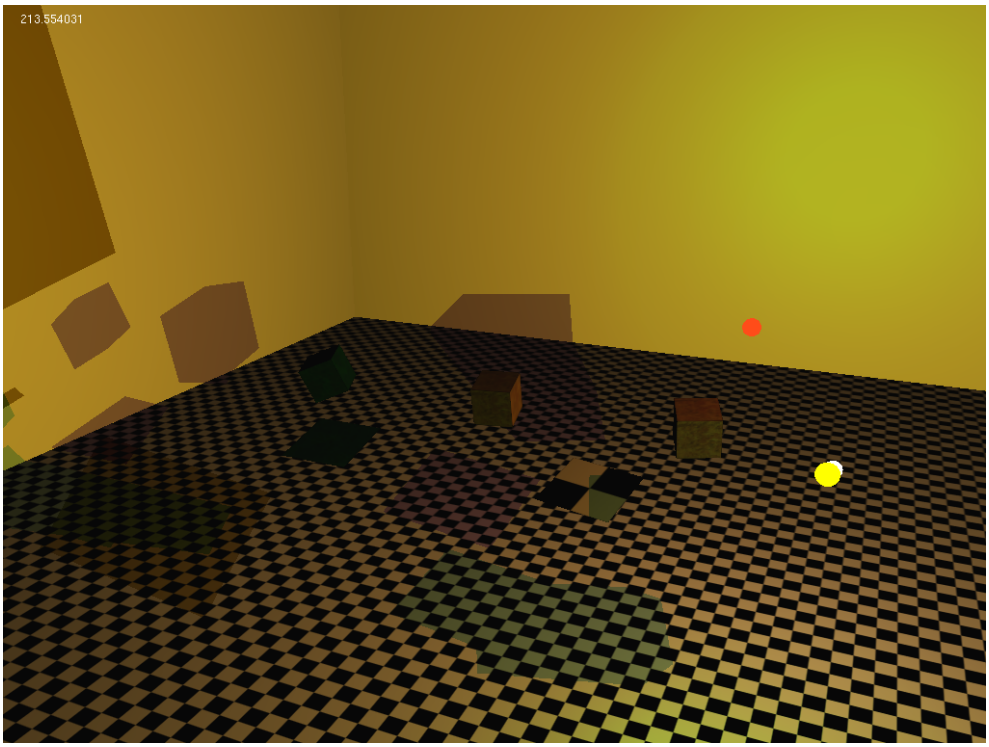
Obrázek 4.13: Test T_8

Jen pro shrnutí výsledků ještě přikládám tabulku demonstrující výkonnostní rozdíly implementovaných algoritmů v rozlišení 1280x1024 ve všech prezentovaných testech:

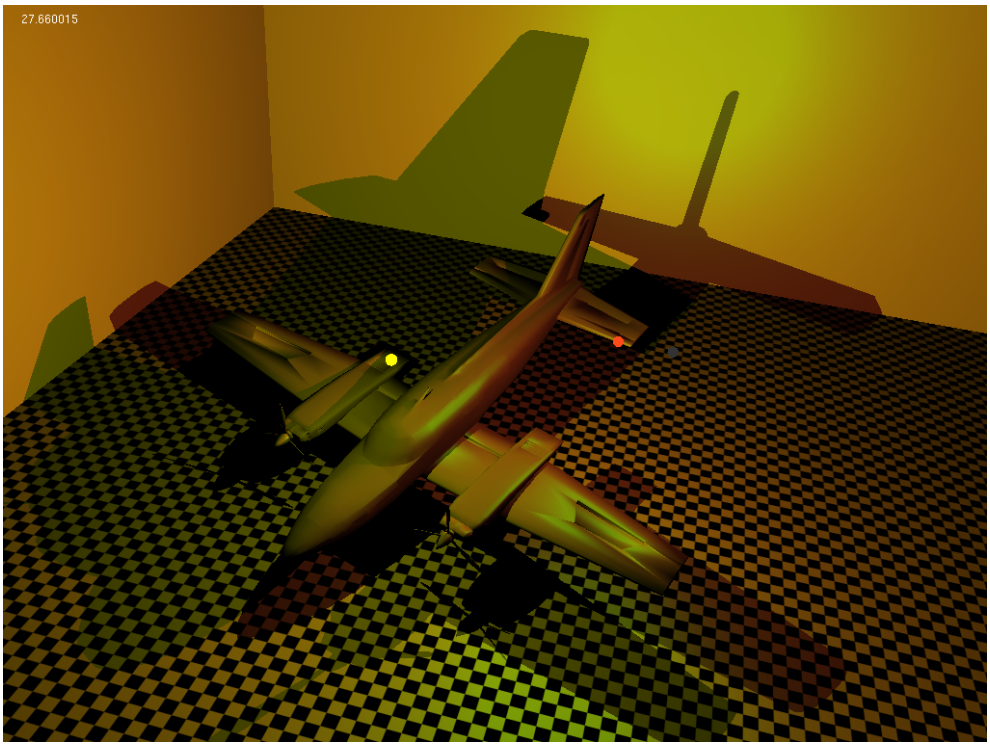
1280x1024	plošný příjemce	upravený plošný příjemce	stínová tělesa	upravená stínová tělesa
T_1	1961	1701	1665	399
T_2	442	430	428	348
T_3	1943	1695	1427	245
T_4	291	291	285	190
T_5	1189	1065	724	331
T_6	386	364	370	300
T_7	777	775	334	191
T_8	232	223	213	160

Z provedených testů je patrné, že implementace výpočtu *stínových těles* na CPU nemá žádné větší negativní dopady na výkon. Naopak rovnoměrně využívá sil všech prostředků. U ještě složitějších scén by bylo vhodné celý algoritmus implementovat přímo na GPU a otestovat výkonnostní rozdíly, případně použít například *stínové mapy*. Ale v použitých scénách byl vliv CPU minimální.

Všechny naměřené výsledky jsou k dispozici ve formátu *.xls* na přiloženém CD.



a)



b)

Obrázek 4.5: Screenshoty z testů.

Kapitola 5

Závěr

5.1 Shrnutí

V první kapitole jsme otevřeli téma vrhání stínů a popsali proč je důležité pro počítačovou grafiku. Dále jsme si objasnili rozdíl mezi *interaktivním* a *neinteraktivním zobrazováním* a uvedli, kterým směrem se práce vydává, včetně stanovení jejích cílů.

V druhé kapitole jsme se seznámili s druhy světelných zdrojů a definovali, co je to vlastně stín, jaké jsou jeho varianty a za jakých podmínek vznikají. Poté jsme krátce představili základní metody generování stínů pro *interaktivní grafiku* a zmínili jejich výhody a nedostatky.

Celá třetí kapitola nám podrobněji objasnila metodu *plošného příjemce* a *stínových těles*. Důraz byl kladen hlavně na *stínová tělesa*, která jsou mnohem pokročilejší, a existuje větší počet různých přístupů jejich implementace. Objasnili jsme rozdíly mezi metodami *depth-pass* a *depth-fail*, a následně určili, kdy je která metoda vhodnější. Nakonec jsme rozebrali možnosti využití CPU a GPU.

Čtvrtá kapitola popisuje program určený pro testování rychlosti a vizuální kvality libovolných metod pro generování stínů. Dále byla popsána implementace algoritmů, představených ve 3. kapitole. Zbytek kapitoly se věnoval jejich testům.

5.2 Splnění cílů

Ve své práci se mi podařilo splnit prakticky všechny vytyčené cíle. Bylo vytvořeno prostředí, vhodné pro implementaci a testování algoritmů, použitelné pro jiné programátory a zároveň poskytující základní funkce

ulehčující práci při tvorbě nových a renovaci starších metod generování stínů. Po prostudování různých technik jsem si vybral hlavně metodu *stínových těles* a pro porovnání výkonu jednoduchou metodu *plošného příjemce*. Metodu *stínových těles* jsem v první fázi implementoval v základní verzi podle [11] a následně odstranil její hlavní vizuální nedostatky. I pro složitější objekty bylo dosaženo relativně dobrých výsledků v řádu stovek FPS. Jedinou nevýhodou původní i vylepšené verze je veliké zatížení jednoho jádra CPU. Na druhou stranu díky rozmachu multiprocesorových počítačů to není tak zásadní problém. Implementace na GPU by jistě přinesla mnoho výhod, ale na úkor výkonu *shaderů*.

5.3 Diskuze a směr další práce

Když jsem začal pracovat na tomto projektu, mé znalosti z počítačové grafiky byly téměř nulové. Během mnoha hodin strávených v knihách a laboratoři jsem se podrobněji seznámil s programovatelnými *shadery*, které mi otevřely svět mnohých možností. Test implementace algoritmu *stínových těles* na komplexnější scéně prokázal, že pokud existuje možnost paralelizovat libovolný grafický výpočet a provést ho přímo na GPU, měla by se vyzkoušet. Algoritmus *plošného příjemce* naopak zatížil *shadery* grafické karty na maximum a otestoval jejich limity.

Algoritmus *stínových těles* je bez diskuze v mnoha ohledech komplexnější než *plošný příjemce*, což se promítá do realističnosti složitějších scén, výpočetní složitosti a náročnosti jeho implementace. Bohužel není univerzální hlavně díky vysoké ceně svého výpočtu, ale počítačová grafika vždy byla a bude o kompromisech.

Do budoucna bych v první řadě chtěl přesunout veškeré výpočty týkající se *stínových těles* přímo do *shaderů* a porovnat tak výkonnostní rozdíly obou řešení. Dalším krokem by mohla být implementace *stínových map* a prozkoumání použitelnosti těchto technik v interakci s kapalinami a plyny, což by vedlo k rozšíření programu o schopnost jejich věrohodné simulace.

Literatura

[1] BRABEC S. 2003. *Shadow Techniques for Interactive and Real-Time Applications.* Max-Planck-Institut fur Informatik. 13-15, 99-108.

[2] AGRAWALA, M., RAMAMOORTHI, R., HEIRICH, A., MOLL, L. 2000. *Efficient Image-Based Methods for Rendering Soft Shadows.* ACM SIGGRAPH 2000, 375-384.

[3] PURCEL, J. T., BUCK, I., MARK W. R., AND HANRAHAN, P. *Ray Tracing on Programmable Graphics Hardware.* ACM Transactions on Graphics 2002, 703-712.

[4] DRETTAKIS G., AND FIUME, E., 1994. *A Fast Shadow Algorithm for Area Light Sources Using Back Projection*, SIGGRAPH 94, 223-230.

[5] PARKER, S., SHIRLEY, P. AND SMITH, B. *Single Sample Soft Shadows.* Computer Science Department, University of Utah 1998.

[6] WILLIAMS, L. 1978. *Casting Curved Shadows on Curved Surfaces.* SIGGRAPH 78, 270-274.

[7] CROW, F. C. 1997. *Shadow algorithms for computer graphics.* SIGGRAPH 1997, 242-248.

[8] CHIN, N. AND FEINER, S. 1989 *Near Real-Time Shadow Generation Using BSP Trees.* SIGGRAPH 89, 99-106.

[9] MCCOOL, M. D. *Shadow volume reconstruction from depth maps.* ACM Transaction on Graphics 2000, 1-26.

[10] CARMACK, J. 2000. *John Carmack on shadow volumes.* http://www.nvidia.com

[11] LENGYEL, E. 2002. *The Mechanics of Robust Stencil Shadows.* http://www.gamasutra.com

[12] DIEFENBACH, P. J., AND BADLER, N. 1994. Pipeline Rendering: Interactive Refractions, Reflections and Shadows.

[13] NISHITA, T., OKAMURA, I. AND NAKAMAE, H. *Shading Models forPoint and Linear Sources.* ACM Transactions on Graphiics 1985, 124-126.

[14] AKENINE-MOLER, T. AND ASSARSON, U., 2002. *Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges.* Eurographics Workshop Proceedings 2002, 297-305.

[15] LEATHLEY, C. *Texture Loader.* http://members.iinet.net.au/~cleathley/openGL/TextureLoader.html

Příloha A

Obsah CD

Součástí práce je disk CD-ROM obsahující výslednou implementaci projektu, příslušné knihovny a další. Strom adresářů je organizován takto:

- **`/docs`**
  - **`/api`** – uživatelská a programátorská dokumentace
  - **`/thesis`** – text této práce ve formátu *.pdf*
  - **`/tests`** – výsledky testů ve formátu *.xls*
  - **`/other`** – další neroztříděné dokumenty

- **`/install`** – instalátor aplikace Shadow Algorithms včetně testovacích scén a všech knihoven

- **`/objects`** – sada použitelných objektů do programu Shadow Algorithms

- **`/scrennshots`** – screenshoty z Shadow Algorithms použité v této práci

- **`/src`** – všechny zdrojové soubory programu i jednotlivých algoritmů včetně kompletní solution psané v MS Visual Studio 8

Příloha B

Shadow Algorithms 1.1 – uživatelská příručka

B.1 Systémové požadavky

Minimální konfigurace pro běh testovací aplikace (Shadow algorithms) je:

- CPU 1,5 GHz single-core
- Operační paměť 512MB
- Grafická karta podporující OpenGL 1.5, shader model 3.0 a 64MB VRAM
- 40 MB volného místa na pevném disku
- Operační systém MS Windows XP nebo vyšší

Teoreticky je možné program spustit i na méně výkonné sestavě v závislosti na složitosti testované scény.

B.2 Instalace

Instalace je opravdu jednoduchá. Stačí spustit soubor *ShadowAlgorithms_1_1.exe* ze složky **`/install`** na přiloženém CD. Průvodce Vás provede celým instalačním procesem. Obsahem instalace je i sada předpřipravených testovacích scén. Popřípadě další objekty je možné nalézt ve složce **`/objects`.**

B.3 Spuštění aplikace

Aplikaci lze spustit různým způsobem v závislosti na průběhu instalace buďto z nabídky **`start / všechny programy / Shadow Algorithms / SA.exe`** nebo přímo z instalačního adresáře spuštěním souboru **`SA.exe`**. Popřípadě při spuštění programu z příkazové řádky je možné zadat jako parametr vstupní XML soubor s testovací scénou, např.: **`C:\Program Files\Shadow Algorithms>SA.exe test.xml`**. V ostatních případech program vyžaduje zvolení XML souboru po svém spuštění. Dále podle konfigurace v XML může proběhnout předzpracování scény nebo přímo spuštění grafické části programu.

B.4 Ovládání

Pro posun kamery po scéně je definována sada kláves:

Klávesa	**Akce**
W, ↑	posun vpřed
S, ↓	posun vzad
A	posun vlevo
D	posun vpravo
Z	posun dolů
X	posun nahoru
Q	rotace vlevo dolů
E	rotace vpravo dolů
R	rotace dolů
F	rotace nahoru
←	rotace vlevo
→	rotace vpravo

Po stisknutí pravého tlačítka myši se zobrazí menu umožňující nastavování parametrů scény (změna stínového algoritmu, vypnutí/zapnutí *shaderů*, zobrazení stínových těles nebo ovládání světel). Světla je možné vypínat/zapínat i kliknutím levého tlačítka na jejich pozici.

B.5 Konfigurace pomocí vstupního XML souboru

Pro lepší představu o nastavování programu pomocí XML doporučuji prohlédnout soubor **`test.xml`**. Do scény programu je možné přidávat objekty, světla a nastavovat jejich parametry.

Obsah celého XML souboru musí být obalen tagem `<scene></scene>`. Uvnitř je možné definovat objekty pomocí tagu `<object>`. Zde je příklad jeho kompletní definice:

```
<object name="cube.obj" castShadow="yes">
    <position x ="-10" z="0"/>
    <transform x="0.2" y="0.3" scale="1.5"/>
    <adjacency mode="l" name = "cubeAdj.txt"/>
    <normals mode ="l" name ="cubeN.txt"/>
    <ambient r="1" g="1" b="1"/>
    <diffuse r="1" g="1" b="1"/>
    <specular r="1" g="1" b="1"/>
    <emissive r="0" g="0" b="0"/>
    <texture name="Stone.tga"/>
    <movement x="0.0" y="0.005" z="0.0" rot="yes"/>
    <color r="1" g="1" b="1" l="0"/>
</object>
```

Zde zadáváme objekt definovaný v souboru `cube.obj`, po kterém požadujeme, aby vrhal stíny. Střed jeho souřadnicového systému posuneme o `-10` jednotek po ose *x* pomocí tagu

```
<position x ="-10" z="0"/>.
```

Tag

```
<transform x="0.2" y="0.3" scale="1.5"/>
```

zajišťuje počáteční natočení objektu po osách *x* a *y*. Vynecháním parametru `z` se natočení po této ose neprovádí. Dále je objekt zvětšen 1,5x.

Tagy

```
<adjacency mode="l" name = "cubeAdj.txt"/>
<normals mode ="l" name ="cubeN.txt"/>
```

v tomto případě načítají sousednost trojúhelmíků a normálové vektory ze souborů `cubeAdj.txt` a `cubeN.txt`. Změnou parametru `l` na `s` se vytvoří nové normály a sousednost, výsledek se do souborů naopak uloží.

Tagy

```
<ambient r="1" g="1" b="1"/>
<diffuse r="1" g="1" b="1"/>
<specular r="1" g="1" b="1"/>
<emissive r="0" g="0" b="0"/>
```

nastavují reakci objektu na jednotlivé barevné složky při použití shaderů. V opačném případě je brána v úvahu hodnota zadaná tagem

```
<color r="1" g="1" b="1" l="0"/>
```

nebo tagem

```
<texture name="Stone.tga"/>,
```

který se samozřejmě uplatňuje i v případě, že jsou shadery použity.

Posledním tagem je

```
<movement x="0.0" y="0.005" z="0.0" rot="yes"/>,
```

který v tomto případě rotuje s objektem o 0,005 jednotky kolem jeho osy *y* za 1s. V případě změny posledního parametru na `rot="no"`, se objekt bude posouvat.

Dalšími objekty scény jsou světelné zdroje zadané např.:

```
<light>
    <ambient r ="0.0" g ="0.0" b="0.0" l="1"/>
    <specular r ="0" g ="1" b="0.0" l="1"/>
    <diffuse r ="1" g ="0.3" b="0.1" l="1"/>
    <position x="2.0" z="0.0" y="5.0"/>
    <centre x="0" y="0" z="0"/>
    <movement x="0.0" y="0.05" z="0.04"/>
    <spotdir x="1" y="1" z="1" w="30"/>
    <size r="0.4"/>
</light>
```

Tagy

```
<ambient r ="0.0" g ="0.0" b="0.0" l="1"/>
<specular r ="0" g ="1" b="0.0" l="1"/>
<diffuse r ="1" g ="0.3" b="0.1" l="1"/>
```

jsou podobné jako u objektů, zde se ale týkají vyzařování.

Tagy

```
<position x="2.0" z="0.0" y="5.0"/>
<centre x="0" y="0" z="0"/>
```

jsou v prvním případě pozice světla, v druhém střed pro jeho případné otáčení definované tagem

```
<movement x="0.0" y="0.05" z="0.04"/>
```

Pro směrová světla existuje tag

```
<spotdir x="1" y="1" z="1" w="30"/>
```

Směrová světla ale musí být implementována v shaderech stínových algoritmů, které by měly dodržovat konvenci směru zadaným vektorem (*x*,*y*,*z*) a výřezu *w* zadaného v stupních.

Posledním tagem pro světla je

```
<size r="0.4"/>
```

definující velikost sféry zobrazující se v programu.

Kamera se definuje následovně:

```
<camera>
    <position x="-5" y="0" z="-35"/>
    <perspective lens="45" near="0.1" far="425"/>
</camera>
```

kde se v druhém tagu nastavuje šířka záběru a vzdálenosti přední a zadní clipping plane.

Poslední dva tagy se týkají zobrazení podlahy nutné pro funkci algoritmu plošného příjemce, a vybrání startovního stínového algoritmu číslovaného od 0.

```
<ground extent="100" step ="2" y="-12.0"/>
<shadow alg="0"/>
```

U podlahy se dá nastavovat její velikost, umístění na ose *y* a velikost jednoho políčka pro kopírování textury.

Příloha C

Shadow Algorithms – přidání stínového algoritmu

C.1 Základní vlastnosti

V první řadě bych doporučil prostudovat programátorskou příručku přiloženou na disku CD-ROM pro pochopení struktury programu. Na druhou stranu je přidávání stínových algoritmů opravdu jednoduché a nevyžaduje jeho podrobnou znalost.

Všechny stínové algoritmy mají společného předka `ShadowAlgorithm`, který vypadá následovně:

```
class ShadowAlgorithm{
public:
      ShadowAlgorithm(){name = "Unnamed Algorithm";};
      virtual ~ShadowAlgorithm(){};
      virtual void Run(){};
      virtual void Init(){};
      virtual void SetupShaders(){};
      string GetName(){return name;};
protected:
      string name;
      Draw DRW;
};
```

Každý stínový algoritmus by tedy měl v konstruktoru uložit své jméno do proměnné `string name`. V případě, že potřebuje inicializaci, která je závislá na globálních proměnných prostředí, je potřeba ji celou provést ve funkci `void Init()`, která se volá automaticky až po jejich plné inicializaci.

`Draw DRW` je třída obsahující všechny funkce potřebné pro zobrazení těles a podlahy.

V dalším případě, pokud algoritmus používá vlastní *shadery*, je zde předpřipravena metoda `void SetupShaders()`, pro jejich nezávislou inicializaci.

Jako poslední a nejdůležitější je funkce `void Run()`, kterou se algoritmus spouští přímo v `void RenderScene()`.

Pro úspěšné spuštění algoritmu je třeba ho ještě zaregistrovat v `AlgorithmReferences.h` a v `AlgorithmReferences.cpp` podle předlohy použitých algoritmů.

C.2 Funkce Run()

Jak je patrné z funkce `void RenderScene()`, matice v OpenGL je nastavena na `GL_MODELVIEW`. Algoritmus je při spuštění funkce `Run()` v souřadnicovém systému přenásobeném pouze maticí kamery.

Vzhledem k faktu, že většina algoritmů má na zobrazení scény různé nároky, je funkce `Run()` zodpovědná za zobrazení objektů a podlahy (ground). Ve funkci `RenderScene()` se tedy zobrazují pouze světla a FPS. Funkce `Run()` musí v případě, že algoritmus nepoužívá žádné vlastní *shadery* obsahovat:

```
if (shaders)
{
	DRW.DrawShaderedGround();
	DRW.DrawShaderedObjects();
}
else
{
	DRW.DrawFullGround();
	DRW.DrawObjects();
}
```

Shadery se dají ovládat globálně, proto je třeba mít obě verze (i když by ta bez *shaderů* nic nedělala). V případě, že si tyto metody algoritmus implementuje sám, nahradí příslušné řádky. Třída `Draw` umí zobrazovat i jednotlivé objekty (transformaci souřadnic ale musí obstarat volající funkce), ale jen v případě vypnutých *shaderů*.
Vlastnosti prostředí se nastavují ve funkci `void SetupRC()`. V případě, že algoritmus něco změní, měl by po svém skončení vše uvést do původního stavu.
U všech globálních proměnných (kamera, objekty, světla) se každý snímek pouze aktualizují jejich matice a další příslušné parametry. Algoritmy se mohou spolehnout na to, že se jejich počet za chodu nezmění.

C.3 Funkce Init()

V době spuštění inicializace má algoritmus připraveny všechny proměnné pro svou potřebu. Jednotlivé trojúhelníky objektů jsou přístupné z `vector<PREOBJECT> preobjects`, odkud je v případě potřeby nutné vše důležité zkopírovat do vlastních struktur, protože po inicializaci algoritmů vektor `preobjects` zaniká (jako jediný). Směr vykreslování trojúhelníků je CCW.

C.4 Funkce SetupShaders()

Funkce `void SetupShaders()` je naprosto nezávislá na zbytku programu (a není povinná). Je možné se inspirovat hotovými algoritmy a použít CG, jehož kompilátor je v solution nastaven.