STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor 01 - Matematika

Gymnázium Dr. Karla Polesného, Znojmo

Složené objekty, řezy těles tělesy, a jejich zobrazování

Autor: Vilém Otte
Škola: Gymnázium Dr. Karla Polesného
Komenského nám. 4
669 75
Znojmo
Studijní obor: všeobecný, 3. ročník

Znojmo, 2010

Prohlášení:

Tímto prohlašuji že jsem práci vykonal samostatně a použil pouze podklady uvedené v seznamu referencí. Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, a právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

Ve Znojmě dne 24.3.2010 Podpis:

Poděkování:

Chtěl bych poděkovat Mgr. Petru Novotnému za trpělivé vysvětlování té nejkrásnější ze všech přírodních věd, dále také RNDr. Haně Krčálové za její pomoc při formálních opravách práce, její konzultace ohledně práce a trpělivost během vytváření práce. Dále také mým rodičům, sourozencům a nejbližším přátelům za jejich ochotu zhodnotit práci z jejich pohledu

Abstrakt

V této práci představuji způsob vypočítání a zobrazení složených těles v reálném čase, a tedy také způsob vypočtení řezů těles tělesy.
Cílem práce je popsat nový algoritmus, jež jsem vymyslel, na výpočet řezů těles tělesy a vykreslení složených těles (někdy také nazývaných jako Booleovská tělesa) dynamicky v reálném čase.

Algoritmus jsem implementoval v počítačové aplikaci, kdy jsem využil dnešního výkonu grafických akcelerátorů k urychlení výpočtu tak, aby jej bylo možné využít v reálném čase a plně dynamicky.

V první části práce zmíním různé způsoby na výpočet složených těles, následně vysvětlím na jakém principu nový algoritmus funguje. V následujících částech se věnuji jeho implementaci a výsledkům naší práce.

Na závěr navrhuji pár metod jak by bylo možné algoritmus rozšířit k praktickému využití a případně rozšířit i ke komerční aplikaci například v procesu vývoje her a 3D modelování.

Abstract

I introduce another way of computing and rendering of compound bodies in real time and thus a way to solve slicing bodies with bodies.

The purpose of this work is to describe a new algorithm, which I created, to compute slicing of bodies with bodies and to render compound objects (sometimes also named Boolean objects) dynamically in real time.

This algorithm has been implemented in a computer application, where I used the computing capacity of current graphics accelerators for acceleration of computing to achieve everything dynamically in real time.

In this first part of this work I'm describing several ways how the compound objects can be computed in several different ways, and then I describe on what principle the new algorithm works. In the further chapters I talk about my implementation and the results of the algorithm.

In the end I propose several solutions, how could be the algorithm further extended, even for practical usage and maybe extended to a commercial application in the process of game development and 3D modeling.

Obsah

1. Úvod.. ..1
1.1 Předešlé koncepty 2
2. Metodika 4
2.1 Stručný popis algoritmu 4
2.2 Implementace 9
3. Výsledky 15
4. Závěr a diskuse 17
5. Reference 18
6. Přílohy.. 19
6.1 Příloha č.1 19
6.2 Příloha č.2 20
6.3 Příloha č.3 24

1. Úvod

Na úvod bych rád nejprve vysvětlil co to tzv. složená tělesa jsou. Jedná se o těleso, které je složené z více těles, ovšem mezi tělesy navzájem jsou provedeny různé operace. Můžeme tak třeba z krychle vyříznout část koule, nebo k válci přidat elipsoid tak, aby nám vznikl tvar nábojnice. Složenými tělesy tak lze vymodelovat prakticky jakýkoliv složitější objekt, často také rychleji než když jej modelujeme pomocí trojúhelníků.

Obr.1 Příklad složitějšího Složeného tělesa

Tato práce se zabývá problémem zobrazení a počítání složených těles interaktivně v reálném čase, a tedy by bylo možné v budoucnu využít zde popsanou metodu v různých aplikacích, především v nástrojích sloužících k modelování 3D objektů, kde prozatím žádný dnes více rozšířený software neobsahuje interaktivní modelování složených těles, ač by se při návrzích architektonických prvků či samotných staveb, anebo v procesu herního vývoje jistě velmi hodily.

Momentálně je k dispozici několik balíků modelovacího software, který je schopen pracovat s Složenými objekty, avšak nikoliv interaktivně a v reálném čase je během práce není možno editovat, jedná se například o aplikace Autodesk 3DS Max, Autodesk Maya, či Maxon Cinema4D. Dále je možné dohledat a použít (či spíše napsat nové) voxelové enginy, které by byly schopné pracovat s složenými objekty na úrovni voxelů také v reálném čase a interaktivně, informace k tomuto tématu lze dohledat na stránkách serveru GameDev.net [GD99], případně také fórech OMPF.org [OM00], anebo také na webových stránkách uživatele Spacerat [SP09], který se zaobíral tématem vykreslování voxelů a samozřejmě také modelování Složených těles v jazyce CUDA.

Ohledně průniků mezi tělesy a tedy tématem na kterém jsou složená tělesa založená se lze dozvědět informace na serveru realtimerendering.com [RTR08] a jejich knize [AKM08]. Další velmi významnou osobností v poli voxelového vykreslování a tedy také v poli složených objektů a jejich zobrazování v reálném čase je Ken Silverman [SIL99].

1.1 Předešlé koncepty

1. Způsobů, jak bychom mohli dosáhnout výpočtu a zobrazení složeného tělesa není příliš mnoho. Nejjednodušší z nich využívá voxelů – tedy 3D pixelů umístěných v prostoru, lze si je představit jako velmi malé krychličky uložené do obrovského prostoru, kde odebereme veškeré, které tam nepotřebujeme … lze z nich vymodelovat prakticky vše, ovšem musejí být velmi drobné v případě že chceme modelovat křivé plochy.

   Výroba složených objektů v obrovském voxelovém prostoru je tedy jednoduchá, prvně si v něm označíme voxely spadající objektu A a poté voxely objektu B. Nyní procházíme voxel po voxelu a zjišťujeme zda je součástí objektu A, objektu B, anebo obou. Pokud je součástí jednoho nebo obou z nich tak jej buď označíme že ve finálním objektu je, nebo není (podle operací zmíněných výše – tedy Subtraction, Union, či Intersection).

   Hlavním problémem je samozřejmě počet voxelů v poli, aby byly plochy dostatečně hladké – je třeba zajistit aby voxely byly opravdu drobné, a to je zase velmi náročné na paměť a výpočet u dnešních počítačů, ne však nemožné.

Obr.2 Voxelový model terénu, který je složeným tělesem, byly od něj odečteny různé velikosti koulí. Stromy jsou zase složeným tělesem vzniklým unií různých objektů

2. Dalším způsobem, dneska prakticky nejpoužívanějším je provést řez a booleovskou operaci na trojúhelníkových modelech (které jsou dnes, na rozdíl od voxelových standardem).

   Vezměme si dva modely položené do prostoru tak, aby se navzájem prolínaly. Nyní považujme každý trojúhelník za rovinu, která je samozřejmě ohraničena, a proveďme řez každého trojúhelníku s každým trojúhelníkem, obecně nám vzniknou na místech, kde se objekty protínají, mnohoúhelníky.

   Dalším krokem je třeba projít všechny trojúhelníky a mnohoúhelníky (obecně všechny několikaúhelníky, dále N-úhelníky) tělesa A a označit je zda jsou pouze na povrchu tělesa A, či uvnitř tělesa B.
   Dále projdeme všechny N-úhelníky tělesa B a označíme zda jsou pouze na povrchu tělesa B, či uvnitř tělesa A.

   Nyní záleží na složené operaci, pokud je operací Union – odstraníme veškeré N-úhelníky označené jako uvnitř tělesa A anebo označené jako uvnitř tělesa B. Ty, které zbyly jsou součástí nového složeného tělesa.
   Pokud je operací Intersection, odstraníme veškeré N-úhelníky na povrchu tělesa A, či na povrchu tělesa B, které však nebyly uvnitř žádného tělesa.

   A pokud je operací Subtraction (předpokládejme že odečítáme těleso B od tělesa A), odstraníme N-úhelníky označené jako uvnitř tělesa B a N-úhelníky označené jako na povrchu tělesa B (nikoliv však uvnitř tělesa A), výsledkem je zase složené těleso.

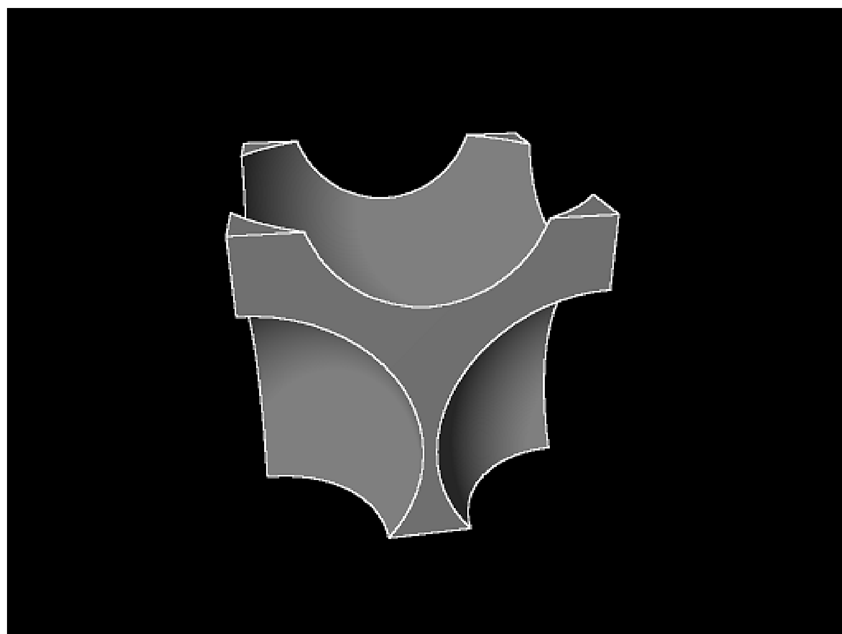
Tato metoda je velice účinná, a od první není ani příliš paměťově náročná. Ovšem výpočetní náročnost je již velmi vysoká a provádět ji v reálném čase je velmi náročné.

Rozhodl jsem se proto prozkoumat ještě něco trochu jiného a dosáhnout algoritmu, který bude běhat v reálném čase a nebude příliš výpočetně náročný.

2. Metodika

2.1 Stručný popis algoritmu

Algoritmus, který jsem navrhnul, je na rozdíl od předchozích silně optimalizován pro vyřešení jak paměťové náročnosti, tak výpočetní náročnosti. Všech těchto problémů se zbavuje odložením velké části práce na grafický procesor, jehož výpočetní síla nemá s podobnými, vysoce náročnými aplikacemi problémy.



Obr.3 Algoritmus je schopen při správné implementaci zvládnout velice komplexní složená tělesa složená z mnoha těles

Celý algoritmus se ale velmi liší od těch výše zmíněných, je zamýšlet především pro vykreslování složených objektů a už méně příjemný pokud bychom je chtěli ukládat. Ovšem ukládání podobných objektů, až na lehké nastínění v závěru není součástí této práce, jelikož samo o sobě je docela rozsáhlé a svým rozsahem by možná tuto práci i předčilo.

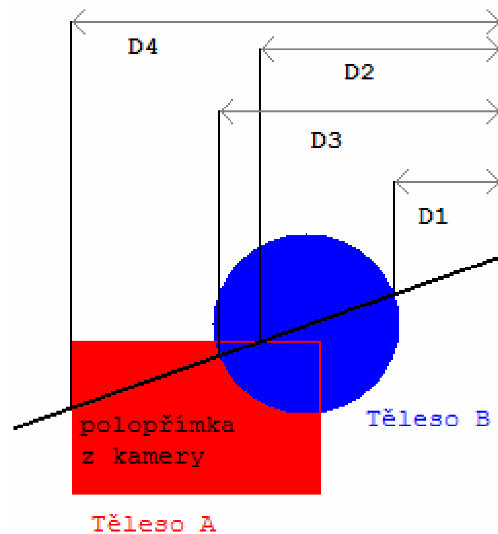
Je založen na nápadu „co kdybychom vypočítávali složený objekt až z prostoru pozorovatele“ – tedy veškerý výpočet tělesa je proveden až ve výsledku, když se díváme z pohledu kamery. Metoda tak vlastně využije metody sledování polopřímek, ale zde potřebujeme však pouze primární polopřímky – tedy zjistit v jaké vzdálenosti jsme strefili první těleso, v jaké další těleso, atd. Tuto informaci ale získáme také rasterizací a následným zapnutím hloubkového testu, informaci kterou požadujeme budeme mít uvnitř hloubkového bufferu.

A tady využijeme naši největší výhodu, OpenGL je knihovna, která provádí rasterizaci hardwarově na grafické kartě, je tedy velmi rychlá, a k tomu máme možnost s veškerými hodnotami pracovat v pixel shaderech (v OpenGL oficiálně pojmenovány také fragment shadery).

Pro vysvětlení rasterizace je proces, kdy se na obrazovku či do bufferu promítnou veškeré N-úhelníky a body a vykreslí. Hloubkový test se stará o to, aby ty nejbližší N-úhelníky byly skutečně před těmi vzdálenějšími a ne naopak (nejjednodušší případ je tzv. Painter’s algorithm – neboli kreslířův algoritmus, kdy kreslíme odzadu, bližší N-úhelník vždy překryje ten vzdálenější, má však velmi mnoho nevýhod a někdy špatnou kvalitu, proto se dnes využívá tzv. hloubkový buffer a test na úrovni pixelů). Shader je program, který se kompiluje za běhu programu a jehož veškeré instrukce provede potom karta na tělese - jeho geometrii, či pixelech – a o veškerý výpočet se stará grafická karta, která je stavěna k těmto operacím v opravdu masivním počtu.

V základu algoritmu jde tedy o porovnávání hloubek na jednotlivém pixelu a zjišťování zda pixel tělesa A je uvnitř tělesa B a označování pixelů, potom vybrání správných pixelů z bufferů a následné stínování pomocí tzv. deferred renderingu/deferred shadingu.

Podívejme se nyní na graficky-geometrickou reprezentaci algoritmů, která nám dá přesnou představu o tom co se děje. Na obrázku lze vidět čtyři různé vzdálenosti průniků od bodu počátku polopřímky (bodu kamery).



Obr.4 Polopřímka z kamery a její průniky s tělesy

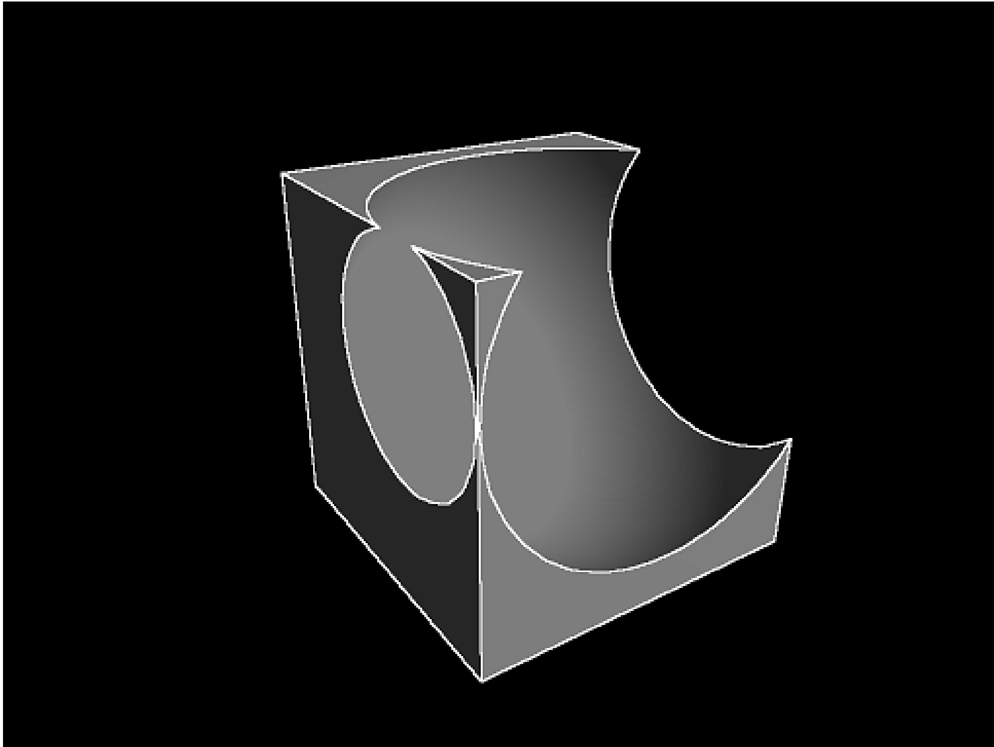
Např. Subtrakci lze nyní zapsat matematicky takto, tedy vracíme vstupní bod do tělesa A (označen A s indexem f – jako přední stěna = front face) právě když vzdálenost D2 (vzdálenost přední stěny tělesa A) je menší nebo rovna vzdálenosti D1 (vzdálenost přední stěny tělesa B).

Výstupní bod z tělesa B (označen B s indexem b – jako zadní stěna – back face) vracíme právě když vzdálenost přední stěny tělesa B je menší než vzdálenost přední stěny tělesa A (D1 < D2) a zároveň musí platit, že vzdálenost zadní stěny tělesa B je větší než vzdálenost přední stěny tělesa A (D3 > D2) a zároveň také musí platit, že vzdálenost zadní stěny tělesa B je nižší než vzdálenost zadní stěny tělesa A (D3 < D4).

Hodnotu 0 vracíme zase právě když v daném bodě složené těleso není – tedy pokud mineme polopřímkou z kamery kterékoliv tělesa utvářející Složené těleso (tedy D3 je rovno nekonečnu), anebo pokud platí, že vzdálenost přední strany tělesa B je menší než vzdálenost přední strany tělesa A, a zároveň vzdálenost zadní strany tělesa B je větší než vzdálenost přední strany tělesa A, a zároveň neplatí, že vzdálenost zadní strany tělesa B je menší než vzdálenost zadní strany tělesa A:

$$x = A_f; D_2 <= D_1$$
$$x = B_b; D_1 < D_2 \wedge D_3 > D_2 \wedge D_3 < D_4$$
$$x = 0; D_3 = \infty \vee D_1 < D_2 \wedge D_3 > D_2 \wedge !(D_3 < D_4)$$

Výsledkem tohoto výpočtu bude složené těleso vzniklé odečtením tělesa B od tělesa A, výsledek v případě koule a krychle může vypadat takto:



Obr.5 Složené těleso vzniklé odečtením

Pokud se podíváme ne předešlý obrázek s polopřímkami, tak je vyřešení sečtení (také nazývané unie) objektů velmi jednoduché, matematicky tedy vracíme přední stěnu tělesa B právě když, vzdálenost k přední stěně tělesa B, D1, je větší než vzdálenost k přední stěně tělesa A, tedy D2.

Přední stěnu tělesa A vracíme právě když, přední stěna tělesa A je blíže k bodu kamery než přední stěna tělesa B – tedy D2 je menší (nebo rovno, ať pokryjeme veškeré hodnoty) než D1.

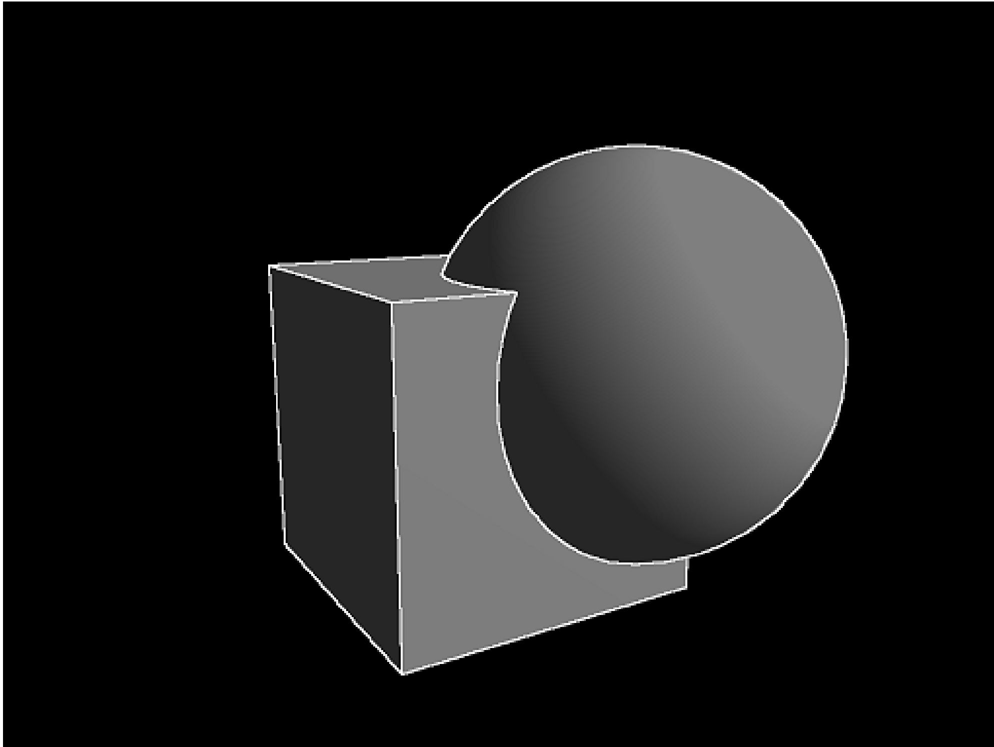
Pokud mineme obě tělesa, je třeba samozřejmě vrátit nulu.

$$x = B_f ; D_1 < D_2$$

$$x = A_f ; D_2 <= D_1$$

$$x = 0; D_1 = \infty \wedge D_2 = \infty$$

Výsledek složené unie může vypadat následovně:



Obr.6 Složené těleso vzniklé sečtením (unií)

Vraťme se ještě naposled k našemu geometricky-schématickému obrázku s polopřímkami, protože nám zbývá vyřešit poslední případ – a to intersekci, neboli Složené těleso vzniklé v prostoru kde jsou tělesa v průniku.

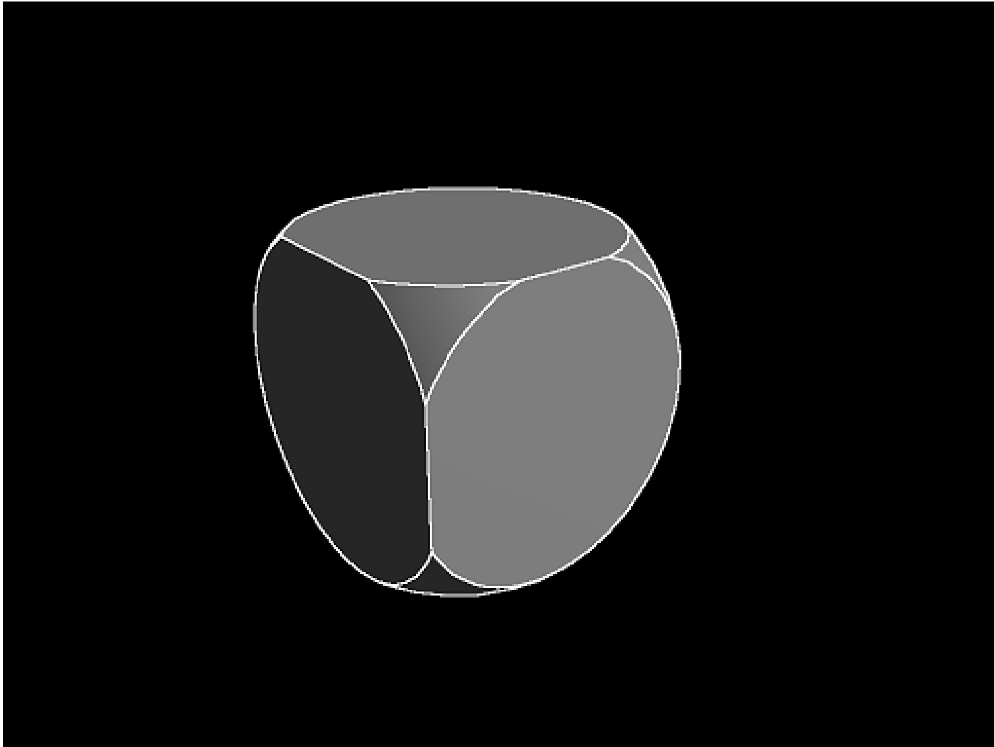
Matematicky je toto těleso nejsložitější, zpravidla je nejjednodušší nejprve říct že máme nulový průnik a poté až testovat zda jsme zrovna strefili průnik těles.

Přední stranu tělesa A vracíme tehdy, když vzdálenost k přední straně tělesa B je menší než vzdálenost k přední straně tělesa A (D1 < D2) a zároveň vzdálenost přední strany tělesa A je menší než vzdálenost k zadní straně tělesa B (D2 < D3).

Přední stranu tělesa B vracíme právě tehdy pokud platí, že vzdálenost k přední straně tělesa B je větší než vzdálenost k přední straně tělesa A (nebo rovna, abychom pokryli veškeré možnosti), a tedy D1 >= D2. Zároveň však musí platit, že vzdálenost k přední straně tělesa B je menší než vzdálenost k zadní straně tělesa A (D1 < D4), tedy takto:

$$x = \infty$$
$$x = A_f; D_1 < D_2 \wedge D_2 < D_3$$
$$x = B_f; D_1 >= D_2 \wedge D_1 < D_4$$

Výsledkem bude složené těleso vzniklé v prostoru, kde jsou tělesa utvářející jej navzájem v intersekci – průniku:



Obr.7 Složené těleso v místě průniku dvou těles

Nyní byste měli mít hrubou představu o přibližném algoritmu na zobrazení složených těles, jež jsem navrhnul a vytvořil. V následující kapitole popíšu některé aspekty mé implementace a také jak jsem algoritmus co nejvíce optimalizoval.

2.2 Implementace

Celá má implementace byla provedena v jazyce C/C++ za používání především knihovny OpenGL, společně s WinAPI k vytvoření okna. K implementaci jsme jako cílovou platformu zvolili Microsoft Windows díky jeho velmi rozsáhlé popularitě a jednoduchosti práce s tímto operačním systémem. Knihovna OpenGL byla zvolena především kvůli jak rychlosti této knihovny, tak jednoduchosti implementace software na ní. Také ji preferuji před knihovnou DirectX (která je možnostmi na stejné úrovni) právě proto, že se mě s ní lépe pracuje.

Největším prvotním problémem bylo rozhodnutí, zda využít pro získání vstupních a výstupních vzdáleností (a tedy bodů) metodu sledování polopřímek (ray tracing) v reálném čase, anebo využít multiple render targets (dále MRT), framebuffer objects (dále FBO) a depth peeling na grafické kartě – což podle prvního nápadu mohlo veškeré výpočty velice urychlit.

Pro doplnění informací, FBO je framebuffer objekt (tedy buffer do kterého kreslíme), ovšem umístěný pouze v paměti grafické karty a není kreslen na obrazovku.

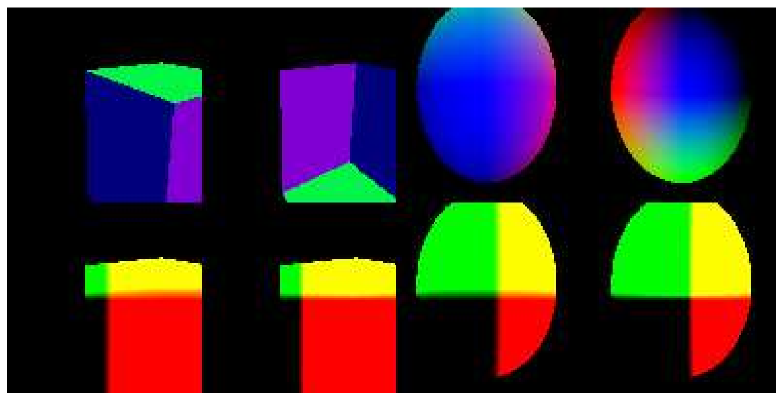
Jedná se tedy o tzv. mimoobrazovkový buffer. Mimoobrazovkový buffer lze uložit do textury (tzv. render-to-texture, nebo také render target). Pokud z framebuffer objektu zapisujeme do více textur najednou (během jediné fáze vykreslení), jedná se o takzvaný multiple-render-target, nebo-li MRT. Pro více informací ohledně framebuffer objektů v OpenGL doporučuji navštívit registr specifikace OpenGL pro rozšíření EXT_framebuffer_object [GL04], kde je také detailnější popis co vlastně FBO je.

Ohledně MRT je dobré podívat se na odkaz pro FBO [GL04], ale také navštívit registr specifikace OpenGL pro rozšíření GL_ARB_draw_buffers [GL02].

Depth peeling je metoda, která nám umožní zajistit, aby se mohly Složené tělesa správně spočíst i v případě, že máme více těles za sebou ležících na jedné polopřímce z bodu kamery. Pro zjednodušení však tuto metodu nebudu rozebírat, neboť její implementace a optimalizace je velmi složitá a samotný popis by se nemusel vejít do rozsahu práce. O depth peelingu bylo napsaných několik prací, především bych doporučil publikaci NVidie o metodě dual depth peeling [NV08], případně publikaci Microsoft corp. [MS06].

Vraťme se nyní k samotné implementaci, rozhodnul jsem se využít grafických karet především kvůli jejich výpočetní síle, a tedy jsem se zaměřil na využití FBO, MRT a shader programů během tohoto výpočtu.
Prvním problémem, který bylo potřeba vyřešit jsou data, jež potřebujeme ukládat a také jak získáme potřebná data.

V první fázi potřebujeme především získat data, prozatím nám postačí pozice a normály předních a zadních stěn obou těles (vzdálenosti si lze vyjádřit z jejich pozice). Tedy pokud máme 2 tělesa, využijeme FBO celkem 4x a pokaždé využijeme MRT – ovšem kreslit budeme pouze do 2 bufferů. Celkem se tedy dostáváme na zisk 8 bufferů (textur) o velikosti obrazovky, výstup bufferů vypadá takto:



Obr.8 Výstup veškerých textur po fázi vytváření dat

O způsobu uložení dat a jejich zisku je tedy jasno, v příloze č. 1 je také ukázán celý kód shader programu k získání/vypočtení dat do těchto bufferů. Jen je potřeba malý dodatek, v případě že tělesa odečítáme, tak je třeba v obou bufferech tělesa jež odečítáme

od druhého nakreslit do zadní ořezávací roviny rovinu, abychom neporovnávali v algoritmu nulu s nulou.

Dále je potřeba vyřešit samotná tvorba a zobrazení složeného tělesa, a tedy zapsat výpočet zmíněný v stručné implementaci pomocí shader programu (a tedy v jazyce GLSL = OpenGL Shading Language).

Tedy ukažme si subtrakci těles A-B. Rovnice jsem v předešlé kapitole zapsal takto:

$$x = A_f; D_2 <= D_1$$
$$x = B_b; D_1 < D_2 \wedge D_3 > D_2 \wedge D_3 < D_4$$
$$x = 0; D_3 = \infty \vee D_1 < D_2 \wedge D_3 > D_2 \wedge !(D_3 < D_4)$$

V případě, že naše buffery a proměnné si zapíšeme takto:

```
// Uložení textur na daném pixelu v shaderu
// Pozice přední stěny tělesa A
vec3 Af_pos = texture2D(Af, v2_texCoord_gout).xyz;
// Pozice zadní stěny tělesa A
vec3 Ab_pos = texture2D(Ab, v2_texCoord_gout).xyz;
// Pozice přední stěny tělesa B
vec3 Bf_pos = texture2D(Bf, v2_texCoord_gout).xyz;
// Pozice zadní stěny tělesa B
vec3 Bb_pos = texture2D(Bb, v2_texCoord_gout).xyz;
// Vzdálenost k přední stěně tělesa A
float Af_d = length(Af_pos);
// Vzdálenost k zadní stěně tělesa A
float Ab_d = length(Ab_pos);
// Vzdálenost k přední stěně tělesa B
float Bf_d = length(Bf_pos);
// Vzdálenost k zadní stěně tělesa B
float Bb_d = length(Bb_pos);

// Výsledná pozice
vec3 booleanPos = vec3(0.0, 0.0, 0.0);
// Výsledná normála
vec3 booleanNorm = vec3(0.0, 0.0, 0.0);
```

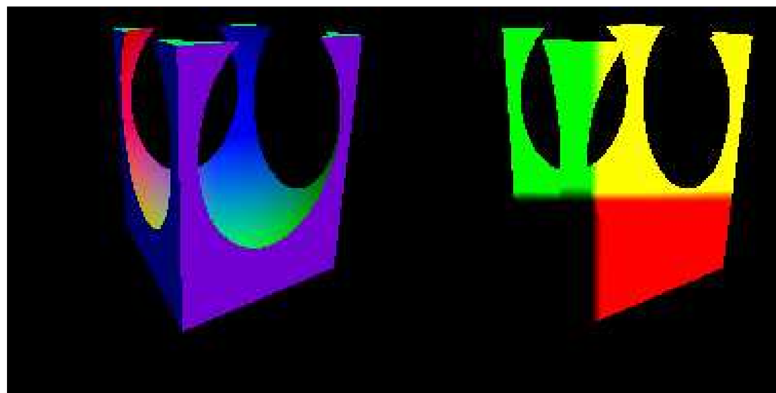
Kdy dodáváme data do shader programu takto:

```
// Buffery dodávané do shaderu (tedy všech 8 textur)
uniform sampler2D Af;
uniform sampler2D Ab;
uniform sampler2D Bf;
uniform sampler2D Bb;
uniform sampler2D Af_n;
uniform sampler2D Ab_n;
uniform sampler2D Bf_n;
uniform sampler2D Bb_n;

// Texturové koordináty vystupující z Geometry shader fáze pipeline
varying in vec2 v2_texCoord_gout;
```

Tímto bychom měli popsané potřebné data, algoritmus můžeme zapsat dle rovnice výše zmíněné takto:

```
// Do pozic i normál uložíme těleso A
booleanPos = Af_pos;
booleanNorm = texture2D(Af_n, v2_texCoord_gout).xyz;
// Pokud vzdálenost přední stěny tělesa B je menší než vzdálenost
// přední stěny tělesa A a zároveň vzdálenost zadní stěny tělesa
// B je větší než vzdálenost přední stěny tělesa A
if(Bf_d < Af_d && Bb_d > Af_d)
{
	// pokud vzdálenost zadní stěny tělesa B je menší než vzdálenost
	// zadní stěny tělesa A
	if(Bb_d < Ab_d)
	{
		// Pozice i normála jsou získány ze zadní stěny tělesa B
		booleanPos = Bb_pos;
		booleanNorm = texture2D(Bb_n, v2_texCoord_gout).xyz;
	}
	// v opačném případě mineme objekt
	else
	{
		// Pozice i normála jsou nulové
		booleanPos = vec3(0.0, 0.0, 0.0);
		booleanNorm = vec3(0.0, 0.0, 0.0);
	}
}
```

Nyní provádíme výstup do dalšího framebuffer objektu, který obsahuje pozice a normály již Složeného tělesa, v další fázi již bude velmi jednoduché těleso nasvětlit pomocí metody Deferred shading (více informací o metodě deferred shading lze vyčíst ze staršího dokumentu vydaného NVidia [NV04]), či zvýraznit obrys tělesa. Výstup po této fázi vypadá takto:



Obr.9 Výstup po fázi vypočtení Složeného tělesa

Shader programy pro vypočtení složeného tělesa jsou zapsány v příloze č. 2

Následujícím krokem bude už pouze fáze stínování, tedy výpočet osvětlení a v našem případě také výpočet obrysu tělesa (vyjdeme z edge-detection filtru, tzv. Sobel operátoru).

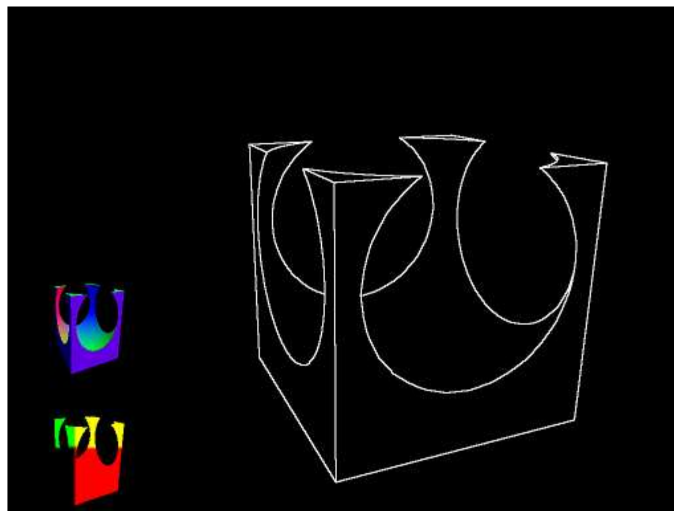
Výpočet obrysu tělesa je velmi jednoduchý, stačí uvnitř shader programu přečíst okolní pixely v bufferu, který drží texturu obsahující normály (normály dají dle mého názoru hezčí obrysový výsledek než pozice) a provést operaci na dvou maticích s RGB složkami textury takto:

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$ - vertikální matice

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$ - horizontální matice

Veškeré složky matice potom sečíst, získáme 2 RGB hodnoty z horizontální a vertikální matice, na těch sečteme absolutní hodnoty jednotlivých složek a máme výsledek (od něj je potřeba odečíst nějakou malou hodnotu, abychom dostali pěkný výsledek obrysu).

Výsledek po sobel operátoru vypadá následovně:



Obr.10 Edge-detection filtr na získání obrysu Složeného tělesa

Osvětlení pomocí metody deferred shading také není nic složitého, jelikož normály i pozici už máme je potřeba dodat jen pozici světla ve správném prostoru. Normály i pozice bodů jsou dodány relativně k bodu kamery, tak je třeba dodat pozici světla relativně k bodu kamery. Toho docílíme tím, že vynásobíme pozici světla (od které odečteme pozici kamery) transformační maticí kamery (modelview matice kamery) a také projekční maticí (matice, která udává velikost bufferu který zobrazíme na obrazovce a také obsahuje hodnoty blízké a vzdálené ořezávací roviny).

Na výpočet světla použijeme velmi jednoduchý Lambertův model, který je na rozdíl od Oren-Nayar modelu nekorektní, ovšem je velmi rychlý a pro naše účely dostačující.

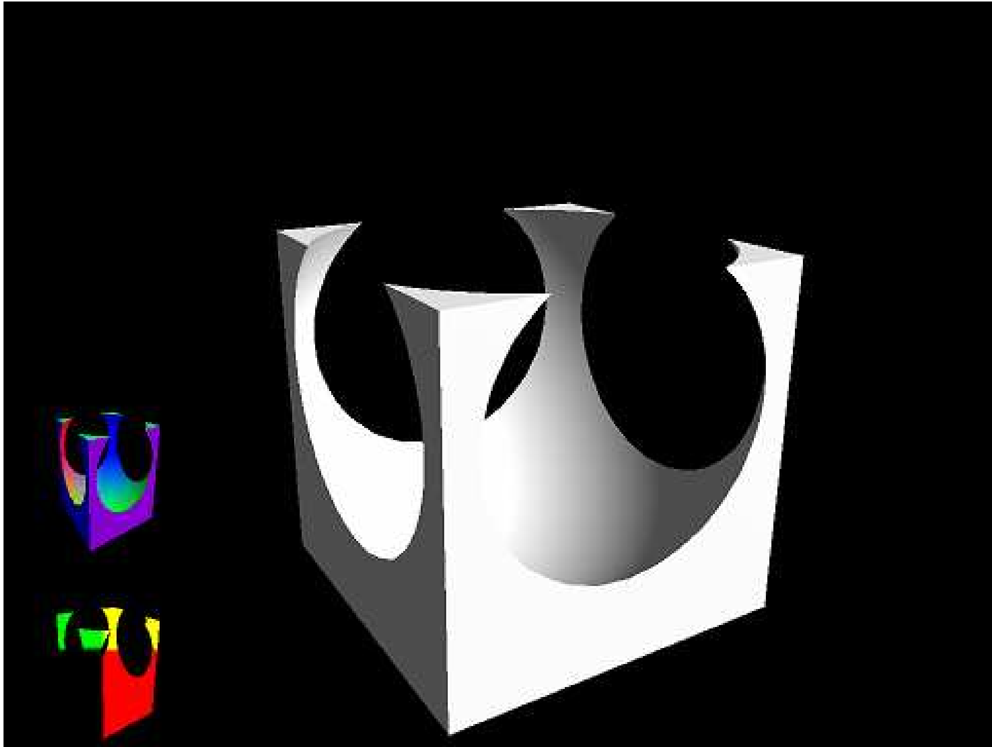
Data, která dodáváme do shader programu, který se postará o deferred shading:

```
// Textury pozicí a normál
uniform sampler2D final_position;
uniform sampler2D final_normal;

// Texturové koordináty a pozice světla, obě hodnoty vystoupili
// z geometry shader fáze renderovací pipeline
varying in vec2 v2_texCoord_gout;
varying in vec3 lightPosMV_gout;
```

A zdrojový kód samotného výpočtu:

```
// Uložení hodnot textur pozicí a normál na pixelu, kde momentálně
// pracujeme
vec3 normalTex = texture2D(final_normal, v2_texCoord_gout).xyz;
vec3 positionTex = texture2D(final_position, v2_texCoord_gout).xyz;

// Vektor směru světla - normalizovaný vektor směrem od pozice
// světla k pozici na daném pixelu
vec3 lightDir = normalize(lightPosMV_gout - positionTex);
// Dopadové světlo spočteme Lambertovým modelem, skalární součin
// normalizovaného vektoru normály a vektoru směru světla k bodu
// nám dá kosinus úhlu mezi nimi, pokud bude menší než 0, necháme
// jej na hodnotě 0. V opačném případě bude výstupem dopadové
// osvětlení
// Tedy pixely přivrácené ke světlu budou osvětleny maximálně a s
// růstem úhlu normály k vektoru světla bude intenzita světla
// ubývat
float diffuseLight = max(dot(lightDir, normalize(normalTex)), 0.0);
```

Výsledek po osvětlení metodou deferred shading bude vypadat následovně:



Obr.11 Výstup po osvětlení metodou Deferred shading

K osvětlení na obrázku jsem připočetl také nějaké rozptýlené světlo, aby byl objekt zvýrazněn oproti pozadí. Na boku pořád zobrazuji výstup bufferu po vypočtení Složeného tělesa.

Výstupní shader program provádějící deferred shading a detekci obrysu tělesa je přidán v příloze č. 3.

Nyní by měla být implementace algoritmu jasná, sami jsme jej implementovali popsaným způsobem a dosáhli jsme velmi pozitivních výsledků. Více k nim v následující kapitole, kde si rozebereme náročnost výpočtu a jeho efektivitu.

3. Výsledky

Program je sice silně optimalizován, ovšem ve výsledku nemá příliš rozsáhlou kompatibilitu. Vyžaduje rozšíření EXT_framebuffer_object a ARB_draw_buffers, spolu s tím velmi velkou výpočetní sílu u grafické karty (tedy grafické karty navrhnuté s fixním počtem pixel a vertex shader jednotek můžeme rovnou vyloučit) – je kompatibilní s kartami GeForce 8 série a vyšší, či kartám se stejnými či vyššími možnostmi od stejného, či jiného výrobce.

Ač se může zdát, že by program byl těžký především pro shadery, není to zase až tak pravdivé. Program je spíše velmi náročný na fill-rate (nebo-li schopnost karty zaplňovat buffery a textury). Podívejme se na tabulku fill-rate hodnot:

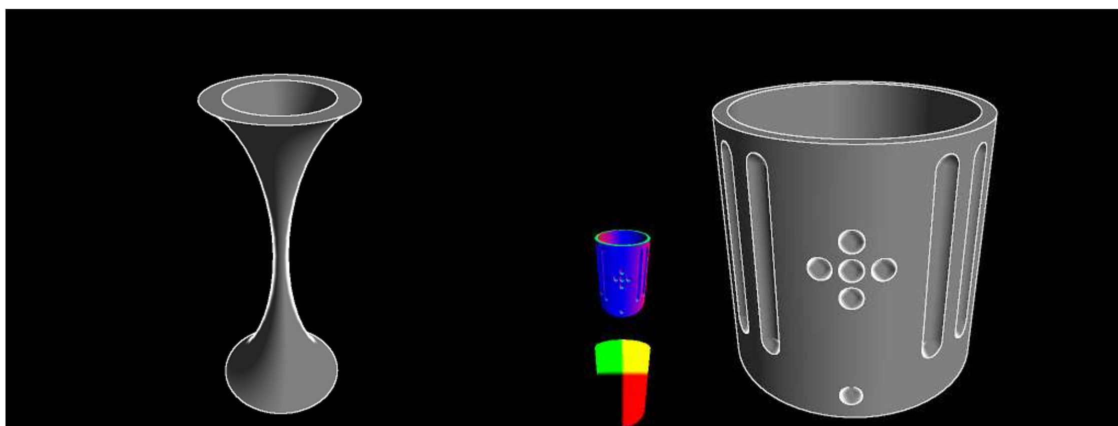
Rozlišení	Počet pixelů	Bitů na pixel	Fillrate pix/frame	Fillrate Mpix/s @ 60fps
800x600	480000	992	476160000	28569,6
1024x768	786432	992	780140544	46808,43264
1280x1024	1310720	992	1300234240	78014,0544
1280x720	921600	992	914227200	54853,632
1440x900	1296000	992	1285632000	77137,92
1920x1080	2073600	992	2057011200	123420,672

Tab.1 – náročnost na fillrate grafického akcelerátoru

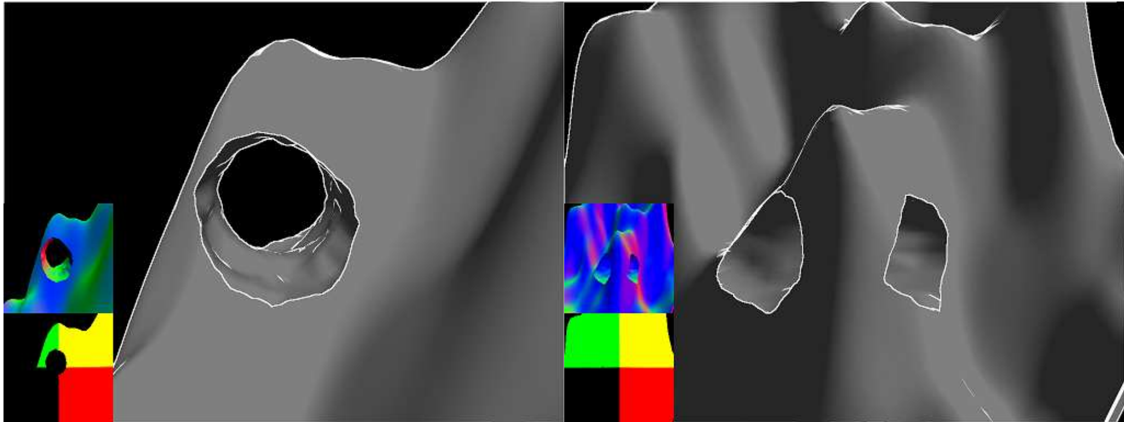
První sloupec udává rozlišení obrazovky. Druhý sloupec celkový počet pixelů na obrazovce. Ve třetím sloupci je uveden počet bitů, které musí karta zaplnit na každém pixelu (máme 5 bufferů s RGBA32F texturou – tedy texturou, která má 32bitové číslo s plovoucí desetinnou čárkou na každý kanál své složky, 5 bufferů s RGBA16F texturou – textura, která má 16bitové číslo s plovoucí desetinnou čárkou na každý kanál své složky a 1 buffer, výstupní, který má RGBA8 plochu – tedy plocha, která má 8bitové celé číslo na kanál. Fillrate je udávám pouze pro barevné složky (a nikoliv hloubkové buffery, které by se také měly započítat a zvedly by jej o něco).

Každopádně při typickém rozlišení 1024x768 pixelů je potřeba karta se schopností zaplnit 46808,43264 milionů pixelů každou sekundu (za předpokladu že kreslíme při 60 snímcích za sekundu, tedy 60krát do sekundy překreslíme veškerý snímek). Samozřejmě to je teoretická hodnota v případě že bychom měli složená tělesa po celé obrazovce. Dále tato hodnota se dělí do framebuffer objektů a tím se redukuje mezi jednotlivé framebuffer, takže ve výsledku to není až tak hrozivé.

Některé Složené modely, které jsme pomocí naší aplikace zobrazili:



Obr.12 Vlevo svícen vymodelován jako Složené těleso, vpravo zdobená sklenička modelována jako složené těleso s mezibuffery



Obr.13 Vlevo i vpravo terén s vymodelovaným tunelem jako Složené těleso, obojí s mezibuffery.

4. Závěr a diskuse

Představil jsem novou metodu na výpočet a zobrazení složených těles, který lze využít k interaktivnímu modelování takových objektů a práci s nimi v reálném čase.

Výsledky tohoto algoritmu jsou velmi uspokojivé, na testovacím hardware si algoritmus vedl velmi rychle a nebyl problém pracovat se složitými modely interaktivně v reálném čase.

Dalšími kroky v této aplikaci by mohly vést k vytvoření modelovacího software zaměřený především na Složená tělesa, který by našel uplatnění jak v modelování architektonických vizualizací a návrzích, tak samozřejmě v oboru herního vývoje.

Aplikace prozatím trpí pár nedostatky, jde především o:

1. V případě, že máme za sebou 2 tělesa která odečítáme od nějakého, mohou se vyskytovat artefakty a výsledek by mohl být nepřesný. Tento problém by šel velmi elegantně vyřešit implementací depth peelingu a tím pádem navýšením počtu bufferů, které u výpočtu použijeme. Bohužel by to také znamenalo nárust náročnosti aplikace na grafickou kartu.
   Dalším možným řešením by bylo využití prokládaných bufferů, které by umožnili uložit až 4 vrstvy do jediného bufferu. Ovšem za cenu nepatrné ztráty kvality, oproti prvnímu návrhu řešení.

2. V případě, že dodáme neuzavřený model – nastává problém, protože v některých místech model neobsahuje přední či zadní stěny a dojde k nepřesnému výpočtu složeného tělesa. Toto není problém aplikace, ale spíše dodaných modelů, protože nelze provádět výpočet Složených těles na neuzavřených tělesech.

3. Aplikace neumožňuje uložení modelu po jeho vytvoření. Tento problém je jeden z největších, jelikož je velmi podstatný, pokud bychom chtěli v budoucnu nějaké modely vytvářet v podobné aplikaci.
Také zde je několik způsobů řešení, zřejmě nejlepším je při ukládání převést veškerou zobrazovanou scénu na voxelizovaný model, tedy získat Složené těleso do voxelů a pomocí některého z algoritmů převést voxely zpětně na trojúhelníkové modely, které jsou z praktického hlediska lepší.
Bohužel však i stručný popis podobného způsobu by zabral minimálně stejnou, ne-li větší práci než je tato. Takže nelze zde vypisovat detaily ohledně možnosti ukládání modelu.

5. Reference

[GD99] GameDev.net: Elektronický portál a diskusní fórum o herním vývoji a programování [online]. Dostupné z URL: http://www.gamedev.net

[OM00] OMPF.org: Elektronické diskusní fórum o ray tracingu, voxelech a dalších tématech 3D grafiky [online]. Dostupné z URL: http://www.ompf.org/forum/

[SP09] Voxels blog: CUDA voxels engine [online]. Dostupné z URL: http://voxels.blogspot.com/

[RTR08] Realtimerendering.com: Object/object intersection [online]. Dostupné z URL: http://www.realtimerendering.com/intersections.html

[AKM08] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. Real-time rendering 3rd edition. A. K. Peters, Ltd., 2008. ISBN 987-1-56881-424-7

[SIL99] SILVERMAN, K. Ken Silverman's Voxlap Engine [online]. 2002. Dostupné z URL: http://advsys.net/ken/voxlap.htm

[GL04] KHRONOS group, OpenGL extensions registry – GL_EXT_framebuffer_object [online]. 2004. Dostupné z URL: http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt

[GL02] KHRONOS group, OpenGL extensions registry – GL_ARB_draw_buffers [online]. 2002. Dostupné z URL: http://www.opengl.org/registry/specs/ARB/draw_buffers.txt

[NV08] NVidia corp., Order Independent Transparency with Dual Depth Peeling [online]. 2008. Dostupné z URL: http://developer.download.nvidia.com/SDK/10.5/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf

[MS06] Microsoft corp., Multi Layer Depth Peeling via Fragment sort [online]. 2006. Dostupné z URL: http://research.microsoft.com/apps/pubs/default.aspx?id=70307

[NV04] Nvidia corp., Deferred shading [online]. 2004. Dostupné z URL: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf

6. Přílohy

6.1 Příloha č. 1

Shader k získání dat pozice a normál a jejich uložení do render target.
Vertex shader:

```
// Budeme využívat Shader Model 4.0 a GLSL verze 1.20
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Výstup pozic a normál z vertex shaderu
varying out vec3 v3_vertex_vout;
varying out vec3 v3_normal_vout;

// Main
void main()
{
	// Získání pozic relativně k pohledu kamery
	v3_vertex_vout = vec3(gl_ModelViewMatrix * gl_Vertex);
	// Získáni normál relativně k pohledu kamery
	v3_normal_vout = gl_NormalMatrix * gl_Normal;

	// Transformace bodů podle modelview projekční matice OpenGL
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Geometry shader:

```
// Budeme využívat Shader Model 4.0 a GLSL 1.20
#version 120
#extension GL_EXT_gpu_shader4 : enable
// Toto je Geometry shader
#extension GL_ARB_geometry_shader4 : enable

// Vstup z Vertex shaderu
varying in vec3 v3_vertex_vout[];
varying in vec3 v3_normal_vout[];

// Výstup z Geometry shaderu
varying out vec3 v3_vertex_gout;
varying out vec3 v3_normal_gout;

// Main
void main()
{
	// Pomocná hodnota
	int i;

	// Projdeme každý vrchol každého primitiva
	for(i = 0; i < gl_VerticesIn; i++)
	{
		// Výstupní pozice je rovna vstupní, to stejné platí
		// i pro normály a vertexy - naše body
```

```
            gl_Position = gl_PositionIn[i];
            v3_vertex_gout = v3_vertex_vout[i];
            v3_normal_gout = v3_normal_vout[i];
            // Vytvoř vertex na daném bodě
            EmitVertex();
      }
      // Vytvoř primitivum dle daných parametrů a počtu bodů
      EndPrimitive();
}
```

Fragment (pixel) shader:

```
// Budeme používat Shader Model 4.0 a GLSL verze 1.0
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Vstup z geometry shaderu
varying in vec3 v3_vertex_gout;
varying in vec3 v3_normal_gout;

// Chceme otočit normály? Z programu, pro odvrácene stěny toto
// musíme provést
uniform float flip_normals;

// Main
void main()
{
      // Do prvního bufferu MRT zapíšeme vertexy
      gl_FragData[0] = vec4(v3_vertex_gout, 1.0);
      // Do druhého bufferu MRT zapíšeme správně otočené normály
      gl_FragData[1] = vec4(v3_normal_gout * flip_normals, 1.0);
}
```

6.2 Příloha č. 2

Shader k výpočtu složeného tělesa

Vertex shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Vystup z vertex shaderu
varying out vec2 v2_texCoord_vout;

// Main
void main()
{
      // Texturove koordinaty
      v2_texCoord_vout = gl_MultiTexCoord0.xy;

      // Nastaveni projekce a transformace vertexu
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Geometry shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable
// Geometry shader
#extension GL_ARB_geometry_shader4 : enable

// Vstup z vertex shaderu
varying in vec2 v2_texCoord_vout[];

// Vystup z geometry shaderu
varying out vec2 v2_texCoord_gout;

// Main
void main()
{
	// Pomocna
	int i;

	// Projdi vsechny vertexy kazdeho primitiva
	for(i = 0; i < gl_VerticesIn; i++)
	{
		// Nastav pozici a texturove koordinaty
		gl_Position = gl_PositionIn[i];
		v2_texCoord_gout = v2_texCoord_vout[i];
		// Vytvor bod
		EmitVertex();
	}
	// Uzavri primitivum
	EndPrimitive();
}
```

Fragment shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Vstupni textury
uniform sampler2D Aoperand_front;
uniform sampler2D Aoperand_back;
uniform sampler2D Boperand_front;
uniform sampler2D Boperand_back;
uniform sampler2D Aoperand_front_n;
uniform sampler2D Aoperand_back_n;
uniform sampler2D Boperand_front_n;
uniform sampler2D Boperand_back_n;

// Jakou operaci budeme provadet?
uniform int method;

// Koordinaty textur z geometry shaderu
varying in vec2 v2_texCoord_gout;

// Main
void main()
{
	// Zisk veskerych dat pro dany pixel
```

```
	vec3 Aoperand_front_pos = texture2D(Aoperand_front,
v2_texCoord_gout).xyz;
	vec3 Aoperand_back_pos = texture2D(Aoperand_back,
v2_texCoord_gout).xyz;
	vec3 Boperand_front_pos = texture2D(Boperand_front,
v2_texCoord_gout).xyz;
	vec3 Boperand_back_pos = texture2D(Boperand_back,
v2_texCoord_gout).xyz;
	float Aoperand_front_d = length(Aoperand_front_pos);
	float Aoperand_back_d = length(Aoperand_back_pos);
	float Boperand_front_d = length(Boperand_front_pos);
	float Boperand_back_d = length(Boperand_back_pos);

	// Vystupni hodnoty jsou nulove
	vec3 booleanPos = vec3(0.0, 0.0, 0.0);
	vec3 booleanNorm = vec3(0.0, 0.0, 0.0);

	// Teleso B-A, prepis matematickeho vzorce
	if(method == 0)
	{
		booleanPos = Aoperand_front_pos;
		booleanNorm = texture2D(Aoperand_front_n,
v2_texCoord_gout).xyz;
		if(Boperand_front_d < Aoperand_front_d && Boperand_back_d >
Aoperand_front_d)
		{
			if(Boperand_back_d < Aoperand_back_d)
			{
				booleanPos = Boperand_back_pos;
				booleanNorm = texture2D(Boperand_back_n,
v2_texCoord_gout).xyz;
			}
			else
			{
				booleanPos = vec3(0.0, 0.0, 0.0);
				booleanNorm = vec3(0.0, 0.0, 0.0);
			}
		}
	}
	// Teleso A-B prepis matematickeho vzorce
	else if(method == 1)
	{
		booleanPos = Boperand_front_pos;
		booleanNorm = texture2D(Boperand_front_n,
v2_texCoord_gout).xyz;
		if(Aoperand_front_d < Boperand_front_d && Aoperand_back_d >
Boperand_front_d)
		{
			if(Aoperand_back_d < Boperand_back_d)
			{
				booleanPos = Aoperand_back_pos;
				booleanNorm = texture2D(Aoperand_back_n,
v2_texCoord_gout).xyz;
			}
			else
			{
				booleanPos = vec3(0.0, 0.0, 0.0);
```

```
                        booleanNorm = vec3(0.0, 0.0, 0.0);
                  }
            }
      }
      // Teleso A+B prepis matematickeho vzorce
      else if(method == 2)
      {
            if(Boperand_front_d < Aoperand_front_d && Boperand_front_d >
0.0 || Aoperand_front_d == 0.0)
            {
                  booleanPos = Boperand_front_pos;
                  booleanNorm = texture2D(Boperand_front_n,
v2_texCoord_gout).xyz;
            }
            else
            {
                  booleanPos = Aoperand_front_pos;
                  booleanNorm = texture2D(Aoperand_front_n,
v2_texCoord_gout).xyz;
            }
      }
      // Teleso pruniku A a B, prepis matematickeho vzorce
      else if(method == 3)
      {
            booleanPos = vec3(0.0, 0.0, 0.0);
            booleanNorm = vec3(0.0, 0.0, 0.0);

            if(Boperand_front_d < Aoperand_front_d)
            {
                  if(Aoperand_front_d < Boperand_back_d)
                  {
                        booleanPos = Aoperand_front_pos;
                        booleanNorm = texture2D(Aoperand_front_n,
v2_texCoord_gout).xyz;
                  }
            }
            else
            {
                  if(Boperand_front_d < Aoperand_back_d)
                  {
                        booleanPos = Boperand_front_pos;
                        booleanNorm = texture2D(Boperand_front_n,
v2_texCoord_gout).xyz;
                  }
            }
      }

      // Vystup pozic a normal do prvniho, resp. druheho bufferu MRT
      gl_FragData[0] = vec4(booleanPos, 1.0);
      gl_FragData[1] = vec4(booleanNorm, 1.0);
}
```

6.3 Příloha č. 3

Shader k výpočtu stínování a siluety objektu.
Vertex shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Vystup z vertex shaderu
varying out vec2 v2_texCoord_vout;
varying out vec3 lightPosMV_vout;

// Vstup z programu (svetlo a modelview inverse matice)
uniform vec3 lightPos;
uniform mat4 ModelViewMat;

// Main
void main()
{
	// Texturove koordinaty
	v2_texCoord_vout = gl_MultiTexCoord0.xy;
	// Svetlo prevedene tak, aby bylo relativni ke kamere
	lightPosMV_vout = vec3(ModelViewMat * vec4(lightPos, 1.0));

	// Transformace vertexu pomoci OpenGL modelview projekcni matice
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Geometry shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable
#extension GL_ARB_geometry_shader4 : enable

// Vstup z vertex shaderu
varying in vec2 v2_texCoord_vout[];
varying in vec3 lightPosMV_vout[];

// Vystup z geometry shaderu
varying out vec2 v2_texCoord_gout;
varying out vec3 lightPosMV_gout;

// Main
void main()
{
	// Pomocna
	int i;

	// Projdi vsechny vertexy
	for(i = 0; i < gl_VerticesIn; i++)
	{
		// Zapis pozice, pozici svetla a texturove koordinaty
		gl_Position = gl_PositionIn[i];
		lightPosMV_gout = lightPosMV_vout[i];
		v2_texCoord_gout = v2_texCoord_vout[i];
```

```
            // Vytvor vertex
            EmitVertex();
        }
        // Uzavri primitivum
        EndPrimitive();
}
```

Pixel shader:

```
// GLSL verze 1.0 a Shader model 4.0
#version 120
#extension GL_EXT_gpu_shader4 : enable

// Textury pozic a normal
uniform sampler2D final_position;
uniform sampler2D final_normal;

// Vstup z geometry shader
varying in vec2 v2_texCoord_gout;
varying in vec3 lightPosMV_gout;

// Velikost
const vec2 mapScale = vec2(1.0 / 1024.0, 1.0 / 768.0);

// Main
void main()
{
        // Normaly a pozice na danem pixelu
        vec3 normalTex = texture2D(final_normal, v2_texCoord_gout).xyz;
        vec3 positionTex = texture2D(final_position,
v2_texCoord_gout).xyz;

        // Vypocet normalizovaneho vektoru ze svetla k bodu na kterem
zrovna pocitame
        vec3 lightDir = normalize(lightPosMV_gout - positionTex);
        // Lambertuv zakon kosinu, viz kapitola implementace
        float diffuseLight = max(dot(lightDir, normalize(normalTex)),
0.0);
        // Pridame nejake rozptylene svetlo
        // x, y ci z se muze rovnat nule i na bode kde polopriimka mine
        if(positionTex.x != 0.0 || positionTex.y != 0.0 || positionTex.z
!= 0.0)
        {
                diffuseLight = min(diffuseLight + 0.3, 1.0);
        }
        // Celkove svetlo ztlumime, aby vynikl obrys
        diffuseLight *= 0.5;

        // Suma vertikalni matice sobelova operatoru
        vec3 Gx = vec3(0.0, 0.0, 0.0);
        Gx += texture2D(final_normal, v2_texCoord_gout + vec2(-mapScale.x,
mapScale.y)).xyz;
        Gx += 2.0 * texture2D(final_normal, v2_texCoord_gout + vec2(0.0,
mapScale.y)).xyz;
        Gx += texture2D(final_normal, v2_texCoord_gout + vec2(mapScale.x,
mapScale.y)).xyz;
```

```
      Gx -= texture2D(final_normal, v2_texCoord_gout + vec2(-mapScale.x,
-mapScale.y)).xyz;
      Gx -= 2.0 * texture2D(final_normal, v2_texCoord_gout + vec2(0.0, -
mapScale.y)).xyz;
      Gx -= texture2D(final_normal, v2_texCoord_gout + vec2(mapScale.x,
-mapScale.y)).xyz;

      // Suma horizontalni matice sobelova operatoru
      vec3 Gy = vec3(0.0, 0.0, 0.0);
      Gy += texture2D(final_normal, v2_texCoord_gout + vec2(-mapScale.x,
-mapScale.y)).xyz;
      Gy += 2.0 * texture2D(final_normal, v2_texCoord_gout + vec2(-
mapScale.x, 0.0)).xyz;
      Gy += texture2D(final_normal, v2_texCoord_gout + vec2(-mapScale.x,
mapScale.y)).xyz;
      Gy -= texture2D(final_normal, v2_texCoord_gout + vec2(mapScale.x,
-mapScale.y)).xyz;
      Gy -= 2.0 * texture2D(final_normal, v2_texCoord_gout +
vec2(mapScale.x, 0.0)).xyz;
      Gy -= texture2D(final_normal, v2_texCoord_gout + vec2(mapScale.x,
mapScale.y)).xyz;

      // Prepis sobelova operatoru do intenzity hrany
      float sobel_operator = abs(Gx.x) + abs(Gx.y) + abs(Gx.z) +
abs(Gy.x) + abs(Gy.y) + abs(Gy.z);
      // A odecteni male hodnoty, abychom zabranili chybam v obrysu
      sobel_operator = clamp(sobel_operator - 1.0, 0.0, 1.0);

      // Vysledna barva pixelu je soucet sobelova operatoru a dopadoveho
svetla
      gl_FragColor = vec4(sobel_operator + diffuseLight, sobel_operator
+ diffuseLight, sobel_operator + diffuseLight, 1.0);
}
```