VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

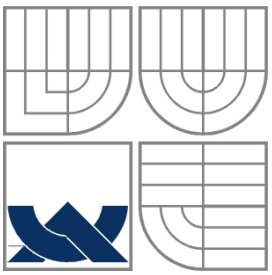
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HRA STYLU DOOM

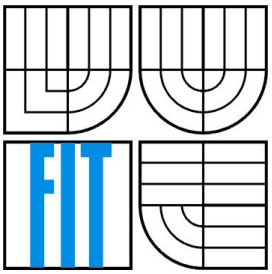
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE ROMAN RAKUS
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HRA STYLU DOOM

DOOM-STYLE GAME

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

ROMAN RAKUS

VEDOUCÍ PRÁCE
SUPERVISOR

ING. JAN PEČIVA

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Rakus Roman**
Obor: Informační technologie
Téma: **Hra stylu Doom (chození v bludišti)**
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte si problematiku rendrování scén se zaměřením na nejnovější techniky a efekty používané v dnešních grafických aplikacích.
2. Vytvořte jednoduchou scénu dle domluvy s vedoucím a zobrazte ji za pomoci knihovny OpenSceneGraph.
3. Navrhněte sadu efektů dle domluvy s vedoucím využívajících i nejnovějších technik dnešních grafických akcelerátorů.
4. Efekty implementujte. Demonstrujte je ve vytvořené scéně.
5. Projekt se pokuste integrovat s projektem Milana Kleibela.
6. Projekt zveřejněte na internetu.

Literatura:

- OpenSceneGraph dokumentace na http://www.openscenegraph.org/

Při obhajobě semestrální části projektu je požadováno:

- Demonstrace 2-3 funčních efektů. Zobrazení jednoduché scény.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Pečiva Jan, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení:	**Roman Rakus**
Id studenta:	89104
Bytem:	Nedbalova 2411, 735 06 Karviná
Narozen:	30. 09. 1983, Havířov
(dále jen "autor")	

a

2. Vysoké učení technické v Brně
Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP:	Hra stylu Doom (chození v bludišti)
Vedoucí/školitel VŠKP:	Pečiva Jan, Ing.
Ústav:	Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP:	.....

VŠKP odevzdal autor nabyvateli v:

tištěné formě	počet exemplářů: 1
elektronické formě	počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	Roman Radus .....
Nabyvatel	Autor

Abstrakt

Využití efektů v počítačových hrách je velmi velké. Kladen je velký důraz na nejpěknější zobrazení za co nejnižší cenu. Využívají se nejnovější grafické adaptéry a techniky, které jsou jimi nabízeny.

V této práci je popsán a implementován postup jak vytvořit některé efekty. Mezi ně patří systémy částic. Dále per-pixel osvětlování, normal bump mapping a fake volumetric lines za využití OpenSceneGraphu, shaderů a dalších technik.

Klíčová slova

Phongovo osvětlování, normal bump mapping, systémy částic, shadery, falešné volumetrické čáry, pohyb těles

Abstract

Usage of effects in computer games is huge. Emphasis is put on the nicest pictures for the lowest price. The newest graphical adapters and technologies are used.

In this thesis is described and implemented procedure how to create some effects such as particle systems, per-pixel lighting, normal bump mapping and fake volumetric lines with usage of OpenSceneGraph, shaders and others.

Keywords

Phong lighting, normal bump mapping, particle systems, shaders, fake volumetric lines, objects motion

Citace

Roman Rakus: Hra stylu Doom, bakalářská práce, Brno, FIT VUT v Brně, 2007

Hra stylu Doom

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pečivy
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Roman Rakus
15. 5. 2007

Poděkování

Poděkování patří vedoucímu bakalářské práce Ing. Janu Pečivovi za veškeré rady a nápady, které mi poskytl.

Obsah

Úvod

Počítačové hry jsou stále populární. Nejeden dospívající muž u nich tráví své dny i večery. Svým zpracováním se z her stávají reality. Hráč se sžívá s hlavním hrdinou a poznává herní svět. Za použití nejmodernějšího technického vybavení můžou herní scény být virtuální realitou, kdy člověk nepozná zda se dívá na film nebo hraje hru.

Informační technologie je jedna z nejrychleji se odvíjejících oblastí moderní vědy. Počítačová grafika, do které spadá i tahle práce, patří mezi přední tahouny v informačních technologiích. Zobrazování grafickou podobou je pro uživatele velmi příjemné. Počítačová grafika se hlavně používá ve hrách. Vývoj je tak rychlý, že to co před rokem bylo jako žhavá novinka, je další rok považováno za standard a mnohdy až za zastaralé.

Počítačová grafika se ve hrách využívá nejen pro základní zobrazování scény, ale i pro nejrůznější efekty. Také díky hrám, je tohle odvětví velmi populární a těší se tak obrovského rozvoje a vývoje.

Grafické efekty jsou také využívány v zábavním průmyslu. Ve filmech se využívají již na tak velké úrovni, že divák málokdy pozná, zda se jedná o realitu nebo počítačem upravenou scénu. Velmi oblíbené jsou taky kreslené filmy, v dnešní době lépe řečeno počítačem animované filmy.

V této práci naleznete postupy pro implementaci některých grafických efektů užívaných ve hrách. Většina efektů zde popsaných vychází ze světelných modelů.

1 Používané grafické efekty

1.1 Grafické knihovny

Kvalitní zobrazování scény není jednoduché. Z toho důvodu je vhodné použít knihovny pro tvorbu počítačové grafiky. Nejpopulárnější a nejvíc používané jsou OpenGL a DirectX.

Tyto knihovny, nebo-li aplikační programové rozhraní (API), jsou univerzální nástroj pro všechny grafické adaptéry. Vývojář nemusí řešit, na kterém adaptéru se scéna bude zobrazovat a může se zabývat pouze scénou samotnou. U OpenGL je navíc situace taková, že je multiplatformní a s otevřenými zdroji (Open source). DirectX je úzce spjatý s platformou Windows, ale lze ho vidět i na některých jiných platformách, např. X-BOX. Za zmínku stojí ještě jeden rozdíl mezi OpenGL a DirectX a to v dokumentaci. OpenGL je mnohem lépe zdokumentovaná, a proto není problém s ní začít i pro úplné začátečníky.

1.2 Standardní OpenGL osvětlování

Je důležité podotknout, že osvětlování OpenGL je empirické, zjednodušené a vhodné pro real-time zobrazování.

Výsledná barva se v OpenGL spočítá ze dvou důležitých částí. Z parametrů materiálu osvětlovaného objektu a parametrů světla dopadajícího na objekt. Dále se ještě připočítávají parametry světelného modelu celé scény. Všechny parametry a jejich implicitní hodnoty jsou k nalezení v [1].

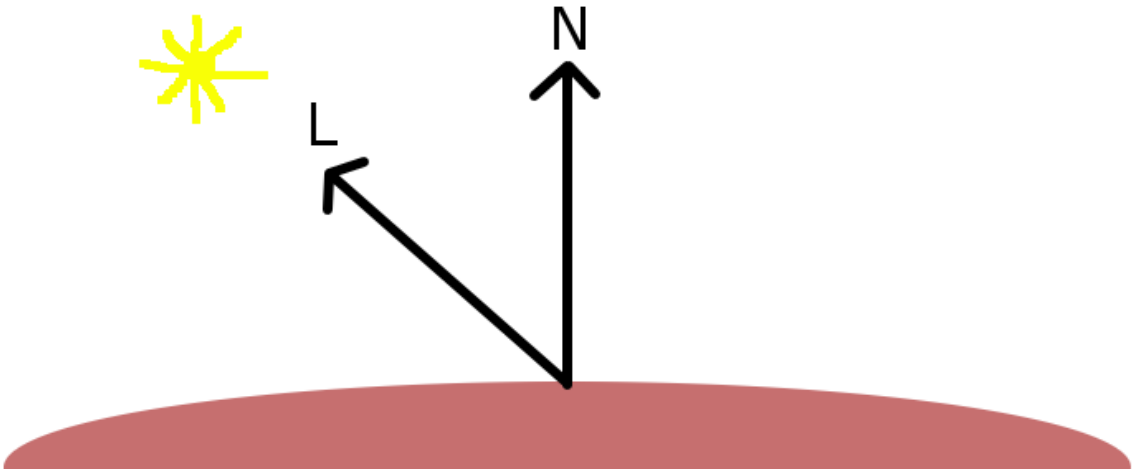
1.2.1 Lambertův osvětlovací model

Empirický osvětlovací model, který počítá pouze s difúzní složkou osvětlení. Můžeme si ji označit `Id`. Difúzní složku světla označme `Dl` a difúzní složku materiálu `Dm`. Potom platí rovnice 1

`LambertTerm` je klíčovým faktorem pro výpočet osvětlení. Vypočítá se z normálového vektoru `N` počítaného bodu a vektoru světla `L` toho samého bodu (viz Obrázek 1). Samotný koeficient pak je skalárním součinem těchto vektorů (viz rovnice 2). Kde je brána hodnota vyšší nebo rovna 0.

$$Id = Dl \bullet Dm \bullet LambertTerm \tag{1}$$

$$LambertTerm = N \bullet L \tag{2}$$



Obrázek 1: Ukázka vektorů vycházejících z bodu pro výpočet Lambertova osvětlovacího modelu

Jedná se o jednoduchý výpočet, ale napomáhá v generování stínování. Pokud vektory `N` a `L` svírají nulový úhel, to znamená, že počítaný bod je přímo před světlem, bude Lambertův koeficient maximální, čili 1. Ve všech ostatních případech bude hodnota mezi 0 a 1, což bude generovat stínování (self-shadowing).

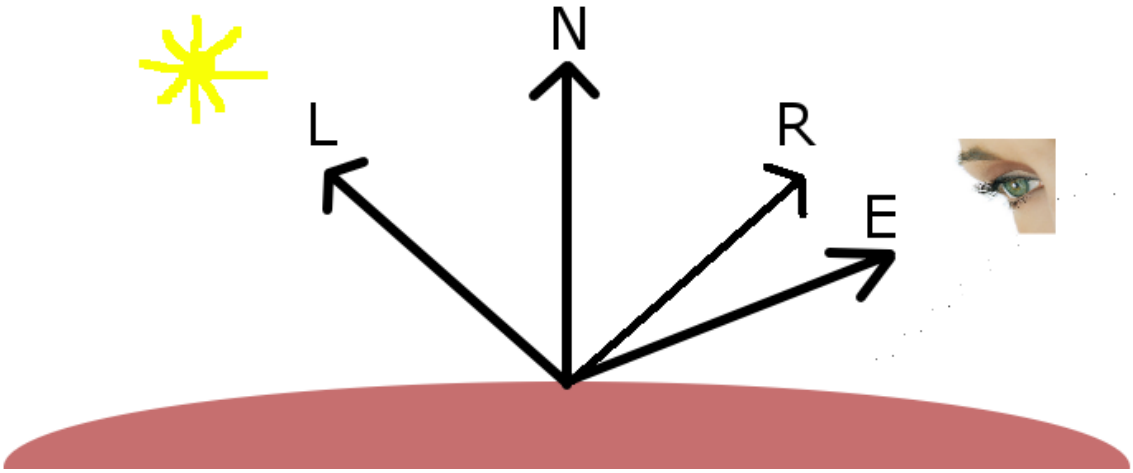
1.2.2 Phongův osvětlovací model

Empirický osvětlovací model, který k difúzní složce přidává odrazovou složku, tak zvanou reflexi `Is`. Reflexi materiálu označíme `Sm`, reflexi světla `Sl`.

Phongův osvětlovací model dále potřebuje pro svůj výpočet lesklost materiálu `f`, pohledový vektor `E` a vektor odraženého světla vzhledem k normálovému vektoru bodu (viz Obrázek 2). Odražený vektor se vypočítá podle rovnice 3 a výsledný výpočet reflexe podle rovnice 4.

$$R = 2 \bullet (N \bullet L) \bullet N - L \tag{3}$$

$$Is = Sm \bullet Sl \bullet (R \bullet E)^{f} \tag{4}$$



Obrázek 2: Ukázka vektorů pro výpočet Phongova osvětlovacího modelu

1.2.3 Ambientní složka světla

Poslední složka OpenGL osvětlování je tzv. ambientní složka, nebo-li rozptýlené světlo. Pro jeho výpočet se použije ambientí složka materiálu `Am` a ambientní složka světla `Al`. Vlastní výpočet ambientní složky `Ia` je jednoduchý (viz rovnice 5).

$$Ia = Am \bullet Al \tag{5}$$

1.2.4 Směrové světlo

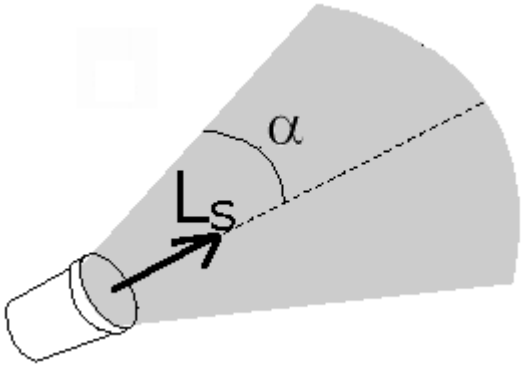
Anglicky *Directional light* je považováno za světlo v nekonečné vzdálenosti od všech objektů scény. Paprsky takového světla dopadají na objekty s konstantní intenzitou a výsledná složka světla osvětlení se nijak nemění. Příkladem takového světla je například slunce.

1.2.5 Reflektor

Anglicky *Spot light* je poblíž nebo přímo ve scéně mezi objekty. Z toho důvodu takové světlo zasahuje mnohem víc do výsledného osvětlení. Jedná se o jedno ze dvou pozičních světel, anglicky *Positional light*. Světelné paprsky, které reflektor vysílá jsou omezeny kuželem. Tento kužel je dán dvěma parametry. Směrovým vektorem L_S a úhlem α mezi osou kuželu a hranou kuželu (viz obrázek 3). Takovým světlem je například svítící baterka.

Výsledná složka světla je ovlivněna parametry kuželu. Navíc je také parametr udávající intenzitu světla, označíme si ho `i`. Tento parametr udává jak moc je světlo koncentrované u osy kuželu. Světlo osvítí bod pokud leží v kuželu světla (viz rovnice 6). Tedy pokud skalární součin vektoru světla reflektoru a vektoru od světla k osvětlovanému bodu je větší než cosinus úhlu α, nebude bod tímto světlem osvětlen.

$$-L \bullet L_S < \cos(\alpha) \tag{6}$$



Obrázek 3: Spot light - reflektor s úhlem α a směrovým vektorem

Výsledné osvětlení bodu je změněno koeficientem `SpotEffect` podle rovnice 7.

$$SpotEffect = \left(-L \bullet L_S\right)^i \tag{7}$$

1.2.6 Bodové světlo

Anglicky *Point light*. Druhé z pozičních světel v OpenGL. Světlo vychází z jednoho bodu do všech směrů a je utlumováno podle nastavených parametrů. Může se jednat například o svítící žárovku.

Lze nastavit tři hodnoty útlumu, anglicky *attenuation*. Konstantní `c1`, ten je implicitně nastaven na hodnotu 1, lineární `c2` a kvadratický `c3` útlum. Tyto útlumy jsou závislé na vzdálenosti (viz rovnice 8) kde `d` je vzdálenost osvětlovaného bodu od zdroje světla. S implicitními hodnotami není útlum použit. Výpočet útlumu je nejnáročnější ze všech výpočtů standardního osvětlování v OpenGL.

$$attenuatio\,n = \frac{1}{c_1 + c_2 d + c_3 d^2} \tag{8}$$

Bodové světlo se může kombinovat s reflektorovým světlem. V takovém případě se pro výpočet útlumu v čitateli objeví koeficient `SpotEffect` (viz rovnice 9).

$$attenuatio\,n = \frac{SpotEffect}{c_1 + c_2 d + c_3 d^2} \tag{9}$$

1.2.7 Výsledná barva bodu

Všechny tři modely osvětlování jsou použity pro výslednou barvu bodu. `Id`, `Is` i `Ia` jsou čtyř prvkové vektory obsahující červenou, zelenou, modrou a alfa složku. Tyto složky odpovídají RGBA barevnému modelu a leží v rozsahu mezi 0,0 a 1,0 včetně.

Podle druhu světla se může dále ovlivnit výsledná barva bodu:

- Směrové světlo již dále barvu nemění
- Poziční světlo ovlivní výslednou barvu podle rovnice 10

Objekt může být potažen texturou. V tom případě připadá jednomu bodu také hodnota RGBA z dané textury a ovlivňuje výslednou barvu bodu. Pokud označíme barvu textury I_T, pak pro výčet barvy bodu `I` za použití bodového světla můžeme použít rovnici 10.

$$I = I_T \bullet attenuation \bullet (Id + Is + Ia) \tag{10}$$

1.3 Standardní OpenGL stínování

1.3.1 Co je stínování?

Anglicky *shading*. Vyhodnocování osvětlovacího modelu v každém bodě, který je vykreslován na obrazovce, je zdlouhavé, a proto bylo vyvinuto několik metod, které umožňují provést podrobný

výpočet osvětlovacího modelu pouze pro několik bodů na povrchu tělesa a odvodit z nich barevné odstíny ostatních zobrazovaných bodů.

Tyto metody shrnujeme pod společný název stínování. Pomocí stínování lze odlišit případné křivosti a zaoblení ploch, a docílit tak přirozeného vzhledu prostorových objektů přesto, že řada výpočtů týkajících se zpracování světla byla zjednodušena či vynechána. Některé druhy stínování dokonce umožňují opticky vyhladit povrchy, které jsou modelovány sítí rovinných plošek, takže přestanou být znatelné drobné hraniční zlomy. [2]

Ačkoliv je stínování spjato se zpracováním světla, nezabývá se nalezením vržených stínů, anglicky *shadows*. Ty lze zpracovat speciálními metodami, které jsou popsané v [2].

1.3.2 Konstantní stínování

Anglicky *flat shading*. Nejjednodušší a zároveň nejrychlejší metoda stínování v OpenGL. Používá se pro zobrazování rovinných ploch. Předpokládá, že každá plocha má jen jedinou normálu. Není-li normála implicitně obsažena v datech prostorového modelu, lze ji u konvexních rovinných plošek určit jako výsledek, který je správně zorientován tak, aby ukazoval do vnějšího poloprostoru. Podle normály je vypočítána jedna barva, která je při rasterizaci plochy přiřazena všem jejím pixelům. Konstantní stínování je používáno tam, kde je třeba docílit vysoké rychlosti kresby.

Pro kresby mnohostěnů je tento způsob stínování postačující a úspěšně znázorňuje umístění a natočení těles v prostoru. U obecnějších těles je konstantní odstín plošek negativním jevem, protože místo zkvalitnění obrázku zdůrazňuje, že oblý povrch je ve skutečnosti jen aproximován skupinou plošek (obrázek 4 vlevo) [2]

1.3.3 Gouraudovo stínování

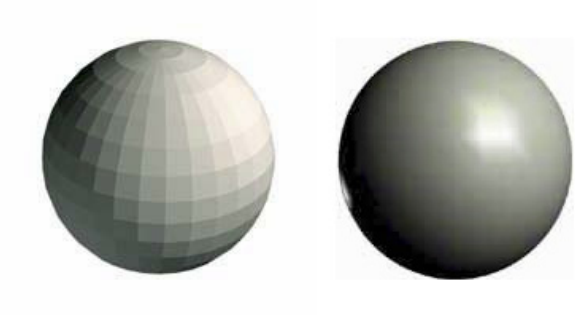
Pro obecná tělesa byly vyvíjeny metody spojitého barevného stínování. Jsou založeny na určování odstínu barvy bodu v ploše (obecně v prostorovém polygonu) ze znalosti hodnot ve vrcholech této plochy. Známe-li v každém vrcholu jeho barvu či normálu, dokážeme tyto hodnoty vypočítat bilineární interpolací pro každý vnitřní bod plochy. Při stínování pomocí bilineární interpolace vybarvujeme plochu po řádcích. Lineární interpolací mezi vrcholy určíme potřebné hodnoty na hranách, další lineární interpolací pak hodnoty uvnitř vybarvovaného řádku.

Gouraudova metoda stínování byla navržena H. Gouraudem a je vhodná pro stínování těles, jejichž povrch je aproximován množinou rovinných plošek. Pro činnost algoritmu je důležitá znalost barev všech vrcholů zpracovávané plochy. Barvu vrcholu určíme vyhodnocením osvětlovacího modelu. Poté jsou vypočítány barevné odstíny vnitřních bodů dané plošky bilineární interpolací. Proto se také Gouraudovo stínování nazývá interpolací barvy.

Barvu v podobě trojsložkového vektoru RGB můžeme interpolovat po jednotlivých složkách. Každou složku v původním rozsahu <0, 1> lze převést do celočíselného intervalu 0-255

a pro interpolaci použít celočíselnou aritmetiku. Celočíselná bilineární interpolace se snadno realizuje technickými prostředky. V současné době je proto Gouraudovo stínování standardní metodou používanou v grafických akcelerátorech.

Gouraudova metoda zajišťuje plynulé stínování křivých povrchů tak, že aproximace povrchů ploškami není zřetelná. Přesto ani tyto obrázky ještě neposkytují zcela věrný obraz reálných objektů – interpolace samotného odstínu barvy totiž nemůže způsobit místní zvýšení jasu na rovinné plošce, která nahrazuje zakřivenou plochu orientovanou kolmo na dopadající světelný paprsek. Tato metoda zahlazuje barevné rozdíly u místních nerovností povrchu. (obrázek 4 vpravo) [2]



Obrázek 4: Srovnání konstantního stínování (vlevo) a Gouraudova stínování (vpravo)

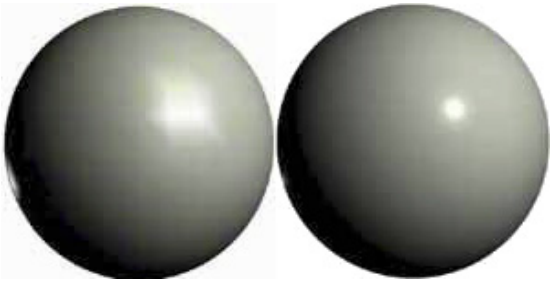
1.4 Phongovo stínování

Také tato metoda je určena k plynulému stínování těles, jejichž povrch je tvořen množinou rovinných ploch. Jméno metody je spojeno s prací Bui-Tuong Phonga. Jeho jméno se tak objevuje v počítačové literatuře ve dvou souvislostech – jako jedna z variant empirického osvětlovacího modelu (viz kapitola 1.2.2) a jako metoda stínování povrchů těles.

Metoda vychází ze znalosti normálových vektorů ve vrcholech stínované plochy. Z nich však nejsou pouze vypočítány barevné odstíny ve vrcholech, jako tomu bylo u Gouraudova stínování, nýbrž jsou použity k určení normálových vektorů ve vnitřních bodech plochy bilineární interpolací. Phongovo stínování je tedy založeno na interpolaci normálových vektorů. Normály jsou interpolovány současně při pasterizaci plochy a z nich je vyhodnocením, ve kterém bereme v úvahu všechny složky světla osvětlovacího modelu, určen odstín barvy každého pixelu.

Časové nároky algoritmu ve srovnání s Gouraudovým stínováním vzrostou, neboť osvětlovací model je vyhodnocován pro každý pixel – z toho důvodu se Phongovo stínování často nazývá osvětlování po pixelu, nebo anglicky *per pixel lighting*.

Gouraudovo stínování (obrázek 5 vlevo) vyniká rychlostí výpočtu, ale nedociluje tak kvalitního vzhledu obrázku jako stínování Phongovo (obrázek 5 vpravo). Zcela nevhodné je jeho použití v dynamických scénách, kdy dochází v průběhu natáčení plošek k viditelným jasovým změnám na povrchu. Pokud bychom například postupně otáčeli válec kolem jeho osy, spatřili bychom na plášti výrazný světlý pruh vždy, když by se hrana pláště natočila kolmo na směr světla. [2]



Obrázek 5: Srovnání Gouraudova stínování (vlevo) a per-pixel stínování (vpravo)

1.5 Bump mapping

Pokud chceme dosáhnout ještě lepších obrazových výsledků než Phongovo stínování, můžeme se zabývat Bump mappingem. Zatímco Phongova metoda stínování vypočítává normálu každého pixelu z normál ve vrcholech, metoda bump mappingu pro určení normály používá externí data, nebo-li bump mapu. Tato bump mapa má stejné rozměry jako textura. Pro zobrazení objektu se použije jak textura, tak bump mapa. Bump mapping je vhodný pro vytváření povrchů lehce nerovných těles, jako jsou třeba golfový míček, pomeranč, cihlová zeď apod. [4]

1.5.1 Prvotní Bump mapping

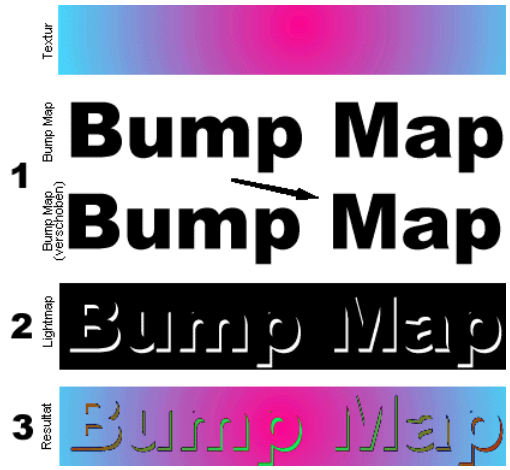
Jim Blinn v roce 1978 přišel s nápadem bump mappingu. Pro vytváření nerovností použil černobílou výškovou mapu. Pro výpočet konečné normály zkombinoval derivace souřadnic vrcholu (či jiných parametrů) s výškovou mapou. Hodnota derivace pak odpovídala strmosti simulovaného povrchu.

Tato technika se v dnešní době moc nepoužívá. Větší roli hrají další techniky. [4]

1.5.2 Emboss Bump mapping

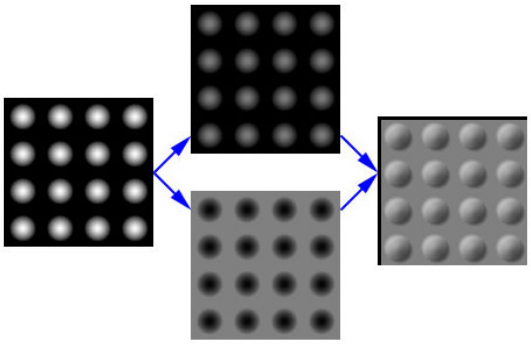
Jedná se o nejméně efektivní metodu Bump mappingu. Základem je také Bump mapa, ale nepracuje s normálami, jde tedy spíš o pseudo Bump mapping. Pro implementaci Emboss Bump mappingu se používají dvě různé metody.

Starší metoda implementace Emboss Bump mappingu používá výškovou texturu posunutou o určitou vzdálenost ve směru světla. Pak se původní a posunutá textura od sebe odečtou a tak vznikla bump mapa. Ta byla aplikována na původní texturu, čímž bylo docíleno prostorového dojmu. Tato metoda je prováděna ve třech krocích (viz obrázek 6). Požadavky na hardware jsou minimální, což je jediná výhoda, více je nevýhod. Při vytváření povrchu jsme vázáni směr světla a dané posunutí. Kdybychom chtěli vytvořit složitější vzor, museli bychom použít více map a průchodů. Z obrázku je také patrné, že tato metoda se hodí pouze na rovné plochy.



Obrázek 6: Ukázka aplikace první metody Emboss Bump mappingu

Současná metoda Emboss Bump mappingu se od starší metody odlišuje jen minimálně a nese s sebou většinu jejich problémů. Z výškové mapy se vytvoří dvě nové textury. První je s poloviční intenzitou a druhá je invertovaná také s poloviční intenzitou. Tyto nové textury se ve směru osvětlení navzájem posunou a nakonec zkombinují s původní texturou (viz obrázek 7). Výsledná výšková mapa může být v případě malého posunutí značně ovlivněna filtracemi což vede až ke snížení kvality výsledného obrazu. V případě pohyblivých scén je Emboss Bump mapping zcela nevhodný. [4]



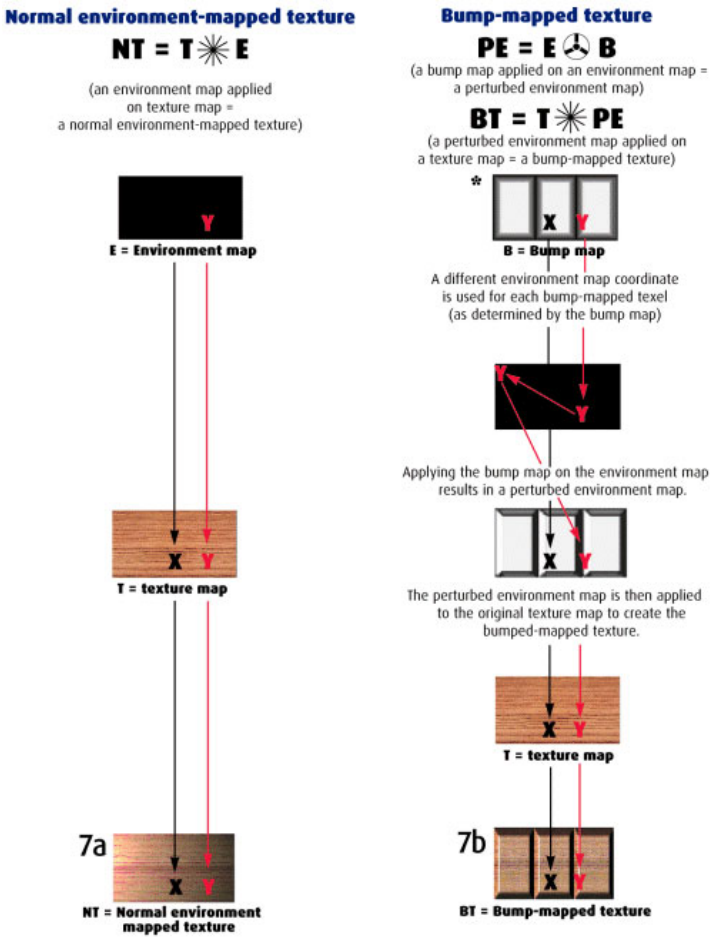
Obrázek 7: Vytvoření bump mapy pro druhou metodu Emboss Bump mappingu

1.5.3 Environment Bump Mapping

Autorem Environment-mapped Bump Mappingu (EMBM) je společnost Bitboys. Jako první tuto metodu s hardwarovou podporou přinesla společnost Matrox, která ho uvedla ve své řadě Millennium G400. Jedná se o velice efektivní metodu, která je ovšem náročnější na zpracování v hardwaru počítače.

Environment Bump Mapping potřebuje celkem tři textury – klasickou texturu, bump mapu a mapu prostředí, anglicky environment map. Bump mapa je trochu odlišná od všech předchozích. Odpovídá v podstatě nanášené textuře a každému jejímu texelu přiřazuje danou souřadnici z environment mapy. Po kombinaci bump mapy a environment mapy s původní texturou pak vznikne mapa osvětlení pro daný objekt nebo scénu.

Na obrázku 8 vlevo je vidět jak se zachová textura s namapováním environment mapy. Mapa přidá textuře osvětlení a stíny a to tím způsobem, že zdrojem je environment mapa sama. Do osvětlení a stínování není započteno žádné světlo scény. Na obrázku 8 vpravo je aplikace bump mapy. Bump mapa přiřadí každému texelu danou souřadnici z environment mapy, tedy i intenzitu osvětlení, a vzniká dojem prostoru.



Obrázek 8: Vlevo – aplikování environment mapy, vpravo – aplikování bump mapy

Environment Bump mapping má dvě základní použití. Za prvé vytváření iluze členitého povrchu. Umí také ale dynamicky zpracovávat pohybující se povrch. Takovým povrchem může být například vodní hladina. Dá se také použít k vytváření reflexních ploch. V takovém případě environment mapa nenese informace o osvětlení scény, ale texturu, která se bude odrážet.

Výhody Environment-mapped Bump Mappingu:

- Pomocí jediné bump mapy lze simulovat i náročnější model osvětlení, tedy včetně několika světelných zdrojů apod. Je to zaručeno tím, že všechny informace o osvětlení jsou uloženy v environment mapě a odkazuje se na ně v libovolně upravitelné bump mapě.
- Protože bump mapa odpovídá přesně původní textuře, u pohyblivých scén není třeba, aby se měnila v každém snímku. Jedná se totiž o per-pixel zpracování, kde se nám jak textura, tak bump mapa pohybují s objektem, zatímco environment mapa je zcela

nezávislá. Na rozdíl od Emboss Bump mapping toto velmi šetří výkon grafického adaptéru.

- Díky možnosti dynamické bump mapy lze simulovat vlnící se vodní hladinu, která se ovšem v GPU zpracovává jako rovná plocha. Aplikací bump mapy, která se cyklicky mění, se vytváří dojem pohyblivé plochy, a to včetně odrazů. Environment mapa v tomto případě obsahuje i texturu, která se od hladiny odráží, např. stromy, hory apod.
- EMBM může být závislý i nezávislý na úhlu pohledu a může na něj být použito globální i bodové osvětlení. To se dále zpracovává klasicky přes normálové vektory použitím Gouraudovo stínováním.

Je třeba zmínit i hlavní nevýhodu, kterou je absence práce s normálovými vektory. Environment bump mapping s nimi nepracuje a je plně vázán na informace v environment mapě. Pokud se ve scéně objeví více světel různě rozmístěných, dochází k nežádoucím efektům v obraze a to hlavně pokud se jedná o reflexivitu. [4]

1.5.4 Normal Bump Mapping

Normálové mapování, anglicky *normal mapping*, se v různých dokumentech označuje různými názvy. Všechny výrazy jako je Dot product Bump Mapping, DOT3 Bump Mapping, Per-pixel shading, Per-pixel lightning označují svým způsobem stejný jev. Poslední dva ovšem v širším slova smyslu.

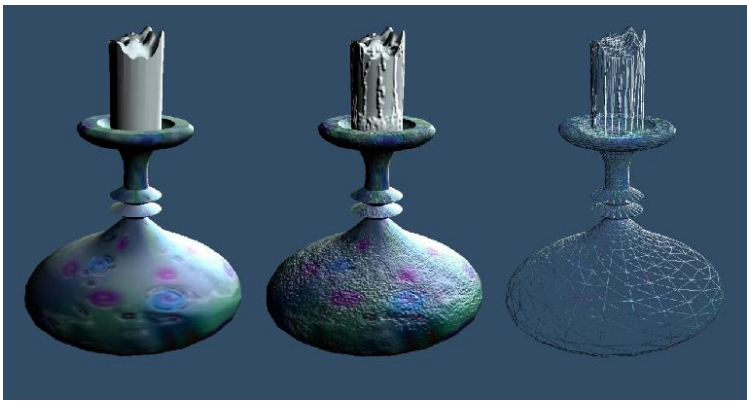
Normálové mapování je dnes nejrozšířenějším typem Bump Mappingu, protože jako jedno z mála využívá k osvětlení přímo normály – proto název Normal mapping. Další výhodou je, že pracuje čistě na per-pixel úrovni. Zatímco předchozí typy vycházely při osvětlení, které nebylo dáno bump mapu, z normál, které byly při Gouraudově stínování interpolovány, normálové mapování přímo určuje normály pro každý pixel uvnitř polygonu. Tím je umožněno vytvářet obrovskou škálu efektů.

Normal Bump mapping se dá použít ve dvou variantách: object-space a tangent-space normal mapping. Odlišují se od sebe v tom, v jakém souřadném systému jsou normály vnímány a uloženy. Tangent-space normal mapping je složitější, ale také přináší širší využití.

V kapitole 1.2 je popsáno jak pracuje osvětlování. Důležitým prvkem pro výpočet osvětlení je normála. Pomocí Phogova stínování se tato normála mohla vypočítat. V Normal Bump mappingu ale používáme normal mapu, která nese informace o normále v každém texelu. Normal mapa je obyčejná textura, která ovšem nenese informace o barvách, ale o normálách. Zatímco v základním Bump Mappingu je použita Bump mapa v osmibitové barevné hloubce (odstíny šedi), normal mapa je v 24 bitové hloubce, tudíž nese 65 536krát více informací. Každý barevný kanál textury koresponduje s prostorovou osou. Červené barvě odpovídá osa X, zelené osa Y a modré barvě osa Z. Tyto osy jsou relativní vzhledem ke konstantnímu souřadnému systému při použití object-space normal mappingu

a nebo jemně se měnící v případě tangent-space normal mappingu. Jinými slovy v případě tangent-space mappingu se normála vypočítá při každém natočení objektu. Je to umožněno tím, že normála je uložena v tečném prostoru, anglicky *tangent-space*. Pokud se otočí objekt, tečny jsou stále stejné. To umožňuje použít normal mapping i v dynamických scénách. [4, 5]

Při vytváření normal mapy se vychází z modelu složeného z mnohonásobně více polygonů než má zobrazovaný model. Zobrazovat složitější model by bylo velmi náročné na grafický adaptér, proto se zobrazuje jednodušší s aplikovanou normal mapou. Na obrázku 9 je vidět jakého efektu lze normal bump mappingem docílit.



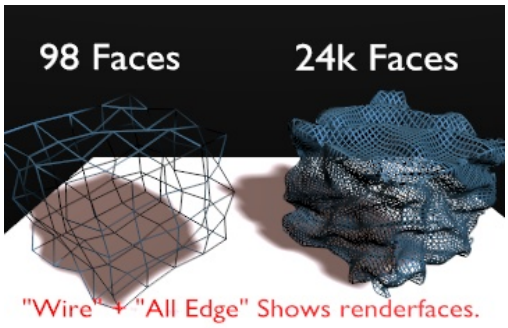
Obrázek 9: Zleva – objekt standardně stínovaný, stejný objekt s použitím normal mappingu, drátěný model objektu

1.5.5 Displacement Mapping

Displacement Mappingu je mnohem složitější technika než všechny předešlé. Bump Mapping zakřivení povrchu pouze simuluje, což má nevýhodu v tom, že silueta objektu a stíny jím vrhané jsou stále jen obrazem původního objektu. Tento problém řeší právě Displacement Mapping, který pomocí textury dokáže členitost povrchu přímo vytvořit.

K použití Displacement mappingu je potřeba výšková mapa. Tato mapa je dále zpracována a ve vertex shaderu (shadery viz kapitola 1.6), který upravuje geometrickou podrobnost daného objektu. Velká výhoda tkví v tom, že se povrch modeluje pomocí jednoduché textury v osmi nebo šestnáctibitové barevné hloubce a nemusí se nahrávat souřadnice všech vrcholů. Dokonce lze celý objekt vymodelovat pomocí Displacement Mappingu.

Displacement mapping zatím není moc používán. Jakých výsledků s ním lze dosáhnout je vidět na obrázku 10. [4]



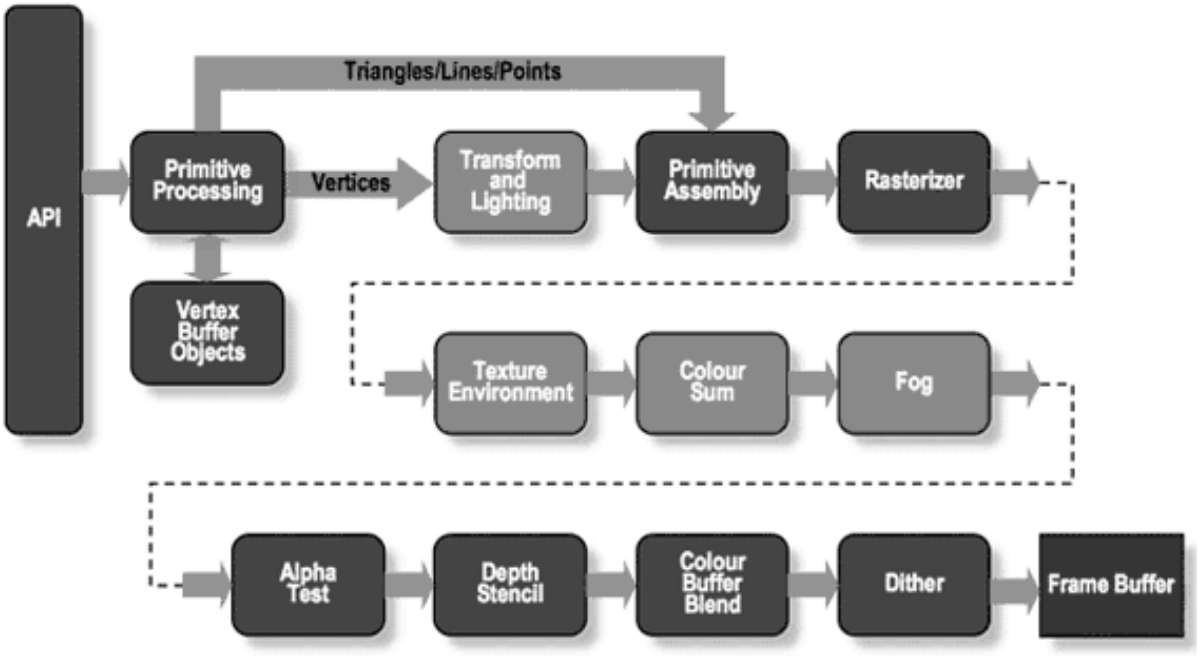
Obrázek 10: Vlevo základní objekt, vpravo objekt za použití displacement mappingu

1.6 Shadery

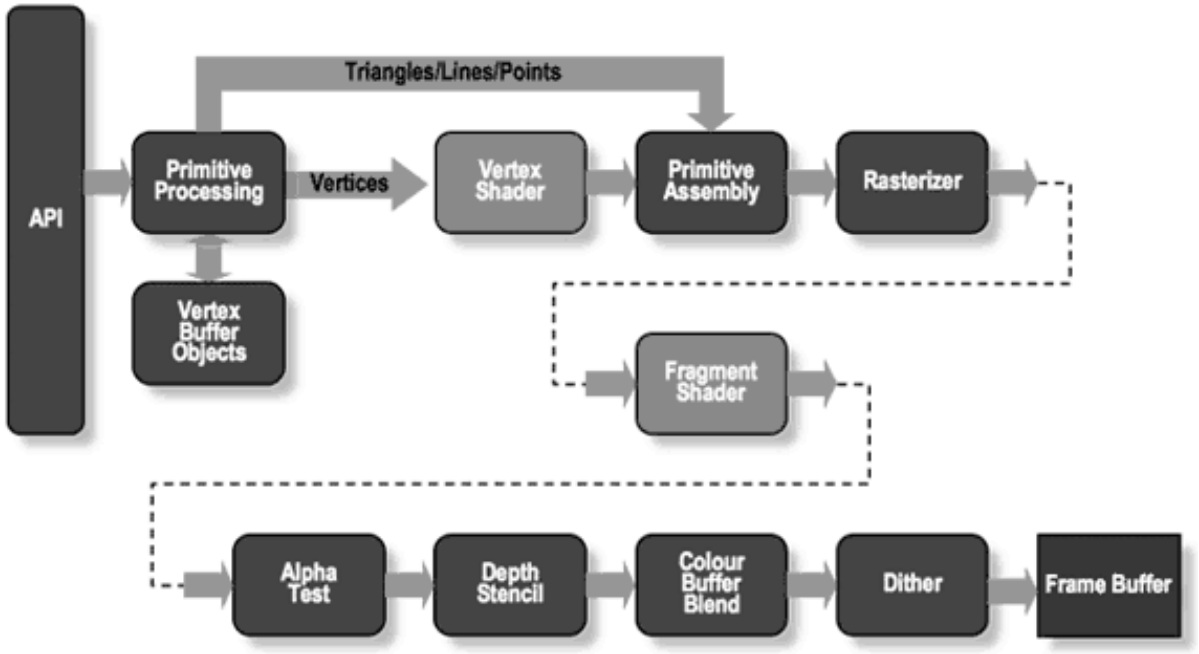
Shadery, anglicky shaders, jsou programy, které se zpracovávají přímo na grafickém adaptéru. Tímto způsobem při renderování scény ulehčují práci procesoru. Takové programy se nejčastěji píšou ve vyšších programovacích jazycích. Nejčastěji používané jsou:

- Cg – jazyk vyvinutý firmou NVIDIA. Jedná se o vyšší jazyk nezávislý na použité API (OpenGL nebo DirectX) a na použitém hardwaru. Jazyk Cg je tedy z tohohle pohledu univerzální. Zkratka Cg znamená *C for graphics* a syntaxe je tedy velmi podobná jazyku C. Zvláštním znakem jazyka Cg je použití tak zvaných konektorů. Jsou to speciální datové struktury, které jsou určené k propojení různých stupňů procesu. Používají se k definování vstupů z aplikace do stádia zpracování vertexů a atributů pro zpracování fragmentů.
- OpenGL shading language – také známý jako GLSL nebo glslang. Jedná se o vyšší jazyk určený pro použití s OpenGL. V OpenGL shading language se píšou dva programy zvlášť. Jeden je vertex shader a druhý fragment shader. Syntakticky je GLSL velmi podobný jazyku C.
- DirectX High-Level Shader Language – Zkráceně označovaný HLSL je pravděpodobně nejúspěšnější jazyk pro psaní shaderů. Hlavní roli zde hraje tlak ze strany Microsoftu kombinovaný s faktem, že se jedná o první jazyk podobný jazyku C používaný pro real-time renderování. Hlavním konkurentem jazyka HLSL je jazyk GLSL.

Shadery verze 3.0 nám umožňují změnit pevné zpracování vertexů ve vertex shaderu a fragmentů (pixelů) ve fragment shaderu. Existují i shadery verze 4.0, které geometry shader, ale podpora existuje zatím jen u DirectX 10 a nejnovějších grafických karet. Na obrázku 11 je pevné zpracování vertexů a pixelů. Pokud ovšem použijeme shader, můžeme si toto zpracování upravit jak budeme chtít. Na obrázku 12 je vidět schéma, jak vypadá programovatelné zpracování.



Obrázek 11: Zjednodušené schéma pevného zřetězení v OpenGL



Obrázek 12: Zjednodušené schéma programovatelného zřetězení v OpenGL

Vertex shadery nám umožňují pracovat s vertexy. Tyto vertexy do našeho shaderu jsou posílány po jednom a právě a jen s tímto jedním vertexem můžeme pracovat. Nemůžeme tedy přidat další vertex nebo ho ubrat. Jedná se tedy o jeden program, který je aplikovaný na každý vertex objektu. Vertex shaderem můžeme provádět matematické operace na datech vertexu. Každý vertex může být definován různými parametry. Například každý vertex je definován svojí pozicí ve 3D prostředí pomocí xyz souřadnic. Může také být definován barvou, texturou nebo světelnou charakteristikou. Vertex shaderem nemůže měnit typy dat, ale můžeme měnit hodnoty těchto dat. Například můžeme měnit pozici vertexu a tím simulovat vodní hladinu.

Pomocí fragment shaderů, někdy označovány pixel shadery, můžeme pracovat s jednotlivými fragmenty, neboli pixely. V tomto shaderu se řeší texturování, výsledná barva pixelu a mlha (viz obrázky 11 a 12). Jelikož se může jednat o velký počet pixelů, může se jednat i o celou obrazovku, a velký počet operací, které jsou zpracovávány, bývají tyto programy co nejvíce optimalizovány. Například při rozlišení 1024 x 768 pixelů, je tento shader aplikován více než sedm set tisíckrát během jednoho snímku. V pixel shaderu můžeme měnit pouze výslednou barvu pixelu, ale jelikož máme přístup k texturám, nabízejí se nám nevídané možnosti. Můžeme aplikovat Phongovo stínování (kapitola 1.4), bump mapping (kapitola 1.5), stíny, efekty exploze a mnohé další.

Geometry shadery mohou vytvářet nové primitivy, jako třeba vertexy, čáry nebo trojúhelníky, z existujících primitiv. Tyto shadery bývají zařazeny za Vertex shaderem a jejich vstupem bývá informace o sousedním primitivu nebo celé primitivum. Například pokud pracujeme s trojúhelníkem jsou tři vertexy zpracovány geometry shaderem. Ten pak může dále poslat nula nebo více primitiv, které jsou rasterizovány a fragmenty pak zpracovány ve fragment shaderu. Geometry shadery jsou ale podporovány pouze v DirectX 10 a v nových grafických adaptérech. [6]

1.7 Částicové systémy

Silnou modelovací a zobrazovací technikou jsou systémy částic, anglicky *particle systems*, které se používají zejména k modelování objektů, jejichž tvar je natolik členitý, nebo se mění takovým způsobem, že ho není možno reprezentovat jako povrch. Takovými objekty jsou hejna ptáků či ryb, padající sníh, déšť, oheň, mlha, dým, tráva, les, atp.

Systém částic je reprezentován jako soubor velmi malých prvků – částic, jejichž vlastnosti se mění v čase. Mezi tyto vlastnosti patří především poloha, barva, rychlost a směr pohybu, zrychlení, atp. Částice mohou v jistém okamžiku a místě vznikat, po určité době života mizí, mohou emitovat další částice, srážet se mezi sebou a s okolními objekty atp. Částice může být reprezentována prakticky jako cokoliv. Nejčastěji je znázorněna bodem, může být však zobrazena i jako kapka, sněhová vločka, koule či jiný geometrický útvar, vše záleží na jevu, který systém částic modeluje.

Klíčovou roli při tomto modelování hrají náhodná čísla. Vlastnosti, které částice mají, jsou vesměs charakterizovány jednoduchým vztahem (rovnice 11), kde `s` je střední hodnota hustoty pravděpodobnosti náhodných čísel, `v` je jejich rozptyl, `rand` je náhodné číslo a `x` reprezentuje například dobu života částice, její barvu, atp. Parametry obyčejně zadává tvůrce systému částic, stejně jako určuje objekt či plochu, která částice vysílá.

$$x = s + rand \bullet v \tag{11}$$

Tvorba jednoho obrázku animované sekvence založené na systému částic se skládá z posloupnosti následujících kroků: [2]

1. Na základě parametrů každé existující částice se aktualizuje její poloha a ostatní atributy

2. V celém systému se vytvoří nové částice. Ty vznikají buď v nějaké konkrétní oblasti, nebo mohou vznikat jako potomci jiných částic. Oblast, ve které částice vznikají, může být libovolná – mrak, ze kterého prší, děravá hadice, ze které stříká voda.
3. Každé nové částici se přiřadí její vlastnosti.
4. Částice, které překročily dobu svého života, nebo jiné hranice, zaniknou.
5. Systém se zobrazí.

2 Vlastní implementace

Úkolem mojí práce bylo vytvořit bludiště, ve kterém budu zobrazovat efekty. Po dohodě s vedoucím bakalářské práce inženýrem Janem Pečivou jsem měl implementovat následující efekty. Per-pixel osvětlování (Phongovo stínování), Normal Bump Mapping, použít systém částic, případně nějaký další a aspoň jeden z výše uvedených za použití shaderů

2.1 Modelování a zobrazení objektů

Důležité pro vykreslení scény je vytvoření objektů, které se ve scéně budou objevovat. Nejjednodušší cestou jak tyto objekty vytvořit, je použít modelovací program. Existují různé modelovací programy. Počínaje profesionálními jako třeba 3D Studio, Blender apod., které jsou určené na širší využití. Jsou ale také modelovací programy určené jen ke specifickému využití, jako například programy pro návrh bytu. Důležitou roli při výběru modelovacího programu hrála jejich cena. Zatímco 3D Studio patří mezi komerční programy, Blender vychází pod licencí GNU GPL. Tím byl výběr programu jasný a práce mohla začít.

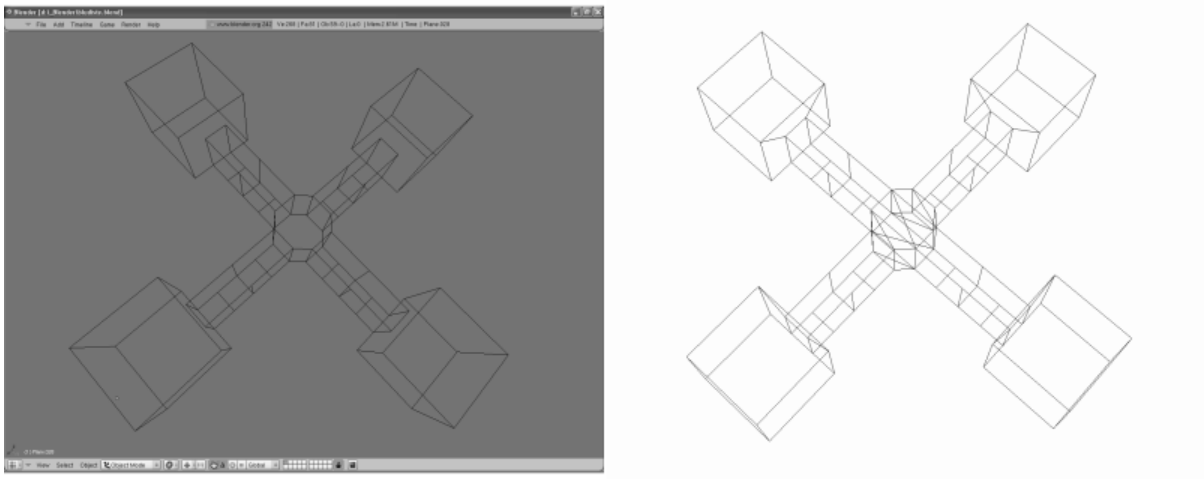
Blender je software určený k modelování a vykreslování 3D počítačové grafiky a animací s využitím nejrůznějších technik, jako je například sledování paprsku nebo radiosita. Celé rozhraní je vykreslováno pomocí OpenGL, takže je hardwarově urychlováno a hlavně je díky tomu Blender snadno přenositelný na různé platformy. Blender umožňuje vytvářet modely pomocí jednoduchých ploch založených na vertexech. Umožňuje ale také vytváření pomocí parametrických ploch a pomocí křivek. Pro svou práci si ale vystačím s jednoduchými plochami.

Další důležitou složkou při modelování je fakt, jestli prostředí, ve kterém objekty potom chceme zobrazovat, umí tyto objekty načíst a zobrazit. Open Scene Graph neumí otevřít soubory tvořené v Blenderu, i když má knihovny pro načtení velkého množství různých formátů. Po nainstalování pluginu do Blenderu, uměl exportovat přímo do nativního formátu Open Scene Graphu a tak načtení modelu bylo zcela stoprocentní.

Při studování jak se modeluje v Blenderu jsem vycházel z internetových stránek BlenderWiki [7]. Největší problém při modelování v externích programech bývá v uložení textur. Modelovací programy většinou dokážou namapovat texturu na objekt automaticky, což ale neumí OpenSceneGraph, ten potřebuje mapovací souřadnice. Tyto souřadnice jsem musel vytvořit v Blenderu pomocí UV mapování, kde jsem každému jednomu vertexu přiřadil právě jeden bod textury. Tím je zajištěno správné zobrazení objektu s texturou.

V Blenderu jsem vytvořil jen jednoduché modely, se kterýma jsem dále pracoval v Open Scene Graphu, kde jsem také vytvářel všechny efekty. Blender se pro mě stal vhodným nástrojem pro modelování, protože zároveň zobrazuje editovaný objekt. Při zobrazení modelu v Open Scene Graphu

byly výsledky stejné jako v Blenderu. Spolupráce těchto dvou prostředí je tedy naprosto bezproblémová.



Obrázek 13: Zobrazení bludiště v Blenderu (vlevo) a v OpenSceneGraphu (vpravo)

2.2 Pohyb v bludišti

Open Scene Graph nabízí už automaticky pohyb kamery pomocí myši. Tento způsob pohybu je vhodný pro prohlížení všech objektů ve scéně, nikoliv však pro hru, kdy by hráč měl mít možnost vidět své okolí jako by v něm byl a pohyboval se v něm. Z toho důvodu bylo potřeba změnit základní postavení kamery a její pohyb ve scéně.

První z věcí, kterou bylo třeba provést, bylo změnit OSG implicitní kameru na svou definovanou. To se dá zařídit tak, že nastavíme, aby se kamera nastavovala podle matice, která se nazývá transformační matice. Transformační matice obsahuje informace o posunu, rotaci, měřítku a zkosení. Pro nastavení kamery nám bude stačit transformační matice pro posun a pro rotaci. Jelikož se jedná homogenní souřadnice, tak je možné provést skládání transformací. V našem případě vynásobíme transformační matici rotace s transformační maticí posunu. Přičemž je důležité dodržet pořadí matic při násobení.

Transformační matice rotace se skládá ze tří matic rotací založených na rotacích kolem základních os. Rovnice 12 ukazuje jak se vypočítá konečná transformační matice rotace R. Pro výpočet matice je zapotřebí znát úhly otočení. Úhel otočení kolem osy x značí úhel α, úhel β označuje otočení kolem osy y a úhel γ otočení kolem osy z. Výsledná transformační matice je pak jejich vynásobením.

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{12}$$

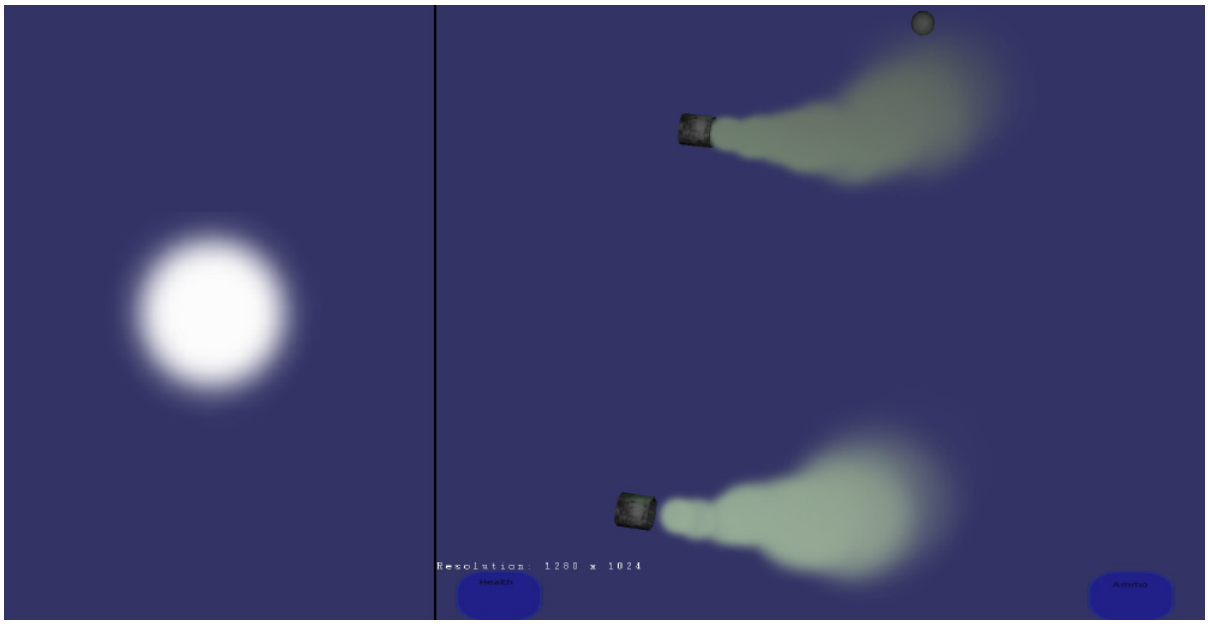
Výpočet transformační matice posunu je o něco málo jednodušší než výpočet rotace. K výpočtu jsou potřeba pouze velikosti posunů na jednotlivých osách. Jak vypadá výpočet transformační matice posunu `P` ukazuje rovnice 13. `ΔX`, `ΔY` a `ΔZ` jsou velikosti posunů na osách x, y a z.

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta X & \Delta Y & \Delta Z & 1 \end{pmatrix} \tag{13}$$

Celková transformační matice `T` vznikne vynásobením matic `T = RP`. Pak už jen stačí nastavit aby se tato matice použila pro nastavení kamery při každém zobrazení. Pohyb ve scéně je zajištěn měněním atributů těchto matic. Pro otáčení se mění úhly matic rotace, a pro pohyb se mění velikosti posunů matice posunu.

2.3 Efekt se systémem částic

Systémem částic je možné implementovat nejrůznější efekty. Já jsem se rozhodl vytvořit rouru a páru z ní tryskající. Nejdůležitější při vytváření systému částic je nastavení parametrů, které se použitou pro každou částici. Nastavit se může doba života, jak často se má vytvořit další částice, jak se má pohybovat a podobně. Důležitý je také vzhled každé částice. Ten je definovaný texturou, nejlépe s průhledností. Na obrázku 14 vlevo je textura, kterou jsem vytvořil a použil pro páru (vpravo).



Obrázek 14: Textura páry (vlevo) a její použití pro systém částic v aplikaci (vpravo)

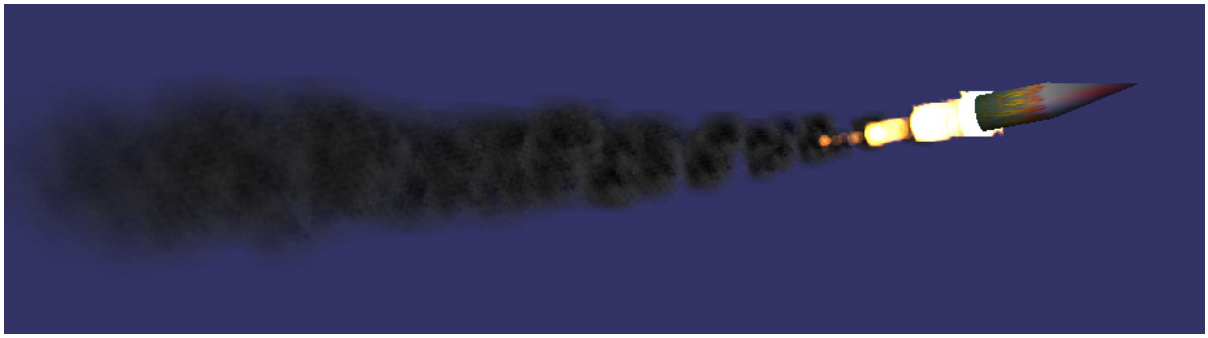
OpenSceneGraph má vytvořené rozhraní v podobě tříd pro definování systému částic. Já jsem všechny atributy částic sjednotil do jedné třídy, s kterou se vytvářeli čističové systémy ještě jednodušeji. OpenSceneGraph také nabízí třídy pro definování chování částic. Lze tedy jednoduše

nastavit počáteční pozici každé částice, jak bude velká, jak se bude zvětšovat nebo zmenšovat, jak se bude měnit její rychlost, kolik částic se vytvoří během jedné sekundy. Důležité je ke každé rouře přidat modulární emitor (anglicky modular emitter), který poskytuje jednoduchý mechanismus pro řízení částicového systému. Samotný částicový systém je potom zařazen v kořenu stromu scény a modulární emitor se postará o vytváření jednotlivých částic. Důležité je vytvořit pro každé místo, kde se mají vytvářet částice, samostatný emitor. Částicový systém stačí když je jen jeden. Podle obrázku 14 je vidět, že jsem použil jeden částicový systém a 2 emitory.

2.4 Pohyb těles ve scéně

Jakým způsobem se umístí těleso na určitou pozici je vysvětleno v kapitole 2.2. V případě měnění pozice kamery je situace o něco jednodušší, protože měnit její polohu stačilo podle stisknutí kláves na klávesnici. Stisknutí klávesy vyvolá událost, která je odeslána aplikace a úkolem aplikace je tuto událost odchytit a vykonat určitou akci na základě toho, jaká klávesa byla zmáčknuta. Jedná se o zcela standardní postup a OpenSceneGraph má pro tento případ vytvořenou abstraktní třídu. Stačí jen udělat soupis, které všechny klávesy chceme odchytit a co se podle nich má vykonat. V naše případě, když chceme přemístit kameru, stiskneme klávesu šipky nahoru. Tím se vygeneruje patřičná událost. Tuto událost odchytíme a posuneme kameru dopředu.

Složitější je ale situace pokud chceme hýbat s ostatními tělesy, které nejsou řízeny klávesnicí ani jiným periferním zařízením. Možností je více a já jsem aplikoval dvě různé. Ve scéně jsou dvě světla, která stále lítají tam a zpět a to v závislosti na počtu zobrazených snímků. Takové měnění pozice by se dalo nazvat pseudo-reálné měnění pozice. Dokud je počet zobrazených snímků za sekundu konstantní je vše v pořádku. Ale v případě, že tento počet kolísá, například z důvodu náhlé větší náročnosti scény, se takhle závislé objekty zpomalí a někdy až zastaví. Někdy může takové chování být užitečné, ale v jiných situacích je zcela nevhodné. Pro měnění pozice světla z důvodů lepší ukázky implementovaných efektů je tento způsob dostačující.



Obrázek 15: Letící raketa

Na obrázku 15 je zobrazena letící raketa. Jedná se o další implementaci systému částic. Zde jsou použity dva systémy. Jeden pro zobrazení ohně za raketou a druhý pro černý dým. V případě pohybu rakety je ale lepší vytvořit systém, který nebude závislý na rychlosti zobrazování. Bylo tedy

potřeba vytvořit systém měnění pozice , který se aktualizuje v závislosti na čase. Vytvořil jsem třídu `PositionUpdater`, která uchovává objekty, jejich pozici, rychlost v podobě vektoru, čas života a čas poslední aktualizace. Aktualizace pozice se provádí při každém vykreslení pro všechny objekty vložené do `PositionUpdateru` a probíhá v následujících krocích:

1. Pokud doba života překročila hranici, tak objekt vyjmi. Tím je zaručeno i další nezobrazování objektu. Pokračuj od bodu 1 pro další objekt.
2. Pokud čas poslední aktualizace objektu je vyšší než perioda obnovení, tak pozici změň o vektor rychlosti. Nastav čas poslední aktualizace na aktuální čas.
3. Pokračuj od bodu pro další objekt.

Měnění pozice objektů v závislosti na čase na rozdíl od závislosti na rychlosti zobrazování se dá použít při zobrazování na více počítačích. Například při hraní hry na dvou počítačích ve hře pro dva hráče.

Jistě je více možností jak provádět měnění pozic objektů v závislosti na čase. Snažil jsem se použít nějakou jednoduchou a dostačující metodu.

2.5 Používání shaderů

Pro další efekty jsem se rozhodl používat shadery. Jako implementační jazyk jsem vybral OpenGL Shading Language hlavně z důvodu velmi dobré podpory ze strany OpenSceneGraphu.

Nejdůležitější součást používání jsou vertex shader a fragment shader. Tyto programy jsou uloženy ve dvou souborech. Doporučuje se pro vertex shader používat příponu souboru `vert` a pro fragment shader příponu `frag`.

Pokud máme vytvořeny shadery musíme je přeložit pro náš grafický adaptér. K tomu nám opět dobře poslouží OpenSceneGraph, který má vytvořené třídy pro aplikování shaderů, takže stačí jen říct, které soubory se mají použít. Pak už jen říkámé, které shadery se mají zrovna použít.

Jelikož obsah jak fragment shaderu tak vertex shaderu závisí na tom, který problém mají řešit, je zbytečné se o nich nějak víc zmiňovat. Důležité je jen dodržet následující pravidla:

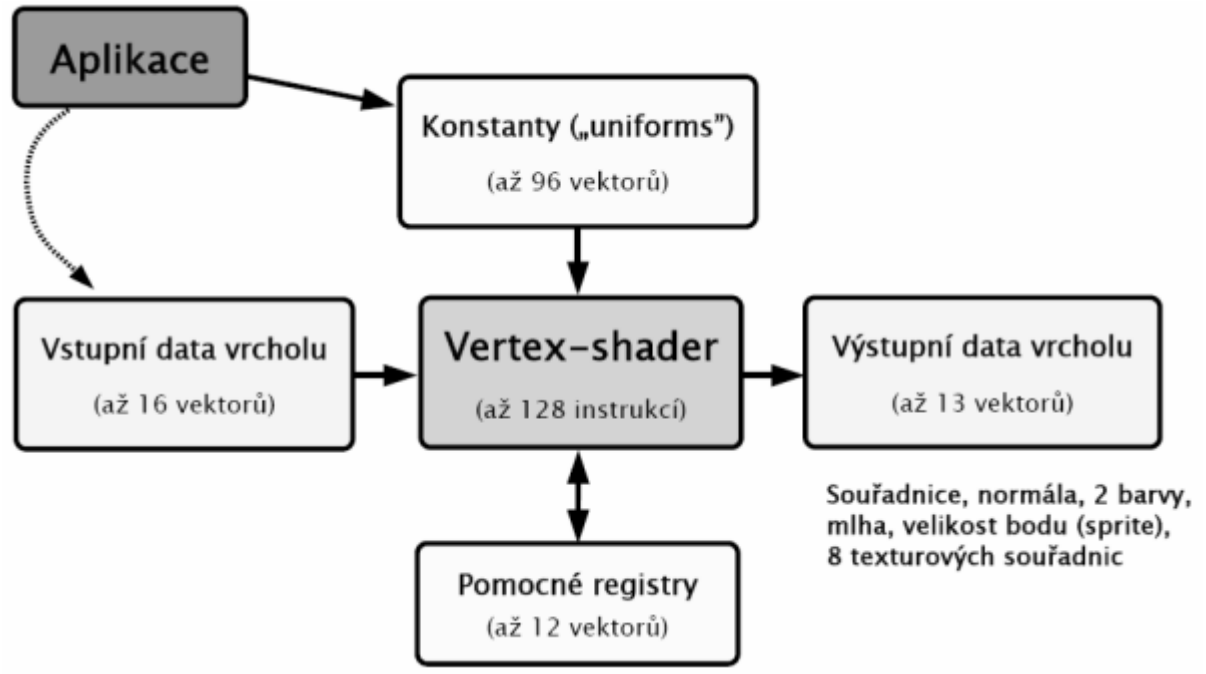
1. Ve vertex shaderu je potřeba uložit pozici každého vertexu
2. Ve fragment shaderu je potřeba uložit barvu každého fragmentu

2.5.1 Vertex shader

Ve vertex shaderu je umožněno:

- Transformovat vertexy
- Transformovat a normalizovat normálové vektory
- Vypočítat nebo transformovat texturovací souřadnice

Neumožňuje však měnit počet vrcholů. Pro komunikaci směrem od aplikace k vertex shaderu je možné použít konstanty, tak zvané uniforms. Uniforms jsou ve vertex shaderu pouze pro čtení, lze je měnit pouze v aplikaci. Dále je možné měnit atributy vertexů. Viz obrázek 16.



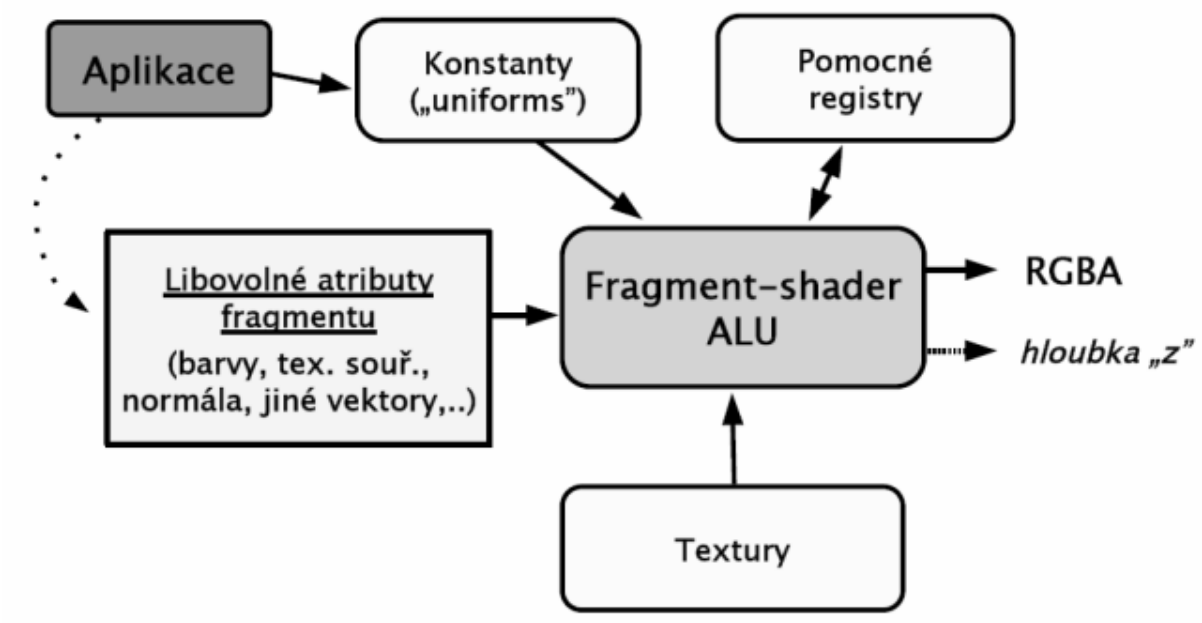
Obrázek 16: Prostředí vertex shaderu

2.5.2 Fragment shader

Ve fragment shaderu je umožněno:

- Provádět aritmetické operace s interpolovanými hodnotami
- Čtení dat z textur
- Aplikace textur a jejich kombinace
- Výpočet osvětlení, mlhy, …
- Závěrečná syntéza barvy fragmentu
- Možnost modifikace hloubky fragmentu „z“

Není však možné měnit počet nebo polohu fragmentu. Ve fragment shaderu je taky možné použít uniform proměnné. Je taky možné převádět proměnné z vertex shaderu do fragment shaderu. Viz obrázek 17.



Obrázek 17: Prostředí fragment shaderu

2.6 Per-pixel osvětlování

První efekt, který jsem implementoval pomocí shaderů se nazývá per-pixel osvětlování, neboli Phongovo stínování.

Ve vertex shaderu se všechny vertexy počítají v souřadnicích obrazu, anglicky *view-space*. Z toho důvodu je třeba mít všechny ostatní pozice v těchto souřadnicích. Proto bylo potřeba přepočítávat pozice světel každý snímek do těchto souřadnic, pokud v nich ještě nebyli. V aplikace jsem použil jedno světlo s nastavením pozice v souřadnicích obrazu a ostatní ve světových souřadnicích. Bylo tedy nutné vytvořit třídu, která přepočítává pozice. Do této třídy přiřadí každé světlo, které má být přepočítané a každý snímek se provede přepočet pro všechna světla.

Při počítání per-pixel osvětlování můžeme vycházet z Phongova osvětlovacího modelu. Musíme tedy znát čtyři vektory pro každý fragment. Všechny vektory kromě vektoru reflexe je třeba vypočítat ve vertex shaderu. Vektor reflexe můžeme vypočítat ve fragment shaderu. Dále musíme nastavit texturovací souřadnice, které nijak neměníme. Nezměněná zůstane taky pozice vertexu.

Ve fragment shaderu probíhá celý výpočet osvětlení pro každý fragment. Přitom se využívá toho, že všechny atributy vertexů nastavené ve vertex shaderu jsou automaticky přepočítány pro každý fragment. Stačí jen vypočítat vektor reflexe, podle druhu světla. Zda se jedná o směrové světlo se zjistí z vektoru pozice světla. Pokud složka `w` je rovna 0, jedná se o směrové světlo. V ostatních případech je třeba vypočítat útlum apod. Odkaz na texturu fragment shaderu pošleme pomocí proměnné uniform. Nakonec jen nastavíme barvu fragmentu. Na obrázku 18 je vidět výsledek shaderu.

Obrázek 18: Osvětlená zídka za použití per-pixel osvětlování

2.7 Normal Bump mapping

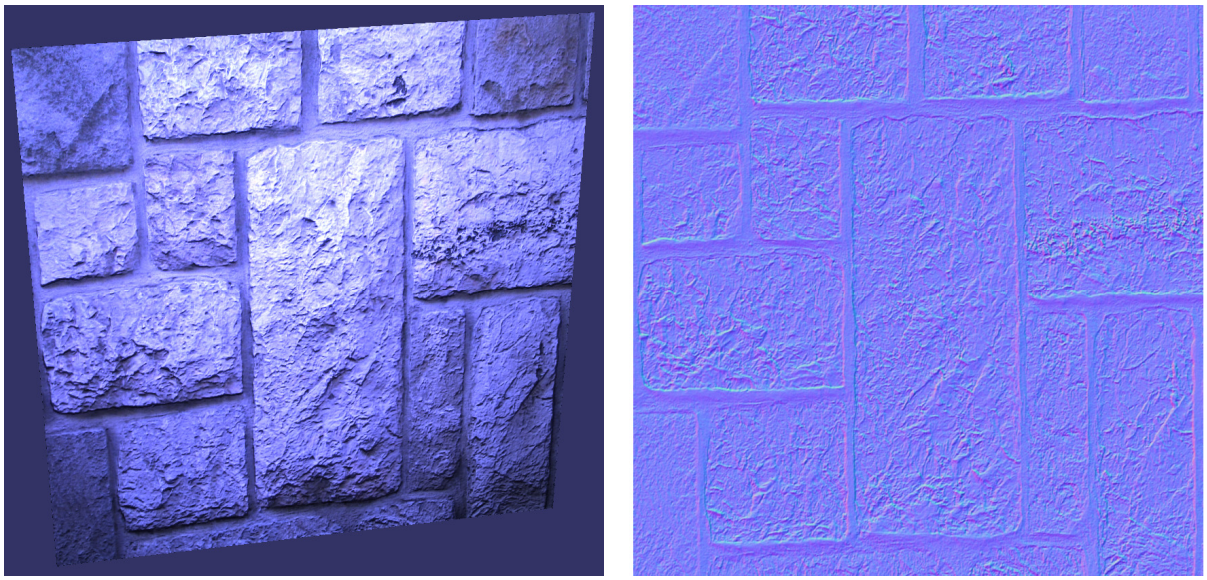
Další efekt vytvářený pomocí shaderů. Ve své podstatě je normal bump mapping velmi podobný per-pixel osvětlování. Je ale zapotřebí jedna textura navíc, která udává každému pixelu, potažmo texelu, směr normály. Pomocí normal bump mappingu bylo dosaženo daleko lepšího zobrazování.

Na obrázku 19 vpravo je vidět použitá normal mapa. Jsou v ní uloženy směry normály v tečném prostoru. Jedná se o prostor, kde vertex je v bodě `{0, 0, 0}` a normála je ve směru `{0, 0, 1}`. Dále jsou zapotřebí další dva vektory označované jako tečna ve směru `{1, 0, 0}` a binormála ve směru `{0, 1, 0}`. Modrá barva z normal mapy odpovídá normále, červená odpovídá tečně a zelená binormále. Z toho také vyplývá odpověď na otázku proč normal mapy mají celkový modrý nádech? Je to proto, že většina normál nejsou nijak deformovány a odpovídají směru normály vertexu.

Pokud chceme požívat normal mapu v tečném prostoru, musíme všechny vektory používané při osvětlování také přepočítat do tečného prostoru. K tomu lze použít rovnici 14, kde se vektor v objektovém prostoru vynásobí maticí složenou z vektorů tečny, binormály a normály a tím získáme hledaný vektor v tečném prostoru.

$$P_{TS} = P_{OS} \begin{pmatrix} T_{n(x)} & B_{n(x)} & N_{n(x)} \\ T_{n(y)} & B_{n(y)} & N_{n(y)} \\ T_{n(z)} & B_{n(z)} & N_{n(z)} \end{pmatrix} \tag{14}$$

Nyní když jsou všechny vektory připravené pro vypočítání barvy v každém texelu může se provést výpočet. Ten je stejný jako při per-pixel osvětlování s tím rozdílem, že normály jsou načteny z druhé textury, kterou pošleme taky jako proměnnou typu uniform. Jak vypadá výsledek je vidět na obrázku 19 vlevo.



Obrázek 19: Osvětlená zídka za použití Normal Bump mappingu (vlevo) a normal mapa (vpravo)

A jakým způsobem se dá vytvořit taková normal mapa? V zásadě jsou dva postupy. První vychází ze dvou objektů, kde jeden je s nízkým počtem plošek a ten druhý s mnohonásobně vyšším počtem plošek. Z nich se pak vytvoří normal mapa. Druhý způsob vychází čistě z textury. Na vstupu takového programu, který vytváří normal mapu z textury, je soubor s texturou a výsledkem je vytvořená normal mapa. Takové vytváření je mnohem jednodušší, ale má jisté nedostatky.

2.8 Falešné volumetrické čáry

Anglicky *fake lines*. Při vytváření falešných čar jsem vycházel z [8]. Pomocí falešných čar lze vytvářet různé efekty. Například laserové paprsky, zvýrazňovat obrysy objektů, apod.

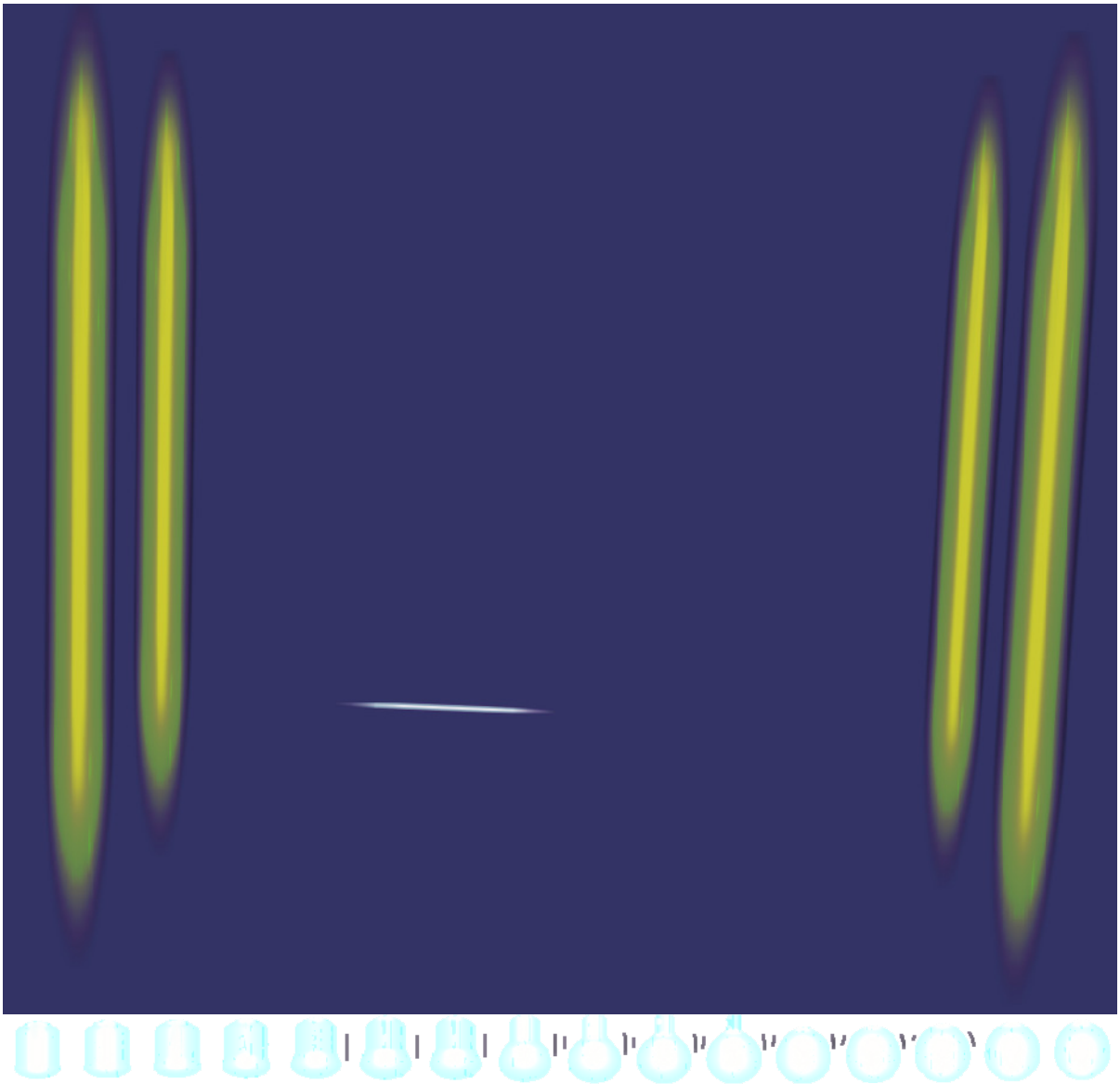
Základní myšlenkou je vytvořit obdélník reprezentující čáru co nejjednodušší formou. To znamená vytvořit falešnou čáru ze dvou bodů, šířky, barvy a textury. Jelikož se ale ve skutečnosti jedná o obdélník, body musí být čtyři. Využijeme toho, že ve vertex shaderu můžeme s body libovolně pohybovat a pošleme mu počáteční a konečné body dvakrát. Poté je od sebe roztáhneme o polovinu šířky každý.

Obdélník reprezentující čáru už máme, zbývá nanést texturu. Nanášení textury na obdélník může ale působit problémy. Dokud pohled na obdélník je kolmý je vše v pořádku. Pokud bychom ale

obdélníkem otáčeli nevypadal by obraz pěkně. Z toho důvodu je třeba zjišťovat pod jakým úhlem čáru vidíme. Textura obsahuje šestnáct částí. Na základě velikosti úhlu se použije určitá část. Pokud úhel se blíží 90° použije se první část. Pokud se úhel blíží 0° použije se poslední šestnáctá část. Jednotlivé části je vidět na obrázku 20 dole.

Použití jednotlivých částí se může ještě vylepšit v závislosti na tom, jak dlouhá a široká se čára zobrazí na displeji. Pokud je čára dlouhá, bude rozpětí úhlu, při kterém se použije daná část, o něco větší. Je to dáno tím, že pokud je čára delší, nedeformuje se tak rychle výsledná textura.

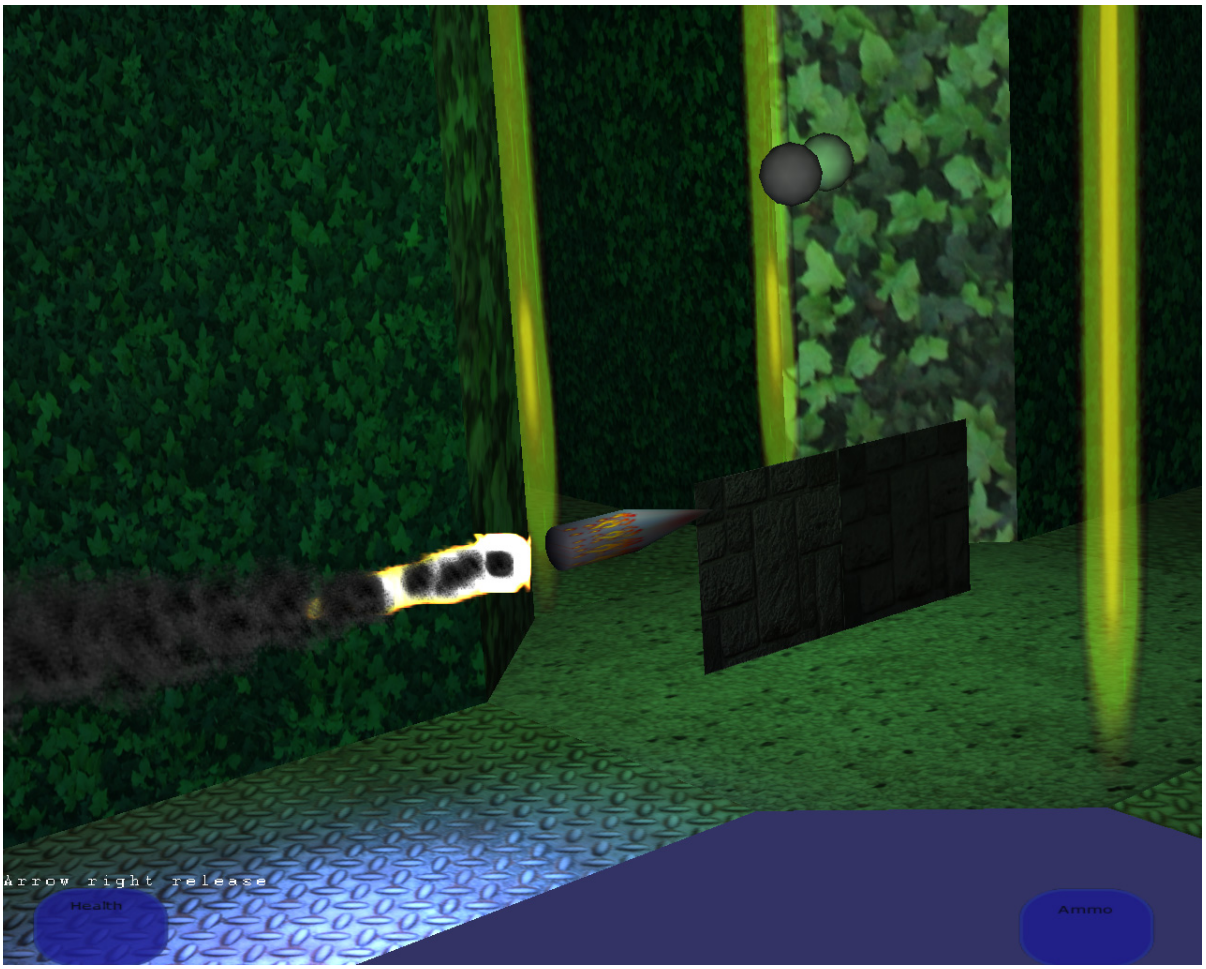
Falešné čáry jsem použil na laserové střely a světelné sloupky. Na laserovou střelu byla použitá bílá barva vertexů, na světelné sloupky žlutá barva. Tyto efekty je možno vidět na obrázku 20 nahoře.



Obrázek 20: Laserová střela a světelné sloupky (nahoře), použitá textura (dole)

2.9 Souhrnné informace

Projekt řeší jednotlivé implementace efektů. Vytvořena je jedna hlavní scéna, ve které se všechny efekty zobrazují. Snažil jsem se o takovou implementaci, aby její použití ve hrách bylo co nejjednodušší. Jeden obrázek z konečné scény je na obrázku 21. Je zde použito per-pixel osvětlování na zdi bludiště, což je velice náročné a může způsobovat na starších grafických adaptérech. Z toho důvodu jsem vytvořil ještě jedno bludiště složené velkého počtu plošek a aplikoval na ni jen Gouraudovo stínování. Výsledné zobrazení je podobné co se týká vzhledu, ale je mnohem rychlejší.



Obrázek 21: Zobrazení konečné scény

2.10 Zveřejnění na internetu

Jedním z požadavků na kompletní projekt, bylo zveřejnění na internetu. Veškeré potřebné soubory, binární a zdrojové texty, popis ovládání a další, je možné najít internetových stránkách `http://www.stud.fit.vutbr.cz/~xrakus01/BP/`. Je pravděpodobné, že se tam také objeví novější verze.

2.11 Integrace projektů

Projekt byl navržen a připraven pro integraci s projektem Milana Kleibela. Z důvodů nekompletnosti ovšem tato integrace nemohla proběhnout.

3 Závěr

Cílem této práce bylo vytvořit efekty, které se dají použít v počítačových hrách za použití nejnovějších technik. Měl jsem zjistit o jaké techniky a efekty se jedná a pokusit se některé z nich implementovat.

Pomocí systému částic se jednoduše vytvořila raketa, která se dá použít ve hrách. Ačkoliv se jedná o jednoduchou implementaci i použití, je tímto efektem dosaženo dobrých výsledků. Mnou navržené per-pixel osvětlování je bez problémů použitelné pro menší objekty. V případě, že chceme použít větší objekty je lepší je vytvořit s větším počtem vertexů a použít Gouraudovo stínování. To samé platí pro objekty s normal mapami, s kterými je dosaženo velmi dobrého zobrazení. Pomocí falešných volumetrických čar se dají jednoduše vytvářet nejrůznější střely, nejlépe však vypadají laserové paprsky. Použití je ovšem velmi závislé na vytvořené textuře.

Pro mě tato práce byla velmi přínosná. Prohloubil jsem své znalosti z oblasti počítačové grafiky. Naučil jsem se pracovat se shaderema, které se v dnešní době hojně používají. Vytváření efektů se mi natolik zalíbilo, že mám v plánu vytvářet další a zdokonalovat již vytvořené.

Vizuální efekty v dnešních hrách hrají velmi velkou roli. Bohužel se tak většinou děje na úkor hratelnosti a celkové zábavy u hry. Vývojáři her se většinou snaží o co nejpěknější vzhled, ale hratelnost zůstává až někde na pokraji. Mezi přední vývojové týmy v oblasti grafických enginů, ve kterých implementují i nejrůznější efekty, patří Valve Corporation. Jejich nejnovější a nejúspěšnější herní engine Source engine byl poprvé použitý ve hře Half-life 2 a je používán i v mnoha dalších nových hrách. Je v něm implementována většina efektů, které jsem zde popsal a velká skupina dalších.

Literatura

[1] Segal, M., Skelet, K. *The OpenGL® Graphics System: A specification*

[2] Žára, J., Beneš, B., Sochor, J., Felkel, P. *Moderní počítačová grafika*, druhé, přepracované a rozšířené vydání, kompletní průvodce metodami 2D a 3D grafiky, Computer PRESS, 2004, ISBN 80-251-0454-0

[3] Silicon Graphics, OpenGL Programming Guide, (cit. 7. 5. 2007)
URL <http://www.glprogramming.com/red/>

[4] Kabát, Z. 3D technologie: Bump Mapping, (cit. 7. 5. 2007)
URL <http://www.svethardware.cz/art_doc-6D2AC72379CA43C6C1256ED6006B1BA2.html>

[5] Cohen, J., Olano, M., Manocha, D. *Appearance-Preserving Simplification*, (cit. 7.5. 2007)
URL <http://www.cs.unc.edu/~geom/APS/APS.pdf>

[6] Kessenich, j., Baldwin, D., Rost, R. *The OpenGL® Shading Language*, (cit. 7. 5. 2007)
URL <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf>

[7] BlenderWiki, (cit. 7. 5. 2007)
URL <http://wiki.blender.org/index.php/Main_Page>

[8] Lorach, T., Fake Volumetric lines, (cit. 7. 5. 2007)
URL <http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/OpenGL/src/cg_VolumeLine/docs/VolumeLine.pdf>

Seznam příloh

Příloha 1

Požadavky na spuštění a popis ovládání

Aplikace byla testována na této sestavě:

Procesor: AMD64 3500+

Paměť: 1024MB

Grafická karta: NVIDIA GeForce 7600

Aplikaci by neměl být problém spustit i na nižší konfiguraci. Největší zatížení je na grafické kartě. Pro spuštění je ale nutné mít nainstalované Visual Studio .NET.

Ovládání je rozděleno na dvě části. Jedna je stejná jako standardní ovládání OpenSceneGraphu. Některé důležité klávesy:

```
C:          Přepnutí ovládání
Mezera:     Návrat kamery na celkový pohled na scénu
F:          Přepnout mezi full screen / window mode
B:          Backface culling
Escape:     Konec aplikace
CTRL:       Výstřel
```

Ovládání myší.

Druhé ovládání je mnou vytvořené. Ovládá se pomocí kláves a je podobné jako ovládání ve hrách. Některé klávesy:

```
C:                Přepnutí ovládání
Šipky:            Pohyb
HOME:             Návrat na nulovou pozici
CTRL:             Výstřel
F, B, Escape:     Stejně jako při ovládání OpenSceneGraphu
, a .:            Posun vlevo, vpravo
PageUp&Down:      Posun nahoru, dolů
```

Je vhodné při přepnutí mezi mým ovládáním a ovládáním OSG klávesou C také zmáčknout mezerník. Bez mezerníku se kamera nevrátí do pozici na celkový pohled na scénu.