VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

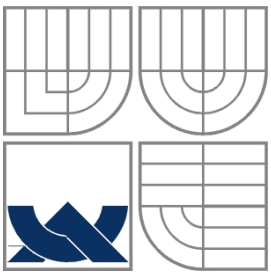
ZOBRAZOVÁNÍ SCÉNY V MODERNÍCH POČÍTAČOVÝCH HRÁCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

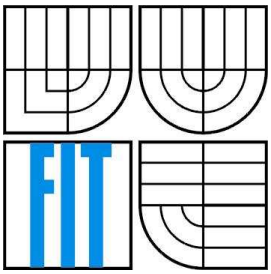
AUTOR PRÁCE
AUTHOR

Bc. MARTIN WILCZÁK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZOVÁNÍ SCÉNY V MODERNÍCH POČÍTAČOVÝCH HRÁCH

SCENE RENDERING IN MODERN COMPUTER GAMES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN WILCZÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RUDOLF KAJAN

BRNO 2011

- Zadání diplomové práce -

Abstrakt

Práce popisuje metody používané pro osvětlování rozsáhlých scén v moderních počítačových hrách. Jsou porovnány možnosti klasického (dopředného) osvětlování a rozšiřující se metoda odloženého stínování. Krátce jsou popsány možnosti vykreslování pomocí metody sledování paprsku. V práci jsou obsaženy i poznatky o vrhaných stínech, částicových systémech a post-processing efektech. Nakonec je navržena architektura pro zobrazování komplexních scén za pomoci XNA a popsána implementace použitá ve výsledné hře.

Abstract

This thesis describes methods for lighting calculations of large scenes used in modern computer games. Forward shading and deferred shading methods are discussed and compared. Capabilities of raytracing are shortly described. There are some information about various methods for casting shadows, simulation of particle systems and applying post-processing effects. In the end there is a design of architecture for rendering complex scenes with use of XNA and description of implementation used in resulting game.

Klíčová slova

Dopředné stínování, odložené stínování, sledování paprsku, částicový systém, stíny, post-processing efekty, XNA.

Keywords

Forward shading, deferred shading, raytracing, particle system, shadow, post-processing effects, XNA

Citace

Martin Wilczák: Zobrazování scény v moderních počítačových hrách, diplomová práce, Brno, FIT VUT v Brně, 2011

Zobrazování scény v moderních počítačových hrách

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Rudolfa Kajana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Martin Wilczák
24. 5. 2011

Poděkování

Děkuji svému vedoucímu, Ing. Rudolfu Kajanovi, za jeho ochotu a vstřícný přístup. Dále chci poděkovat kolegům, Bc. Davidu Jozefovovi a Bc. Michalu Zachariášovi, se kterými jsem pracoval na výsledné aplikaci.

Obsah

Obsah....1
1 Úvod....2
2 Renderování scény....3
2.1 Forward shading....3
2.2 Deferred shading....6
2.3 Raytracing....10
3 Speciální efekty....12
3.1 Částicové systémy....12
3.2 Stíny....14
3.3 Post-processing....17
4 XNA....21
5 Návrh....22
5.1 Požadavky....22
5.2 Renderer....22
5.3 Částicové systémy....24
6 Implementace....26
6.1 Renderování....26
6.2 Post-processing efekty....32
6.3 Částicové systémy....34
6.4 Integrace do hry....39
7 Experimenty....41
7.1 Porovnání metod vykreslování....41
7.2 Osvětlení metodou deferred shading....42
7.3 Částicové systémy na CPU a GPU....43
8 Závěr....46
Literatura....47
Přílohy....49

1 Úvod

V dnešní době se hardware i software vyvíjí závratnou rychlostí. To, co bylo ještě před pár lety nemyslitelné kvůli technickým limitům, se dnes stává skutečností. To samozřejmě platí i v oblasti počítačové grafiky. Výkon grafických karet je dnes na úrovni superpočítačů z doby minulého století a stále roste.

Největším tahounem v oblasti zobrazování v reálném čase jsou samotní hráči. Požadují stálé nové a lépe vypadající hry a vývojáři se jim v honbě za ziskem snaží vyhovět. A jak hráči, tak i vývojáři tlačí na výrobce grafických řešení a chtějí vyšší výkon. Ať už kvůli vyššímu počtu snímků za sekundu pro současné hry, nebo novým schopnostem pro hry budoucí.

Díky této neutuchající honbě se stále rozšiřují možnosti zobrazování. Efekty dosažitelné na moderním hardware se tak již velmi blíží fotorealistickým obrázkům. A to nejen proto, že se navyšuje výkon a tím i komplexita zobrazovaného obsahu, ale také proto, že se díky novým schopnostem mohou používat nové způsoby ve vykreslování. Často to bývají již dávno vymyšlené, ale dosud nerealizovatelné metody.

Tato práce rozebírá různé metody používané pro zobrazování virtuálních scén v reálném čase. Začíná metodou klasického renderingu se standardním výpočtem osvětlení scény (forward shading), která se používá již od samého počátku grafických akcelerátorů. Kromě ní rozebírá na poměry v IT celkem starou, ale až nyní využívanou metodou odloženého stínování (deferred shading), a nakonec také raytracingem, který sice není nasazován v komerčních hrách, ale již je možné jej použít pro zobrazování v reálném čase a zdá se, že by mohl být ne příliš vzdálenou budoucností.

Práce se rovněž věnuje různým speciálním efektům, které se používají v moderních hrách, počínaje částicovými systémy pro simulaci kouře, ohně a podobných jevů. Dále jsou popsány různé způsoby vytváření stínů a v neposlední řadě se věnuje možnostem post-processingu, díky kterému se dá dosáhnout množství zajímavých efektů.

Dále je v práci rozebrán framework XNA, který lze použít pro rychlou tvorbu her pro různé platformy od firmy Microsoft. XNA je velmi mocný nástroj a vývojářům předkládá velké množství struktur a principů, které se používají v herních enginech, a které by vývojář bez jeho použití musel programovat sám. Ušetří tak čas a může se zaměřit na ostatní aspekty samotné hry.

V další části se práce věnuje návrhu rendereru, který využívá kombinaci metod forward a deferred shadingu pro zobrazování komplexních scén. Rovněž je zde rozebrán návrh pro aplikaci post-processing efektů na vytvořený obraz. Je zde také navrhnut způsob, jakým lze na moderních GPU akcelerovat simulaci rozsáhlých částicových systémů.

V následující kapitole je popsán způsob, jak byly navržené části implementovány, jaké výhody a problémy XNA se během těchto prací projevily a jak bylo nutné upravit návrh pro dosažení funkčnosti či pro zajištění lepších výsledků. Dále je popsáno, jak byly tyto implementované části začleněny do týmově vyvíjené hry.

V kapitole Experimenty jsou porovnány metody forward a deferred shading a také různé verze částicových systémů. Také je zde ověřena komplexita výpočtů osvětlení u metody deferred shading.

V závěru práce jsou shrnuty poznatky o implementované aplikaci, použitých metodách, XNA frameworku a také jsou rozebrány možnosti budoucího rozšíření.

V diplomové práci je uvedeno množství původně anglických výrazů, pro které buď neexistuje český ekvivalent, nebo jeho použití není příliš vhodné. K těmto termínům bude připsán jejich přibližný překlad a následně bude používána jejich anglická podoba.

2 Renderování scény

V současné době se v počítačových hrách nejčastěji používají dvě metody renderování: Forward shading a Deferred shading. Forward shading je klasický způsob stínování, kdy se ihned při rasterizaci spočítá také osvětlení kresleného objektu. Deferred shading využívá nových schopností GPU a funguje ve dvou krocích. V první fázi se vykreslí pouze informace o geometrii a v druhém kroku se tyto uložené informace použijí pro výpočet osvětlení a vykreslení výsledného obrazu.

Raytracing se pro hry stále používá pouze experimentálně, ale výkon je dnes již dostačující a dá se tedy předpokládat, že se s novějšími a ještě výkonnějšími procesory začne pomalu prosazovat.

2.1 Forward shading

Forward shadingem je označován klasický způsob stínování, který se používá již od prvních grafických karet. Stále se často používá pří vykreslování v počítačových hrách.

Obrázek 1: Snímek ze hry Crysis vydané v roce 2007, jejíž engine využívá standardní forward shading.

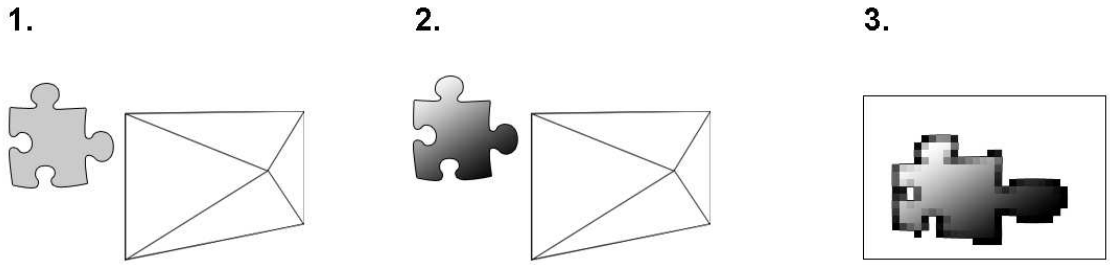
2.1.1 Rasterizace

Vykreslování pomocí forward shadingu věrně kopíruje vykreslovací řetězec grafických karet. Přestože lze pipeline moderních GPU programovat pomocí komplexních shaderů, je základní princip forward shadingu značně jednoduchý. Do grafické karty se předá geometrické primitivum, to je zpracováno po vrcholech, poté rasterizováno a nakonec se zpracovávají pixely. Každé primitivum je zpracováno nezávisle na ostatních a jejich poloha vůči kameře (hloubka) a vzájemné překrývání jsou řešeny pomocí z-bufferu, v němž jsou uloženy vzdálenosti nejbližších pixelů dosud rasterizovaných primitiv. Tento princip je sice jednoduchý a hardwarově snadno implementovatelný, ale má nevýhodu v tom, že pokud je ve scéně spousta překrývajících se objektů, pak se zbytečně vykreslují pixely,

které budou o chvíli později překresleny pixely novými. Při použití forward shadingu se situace zhorší, máme-li ve scéně komplexní osvětlovací model.

2.1.2 Výpočet osvětlení

K výpočtu osvětlení dochází na úrovni vrcholů, pixelů či nějaké kombinaci. To závisí na výkonu cílového hardwaru a požadované vizuální kvalitě. Tak jako se pro modely používají LOD techniky, i v případě výpočtu osvětlení se využívá faktu, že vzdálené objekty nemají tolik detailu a není třeba je stínovat po pixelech. Stačí výpočet provést na vrcholech a výsledky pak pouze interpolovat mezi pixely. Jak je naznačeno výše, k výpočtu osvětlení dochází vždy v okamžiku rasterizace objektu. Vykreslený výstup je nakonec zapsán do frame-bufferu a zobrazen na monitoru.



Obrázek 2: Na počátku máme 3D údaje o scéně (1). Na objektech je proveden výpočet osvětlení (2). Nakonec jsou 3D objekty rasterizovány a zobrazeny na monitoru (3).

Popsaný způsob funguje jak pro jedno, tak pro více světel najednou. Vzhledem k používání shaderů nejsme teoreticky nijak omezeni na počet takových světelných zdrojů. Limituje nás akorát výkon a schopnosti hardwaru, na kterém požadujeme korektní funkčnost. Shader, který se stará o výpočet osvětlení, naplníme informacemi o světelných zdrojích a při vykreslení tento shader zkombinuje osvětlení zpracovávaného vrcholu či pixelu všemi světly. Vzhledem ke schopnostem hardwaru ale velmi snadno dosáhneme limitu provedených instrukcí pro takový shader a proto musíme výpočet zjednodušit vynecháním některých světel, nebo výpočet provést na několikrát.

```
pro každý objekt:
      vykresli se všemi světly
```

Text 1: Pseudokód postupu vykreslování při aplikaci všech světel najednou.

```
pro každé světlo:
      vykresli celou scénu
```

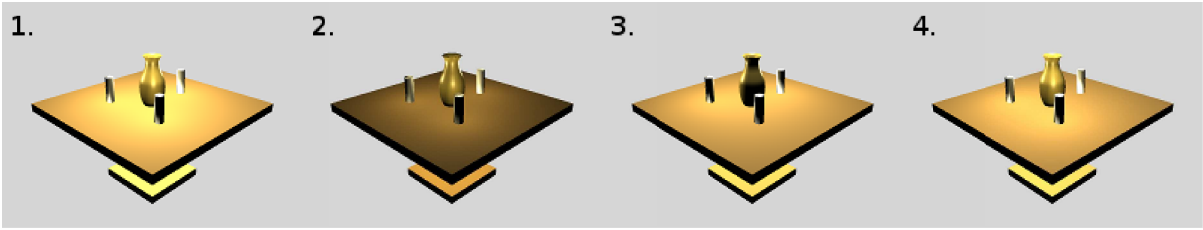
Text 2: Pseudokód postupu vykreslování při využití akumulačního bufferu.

Zjednodušení se provádí tak, že ze všech světel ve scéně vybereme ta, která zobrazovaný objekt nejvíce ovlivňují, a pak při rasterizaci objektu aplikujeme všechna vybraná světla najednou. Nejprimitivnějším způsobem výběru je zvolení několika nejbližších zdrojů světla. Hlavní nevýhodou je ale fakt, že velmi intenzivní zdroje světla mohou být potlačeny na úkor několika slabých zdrojů, které jsou blíž kreslenému objektu. Můžeme si představit scénu, kdy je na stole hliněná váza, kolem ní jsou ve vzdálenosti cca půl metru tři svíčky a jeden metr nad stolem visí ze stropu žárovka. Pokud se omezíme na tři světelné zdroje, pak jsou tyto zdroje naplněny informacemi o svíčkách a osvětlení žárovkou je ignorováno.

Můžeme tedy upravit ohodnocování světelných zdrojů a doplnit jej o intenzitu vyzářeného světla. Tím dosáhneme toho, že žárovka bude upřednostněna před svíčkami, přestože je od vázy dále. V tomto způsobu výběru světel je ale stále jeden důležitý nedostatek. Všechny svíčky jsou od vázy stejně daleko a nemáme žádnou kontrolu nad tím, která svíčka bude ignorována. Snadno se tak může

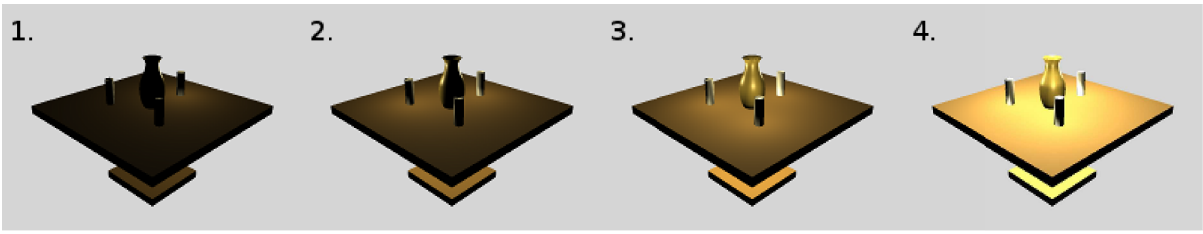
stát, že svíčka osvětlující čelní stranu vázy bude označena jako nejméně významná a její místo zabere svíčka, která osvětluje odvrácenou a tedy nezobrazovanou stranu vázy.

Finálním řešením je tedy stav, kdy prioritu světelného zdroje určujeme podle vzdálenosti od objektu, intenzity vyzářeného světla a vzájemné polohy světla, objektu a pozorovatele ve scéně.



Obrázek 3: Vlevo (1.) je referenční scéna a je osvětlena žárovkou i svíčkami. Ve druhé scéně zleva (2.) osvětlují stolek a vázu svíčky, žárovka je ignorována. Ve druhé scéně zprava (3.) je ignorována čelní svíčka. Ve scéně napravo (4.) je ignorována zadní svíčka.

Pokud výpočet osvětlení provádíme v několika krocích, využíváme akumulačního bufferu. Ten slouží k dočasnému uložení grafického výstupu a kombinování s nově vykreslovanými objekty. Pokud tedy máme velké množství světel, vykreslíme nejdříve scénu s první sadou světel a výsledek vložíme do tohoto bufferu. Poté scénu vykreslíme s další sadou světel a výsledek zkombinujeme s obsahem akumulačního bufferu. Takto postupujeme pro další a další světelné zdroje. Typicky se postupuje po jednom světle a scéna se tak vykresluje tolikrát, kolik je v ní světelných zdrojů. Pro ilustraci se opět přeneseme do scény se soškou a budeme postupovat po jednom světelném zdroji. Nejdříve sošku osvětlíme první svíčkou a do akumulačního bufferu vložíme výsledek. Máme v něm obraz s nasvícenou odvrácenou stranou sošky. Poté provedeme výpočet pro druhou svíčku. Výsledek sečteme s obsahem akumulačního bufferu a zapíšeme do něj výsledek součtu. Poté stejným způsobem provedeme výpočet i pro třetí svíčku a nakonec i pro žárovku. V akumulačním bufferu tedy akumulujeme veškeré osvětlení a až když je výpočet kompletně hotový, zapíšeme obsah akumulačního bufferu do frame-bufferu.



Obrázek 4: Ve scénách 1. až 4. jsou postupně akumulovány příspěvky od světelných zdrojů.

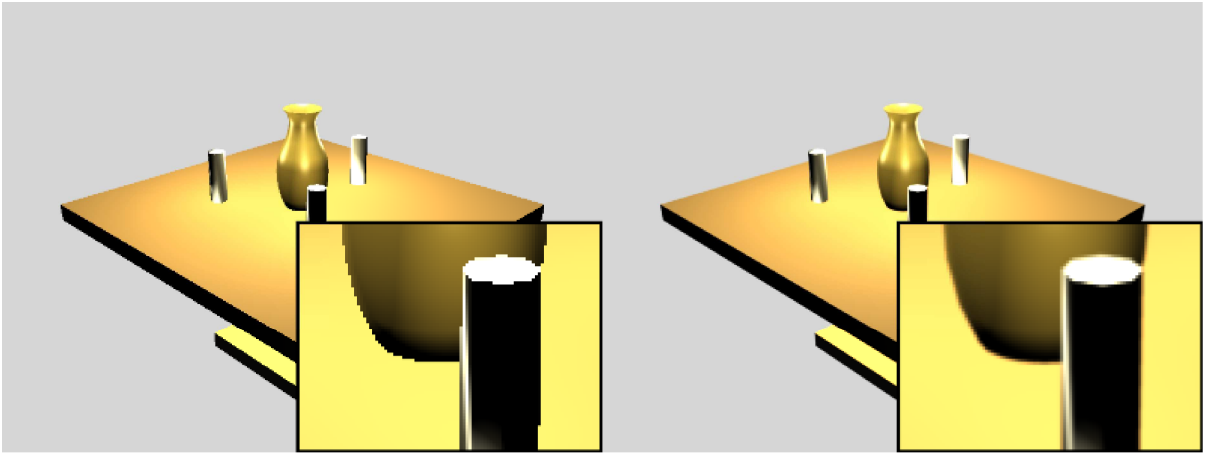
Zdálo by se, že řešení je tedy jednoduché a nic nebrání používání forward renderingu i nadále. S rozšiřujícími se možnostmi hardwaru se ale začala používat řada metod jak počítat osvětlení nejrůznějších materiálů, které nelze jednoduše parametrizovat. Vše je ještě komplikovanější pokud používáme více typů zdrojů světla. Standardně se používá bodové světlo (žárovka), směrové světlo (Slunce) a kuželové světlo (baterka). Pak je nutné mít shader pro výpočet osvětlení daného materiálu každým typem světla a to pro každý materiál. Pro takový případ se buď používá množství specificky naprogramovaných shaderů a při vykreslovaní se vždy zvolí ten odpovídající a nebo je použit jeden obecný shader (tzv. über-shader), který podle několika přepínačů provede výpočet pro daný materiál a daný zdroj světla.

Jak velké množství specifických shaderů, tak über-shader ale trpí závažnými nedostatky. Pokud použijeme více jednodušších shaderů, často se v nich nachází identické bloky kódu a pokud v něm nalezneme chybu, je třeba ji opravit ve všech shaderech. V případě über-shaderu se toto nestane, na rozdíl od jednoduchých shaderů je tu ale určitá režie způsobená obecností výpočtu a pokud chceme přidat další materiál, musíme výpočet jeho osvětlení provádět tak, aby respektoval parametry über-shaderu, případně upravit všechny ostatní materiály. V obou případech ale narůstá komplexita

enginu či shaderů a kvůli omezenému výkonu často omezujeme i počet světel na úkor vizuální kvality.

2.1.3 Výhody

Díky jednoduchému principu lze pro tento způsob vykreslování a osvětlování použít antialiasing. Grafické primitivum je během rasterizace vzorkováno několika body a výsledný pixel je smíchán ze současné barvy a barev těchto vzorků. Dojde tak k rozmazání ostrých přechodů a obraz působí jemněji. V současnosti existuje velké množství různých druhů vyhlazování, lišících se způsobem vybírání vzorků a situací, kdy je vhodné je použít.



Obrázek 5: V levé části je obrázek s aliasingem. Vpravo je zobrazena tatáž scéna s antialiasingovým filtrem. Je patrné vyhlazení ostrých hran mezi objekty.

Další neopomenutelnou vlastností je jednoduchá možnost používat pro kreslené objekty alpha-blending. Vzhledem k použití z-bufferu je ale v některých případech nutné seřadit vykreslovaná primitiva podle vzdálenosti od kamery a vykreslovat je od nejvzdálenějších po nejbližší. Případně se musí použít varianta míchacích faktorů, u které na pořadí vykreslování nezáleží.

2.1.4 Nedostatky

Z výše uvedených informací o forward shadingu lze odvodit řadu nepříjemných vlastností. S komplexními scénami v moderních hrách se tyto nevýhody projevují čím dál výrazněji. V současnosti totiž zobrazované scény dosahují značných velikostí a obsahují spoustu objektů. Narůstá i komplexita těchto modelů a je tak nutné rasterizovat čím dál více primitiv. Při jejich překrývání se plýtvá výkonem, protože se při rasterizaci zbytečně počítá osvětlení později překreslených primitiv. Navíc je dnes používáno mnoho světelných zdrojů a problém se tak projevuje ještě více, protože se celá scéna vykresluje několikrát a zrovna tak několikrát dochází ke stejně zbytečným výpočtům.

2.2 Deferred shading

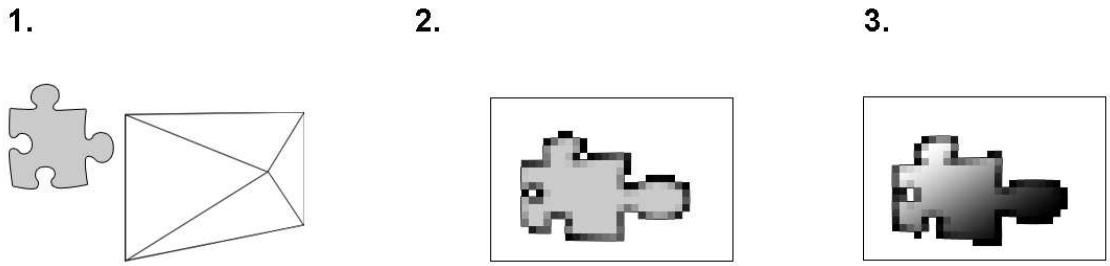
Deferred shading česky znamená odložené stínování. Někdy bývá nesprávně zaměňována s technikou deferred lighting, jejíž princip má s deferred shadingem stejný základ. Přesto se ale v některých ohledech liší a považují se tedy za dvě různé techniky.

Deferred shading není nijak nová metoda, jeden z prvních článků o tomto principu napsali Takafumi Saito a Tokiichiro Takahashi již v roce 1990 [2]. Až s novými schopnostmi a výkonem hardwaru byla tato metoda použita při renderování virtuálních scén v reálném čase. První hrou

využívající tuto metodu byl STALKER [8] a po jejím úspěchu se začal deferred shading více používat. Dnes se tento způsob výpočtu osvětlení stále více rozšiřuje jako primární metoda stínování a to nejenom ve hrách s pohledem z první osoby, ale i třeba ve strategiích s kamerou místěnou vysoko nad terénem. Příkladem budiž nedávno vydaná hra StarCraft 2.

2.2.1 Princip funkčnosti

Předpokladem je, že grafická karta umí renderovat obraz do textury. Taková textura se pak nazývá render-target (cíl vykreslování). Celá scéna je v prvním kroku vykreslena do tzv. Geometry-bufferu (zkráceně G-buffer), což je právě několik takových textur. Tak jako z-buffer, i tento buffer funguje po pixelech a každý pixel G-bufferu tak představuje část scény promítnuté na tento buffer. Rasterizace scény probíhá stejně jako u forward shadingu, ale neukládají se ostínované pixely, ale pouze informace pro další zpracování. V druhém kroku se počítá osvětlení, kdy se využívá informací v G-bufferu naplněného z předešlé fáze. Tento výpočet se provádí stejným způsobem jako post-processing efekty, detaily budou uvedeny později.



Obrázek 6: I zde na počátku vystupují 3D údaje o scéně (1). Objekty jsou rasterizovány a obraz je uložen v G-bufferu (2). Nakonec je proveden výpočet osvětlení za použití 2D dat (3).

Na rozdíl od forward shadingu se v tomto případě scéna rasterizuje pouze jednou. Navíc se při této rasterizaci pouze ukládají data o objektech scény a neprovádí se téměř žádné složité výpočty. Nedostatky spojené se z-bufferem se tedy projevují, ale výrazně méně. Pokud se zaměříme pouze na výpočet osvětlení, pak u forward shadingu jsme dosáhli polynomiální složitosti výpočtu. U deferred shadingu pouze několikrát osvětlujeme pixely z G-bufferu a výpočet má tedy lineární složitost.

Oproti klasickému stínování probíhá deferred shading výhradně po pixelech, což by v případě jednodušších scén mohlo být nevýhodou. Dnešní hry už ale osvětlení po vrcholech téměř nepoužívají. V naprosté většině z nich se totiž používají některé metody mapování virtuálních nerovností na vykreslované povrchy a u těchto metod je práce po pixelech nutná.

Z deferred shadingu tedy profitují především scény s velkým množstvím dynamických světelných zdrojů a s objekty, které je třeba stínovat po pixelech.

2.2.2 Geometry-buffer

G-buffer již podle názvu obsahuje především geometrické informace o scéně. 3D objekty scény jsou ve fázi rasterizace transformovány na 2D data a právě G-buffer je jimi naplněn. Mezi ukládané informace patří pozice ve scéně vůči kameře, normálový vektor povrchu a různé vlastnosti materiálu. Pro Phongův osvětlovací model to typicky bývá barva a lesklost povrchu.

Vzhledem k tomu, že těchto informací je poměrně mnoho, je velkou výhodou, že dnešní grafické karty umí použít více renderovacích cílů najednou (technologie zvaná MRT – multiple render-targets). Všechna potřebná data lze tedy uložit v jediném průchodu. Dřívější GPU uměla označit za cíl pouze jeden render-target a celá scéna musela být vykreslena několikrát. Potenciál výkonového nárůstu byl tak značně omezen.

G-buffer je nezbytným prvkem v osvětlení scény a zobrazení výsledného obrazu. Má proto rozměry odpovídající použitému rozlišení. Každou uloženou vlastnost potřebujeme uchovávat v co možná nejvhodnější přesnosti. Pro pozici tak potřebujeme tři 32 bitové floating point hodnoty. Pro normálový vektor postačí pouze 16 bitové floating point hodnoty. Barvu materiálu stačí reprezentovat standardním RGB formátem, tedy třemi bajty. Nakonec pro lesklost materiálu stačí jediná bajtová hodnota. To je dohromady 28 bajtů na jeden pixel obrazu. Při použití stále více se rozšiřujícího FullHD rozlišení 1920x1080 tak dostávámevelikost G-bufferu 45.619.200 bajtů. To se sice nezdá jako mnoho, je ale nutné vzít v úvahu, že v paměti grafické karty potřebujeme mít uložené i textury objektů a že tento G-buffer je relativně úsporný. V případě použití složitějšího osvětlovacího modelu jsou potřeba další parametry, které spotřebují další paměť navíc (např. barva světla emitovaného z povrchu, barva odlesku, barva odrazu u zrcadlících povrchů a další).

Grafické karty mají navíc další limitující faktor. Při použití více render-targetů najednou musí mít všechny tyto render-targety stejnou bitovou hloubku. U pozice nemůžeme přesnost obětovat, proto musíme změnit formát barvy a lesklosti materiálu. Pokud tedy použijeme tři 32 bitové floating point hodnoty pro každou složku (pozice, normála, barva i lesklost), dostáváme se již na 99.532.800 bajtů zabrané paměti na grafické kartě.

Je zřejmé, že u barvy a lesklosti materiálu zbytečně plýtváme pamětí. Proto se používají způsoby, jak některé hodnoty zakódovat či využít některých implicitních znalosti. Takové představil ve své práci Shawn Hargreaves [3]. Pozice ve scéně je určená vůči kameře a je daná třemi osami – X, Y a Z. Osy X a Y ale kopírují osy texturovacích souřadnic G-bufferu. Stačí nám tedy uložit pouze Z souřadnici. Pozice na osách X a Y jednoduše dopočítáme. Pokud známe vektor pohledu kamery, dopočítáme si vektory vedoucí z pozice kamery do všech čtyř rohů obrazu. Tyto vektory předáme do vertex shaderu a při rasterizaci dojde k jejich interpolaci a správnému určení vektoru pro každý pixel. Známe tedy vektor pohledu $\vec{v}$ ve zpracovávaném pixelu a souřadnici Z jeho pozice p_z. Rovnice pro výpočet zbylých souřadnic p_x a p_y jsou následující:

$$p_x = \frac{v_x}{v_z} \cdot p_z \tag{1}$$

$$p_y = \frac{v_y}{v_z} \cdot p_z \tag{2}$$

Další možností je ukládat si Z souřadnici po aplikování projekční matice. Dopočítat zpětně souřadnice X a Y je stejně triviální jako v předchozím případě. V tomto případě pro rekonstrukci nepotřebujeme počítat žádné vektory, vystačíme si s pozicí pixelu v obrazu a Z souřadnicí v G-bufferu. Pro výpočet pak použijeme inverzi projekční matice.

Ať už použijeme první nebo druhou optimalizaci, vždy nám zůstane uložená pouze hloubka. To je ale jen z-buffer. Pokud bychom tedy měli přístup k hardwarovému z-bufferu, dala by se pozice kompletně vyloučit. Tuto možnost ale zatím nemáme, souřadnici Z si tak musíme ukládat. Je to ale pouze jediná 32 bitová hodnota.

Při ukládání normály můžeme využít toho, že lze ze znalosti délky normálového vektoru a jeho dvou souřadnic dopočítat třetí. Do G-bufferu tedy budeme chtít ukládat normalizovaný normálový vektor a tím jeho délku známe jako konstantu. Dále využijeme toho, že všechna vykreslená primitiva jsou přivrácená ke kameře a souřadnice Z normálového vektoru má tak pořád stejné znaménko. Není tedy třeba ukládat Z souřadnici normály a dopočítáme ji ze souřadnic X a Y podle následující rovnice:

$$z = \pm\sqrt{1-(x^2+y^2)} \tag{3}$$

Pokud se omezíme na 16 bitovou hloubku pro hodnoty normály, dostáváme se na stejnou bitovou hloubku jako u pozice.

Pro barvu potřebujeme RGB formát a pro lesklost jeden bajt. Použijeme tedy RGBA a tím dostáváme posledních 32 bitů. Pokud takto optimalizujeme konstrukci G-bufferu, dostaneme se na 24.883.200 bajtů zabrané paměti, což je zhruba polovina oproti neoptimalizované verzi s libovolným formátem jednotlivých složek a dokonce čtvrtina oproti verzi, která by měla všechna data a ta by byla ve stejném formátu.

V praxi tedy musíme konstrukci G-bufferu věnovat velkou pozornost a především si ujasnit, jaký osvětlovací model budeme používat. Také se snažíme ukládat jen nezbytné atributy a ostatní se snažíme dopočítávat. Tím sice narůstá komplexita shaderu, který plní G-buffer, i shaderu, který poté musí atributy dopočítat a provést osvětlení, ale pokud se nejedná o příliš složité výpočty, je pro nás ušetřené místo větším plusem než obětovaný výkon.

2.2.3 Osvětlení scény

Výpočet osvětlení probíhá na stejném principu jako post-processing efekty. Vykreslí se obdélníkový polygon přes celou obrazovku (tzv. full-screen quad) a jako jeho textury se nastaví jednotlivé části G-bufferu. Při rasterizaci se v pixel shaderu načítají data a pro zpracovávaný pixel obrazu se vypočítá osvětlení.

Každý typ světelného zdroje má vlastní pixel shader a každý zdroj světla se zpracovává nezávisle. Pro každé světlo se tak vybere odpovídající pixel shader, naplní se informacemi o tomto světle (např. pozice ve scéně, barva a další) a poté dojde k výpočtu podle výše uvedeného postupu. Z toho logicky plyne, že výpočet závisí jen na počtu světel a rozlišení obrazovky. Složitost je tudíž lineární a má tedy větší potenciál než forward shading. Toho bývá často využíváno při návrhu scény.

Chystané scény mívají nějaké ambientní světlo, v případě zobrazování exteriérů sluneční či měsíční svit a pak velké množství lokálních světelných zdrojů, u kterých se právě projeví síla deferred shadingu. Takovými světly mohou být pouliční lampy, různé pochodně či světla automobilů, dokonce i jiskry tvořené částicovým systémem. Fantazii se meze nekladou a umělci vytvářející scénu tak mají ve své práci volnou ruku.

2.2.4 Významné vlastnosti

Deferred shading je v současnosti rozšiřující se technika pro stínování scény. Vzhledem k principu má nejvýše lineární složitost výpočtu a má proto dobrý potenciál pro zobrazování velmi komplexních scén v reálném čase.

Nevýhodou může být významná náročnost na paměť grafické karty. G-buffer musí obsahovat pro každý pixel obrazu všechny atributy nutné pro výpočet osvětlení a i při použití technik pro redukování počtu těchto atributů může být při složitém osvětlovacím modelu značně velký.

Dalším nedostatkem je nemožnost použít hardwarový antialiasing, protože grafické karty nepodporují vyhlazování u render-targetů. Antialiasing se tak musí řešit jako post-processing efekt, kdy se nejdříve aplikuje edge-detection filtr a podle něj se pak dalším filtrem rozmazávají nalezené hrany.

Nejdůležitější vadou na kráse je ale problém s blendingem. Už z principu je v G-bufferu uložena pouze informace o nejbližší vrstvě povrchů a cokoliv částečně průhledného naprosto překryje vše za sebou. Pro zobrazování skleněných povrchů lze použít G-buffer, který má několik vrstev. Tím se ale násobí problém s paměťovou náročností, protože každá vrstva musí mít možnost uchovávat informace o celém obrazu a stále jsme omezeni zvoleným počtem těchto vrstev. Jiným a často používaným řešením je použití deferred shadingu pouze pro neprůhledné objekty a forward shadingu pro objekty používajících blending (např. částicové systémy, voda s reflexí a refrakcí).

2.3 Raytracing

Raytracing není v oblasti počítačové grafiky žádnou novinkou a existuje dokonce několik variací tohoto způsobu renderování. U nejtypičtějšího z nich, raycastingu, se paprsky vrhají z kamery skrz každý pixel stínítka a hledá se část povrchu tělesa, se kterým paprsek koliduje. Pro tento kousek tělesa se spočítá osvětlení vržením paprsku z tohoto bodu do zdroje světla a výpočtem jejich vzájemných vztahů. Velmi snadno se dá dosáhnout reflexe, průhlednosti či výpočtu stínů. Jedná se o vrhání dalších a dalších paprsků a hledání nových kolizí s jinými tělesy. Refrakce je také jednoduchým prvkem, jelikož se při kolizi s povrchem pouze upraví směr paprsku podle fyzikálních zákonů. Raytracing dokáže velmi dobře simulovat řadu optických jevů, které se u rasterizace získávají více průchody, zjednodušováním či jiným šálením hráče.

Tato metoda syntézy obrazu velmi dobře škáluje s větší výpočetní silou. Každý paprsek je totiž unikátní a nesdílí s ostatními žádná data. Výpočty různých paprsků lze tedy provádět nezávisle a výpočet tak paralelizovat. Dá se říct, že když zdvojnásobíme počet pracujících procesorů, zdvojnásobí se nám i počet snímků, které za jednotku času získáme. A naopak, pokud dvakrát zvětšíme plochu obrazu, zdvojnásobíme tím i počet vržených paprsků a tedy množství práce pro procesory. Výkon je pak poloviční.

U raytracingu je nejnáročnějším výpočtem nalezení průsečíku paprsku s objektem resp. s trojúhelníkem. V herní scéně totiž bývá mnoho objektů složených z ještě více trojúhelníků. Pro každý paprsek musíme najít kolizi se všemi trojúhelníky a vybrat tu nejbližší. Tím se dostáváme na kvadratickou složitost.

Pro urychlení se používají metody dělení prostoru. Celý prostor scény se rozdělí na několik podprostorů a při hledání kolizí paprsků s objekty se nejdříve zjišťuje, zdali paprsek vůbec projde daným podprostorem. Pokud ne, lze tuto část scény kompletně vynechat. Pokud ano, pak se počítají kolize s objekty uvnitř tohoto podprostoru. Metod dělení je opět několik. Používají se pravidelné mřížky, různé varianty stromového dělení či kombinace. Každá s různou úrovní urychlení.

V dnešních hrách ale požadujeme velmi dynamické prostředí. Snadno si tak lze domyslet, že by bylo nutné v každém okamžiku zobrazování přebudovat rozdělení scény podle aktuálního rozmístění objektů, což v případě velmi se měnících scén vyžaduje spoustu výkonu.

Obrázek 7: Hra Enemy Territory vykreslovaná metodou raytracing v reálném čase. Obrázek vlevo zobrazuje komplexní scénu. Na obrázku vpravo je zobrazený pohled pod vodní hladinou. Je zde patrný totální odraz na rozhraní dvou prostředí s odlišným indexem lomu.

Přestože jsou dnešní procesory velmi výkonné, stále je raytracing velice náročnou metodou zobrazování scén. Inženýři ve firmě Intel již nějakou dobu experimentují s raytracingem v reálném čase a dokazují, že již dnes lze raytracing použít. V roce 2007 byla předvedena starší hra Quake 4 renderovaná raytracingem [10]. Za použití dvou čtyřjádrových procesorů hra běžela kolem 90 snímků za sekundu v rozlišení 1280x720. V roce 2009 pak představili raytracovaný Enemy Territory: Quake Wars [11]. Ten na čtyřech šestijádrových procesorech běžel na 20 až 35 snímků za sekundu při stejném rozlišení jako Quake 4. Nižší výkon i při použití většího počtu výkonnějších procesorů lze

přiřknout především náročnější scéně. Poslední novinkou z dílen Intelu je prezentace raytracované verze hry Wolfenstein z roku 2010 [12].

Podle dosavadních událostí je jasné, že raytracing bude možné použít i v komerčních hrách. Již při používání standardních procesorů lze zobrazovat v reálném čase portace relativně nových her. Pokud by byl uvedený akcelerátor podobný dnešním grafickým kartám, pravděpodobně by se raytracing mohl začít rozšiřovat. V Intelu pro tyto účely použili prototypy projektu Knights Ferry, který by takovým akcelerátorem mohl být.

Oproti rasterizaci jde o mnohem snadněji paralelizovatelný výpočet, dovoluje použití přesných odrazů a lomů světla, zrovna tak podporuje zobrazení ostrých stínů. Všechny tyto efekty lze použít i u rasterizace, ale za cenu značného poklesu výkonu a renderování scény více průchody. Zároveň je ale raytracing stále velmi pomalý a na urychlování se pořád pracuje. Přesto lze předpokládat, že díky kladům této techniky a díky zvyšování výkonu tato metoda zobrazování scény najde v budoucnosti uplatnění i v počítačových hrách.

3 Speciální efekty

Zobrazovaná scéna by nepůsobila příliš realisticky, pokud by se nevyužívalo různých speciálních efektů. Téměř v každé hře se můžeme setkat s nějakou formou částicového systému. Rovněž se ve většině her objevuje nějaká forma vrhaných stínů, i když jdou většinou zcela vypnout v rámci zvyšování framerate na pomalejších konfiguracích. Dále se na zobrazovanou scénu aplikují různé post-processing efekty, což jsou pouze úpravy 2D obrazu scény na základě nějakých dalších informací.

3.1 Částicové systémy

Některé efekty by bylo velice složité realizovat pomocí standardních způsobů vykreslování, protože jsou ve své podstatě simulací nějakého jevu probíhajícího na větším prostoru. Typickým příkladem je kouř, kdy se malé částečky spalin pohybují vzduchem a reagují na vítr, gravitaci a kolidují s ostatními objekty.

3.1.1 Základní princip

Na základě těchto a pár dalších vlastností můžeme každou částici popsat několika atributy. Nejčastěji mezi ně patří pozice v prostoru, rychlost pohybu, velikost, barva, hmotnost a životnost. Není nutné, aby částice měly všechny tyto vlastnosti a rovněž mohou být použity i vlastnosti jiné.

Efekt pak můžeme realizovat pomocí soustavy takových částic a simulací jejich pohybu v prostoru během uplynulého času. Celkový průběh simulace se řídí několika kroky:

- Odstranění částic, které přesáhly svou životnost
- Vygenerování nových částic
- Krok simulace
- Vykreslení částic

V případě, že do systému nepřibývají nové částice, mohou ty existující žít po nekonečně dlouhou dobu. Obvykle ale potřebujeme generovat stále nové částice. Abychom se neocitli v situaci, kdy se snažíme simulovat téměř nekonečné množství částic, musíme omezit jejich životnost. Pak dochází k tomu, že částice umírají a potřebujeme je ze systému odstranit a uvolnit tak místo pro nově vznikající částice.

Při generování částice jí přidělíme určité počáteční vlastnosti, které jsou určeny typem efektu, který má částice popisovat. Tyto vlastnosti jsou určeny pomocí nějakého vhodného náhodného rozložení pravděpodobnosti a tím zajistíme, že se každá částice chová trochu odlišně i přesto, že všechny mají společné rysy a respektují stejné podmínky. Dosáhneme tak dobrého stupně uvěřitelnosti simulace.

Krok simulace je pouhá aktualizace vlastností částice na základě jejich minulých hodnot. Používají se různé integrační metody a různě složité výpočty. Vždy záleží na požadavcích na přesnost a použitých vlastnostech částic. Vzhledem k tomu, že se částice v systému liší pouze aktuálními hodnotami, lze tento výpočet masivně paralelizovat. V současné době je tedy běžné, že se tento výpočet provádí na grafických kartách, které disponují velkým množstvím výpočetních jednotek.

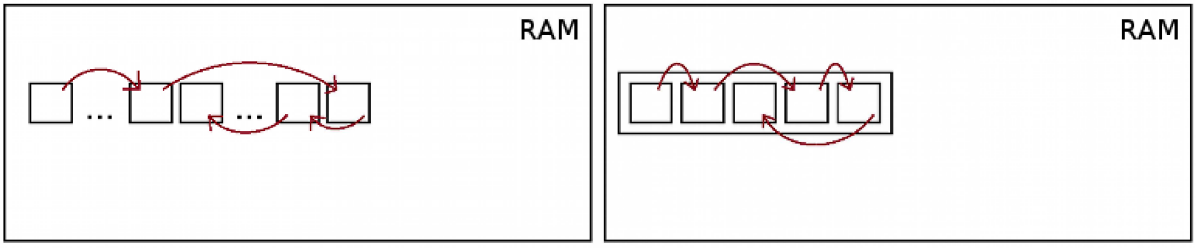
Na závěr potřebujeme částice vykreslit. Z podstaty částic se snažíme kreslit pouze body v prostoru. V minulosti, kdy nebylo možné akcelerovat výpočet na GPU, byly simulace omezené pouze na několik málo tisíc částic v systému. Pokud by tyto částice byly obyčejnými body, nevypadal by efekt kvůli jejich malému množství příliš dobře. Proto se pro částice začaly používat jiná grafická primitiva jako trojúhelník či obdélník (quad). Na ten se nanesla barevná textura a tím se aproximoval celý shluk částic. I v současné době se tento princip používá, protože je velice jednoduché simulovat

pohyb miliónů částic, ale není v silách grafických karet vykreslit celou scénu s takovým množstvím částic.

3.1.2 Možnosti implementace

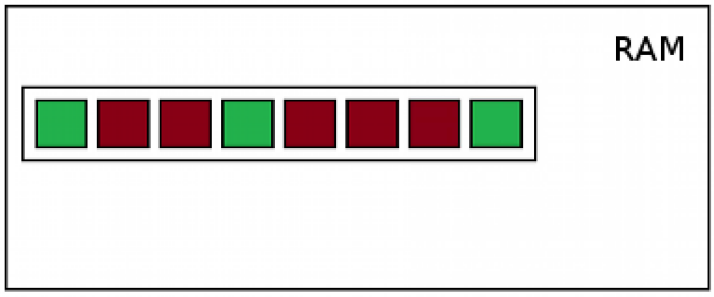
Pokud nám stačí malé množství částic, nemusíme se pouštět do varianty akcelerované pomocí GPU. I tak ale musíme dbát na dobrou stavbu datových struktur a volbu vhodných algoritmů.

Respektováním principu částicového systému, kdy jednotlivé částice vznikají a umírají v náhodný čas, se nabízí použití seznamu. Každý prvek seznamu ukazuje na další prvek v pořadí, při přidání nového prvku jej jednoduše vložíme na konec tohoto seznamu. Při odstraňování provážeme prvky v okolí odstraněného. V tomto případě ale velmi brzo narazíme na problém s výkonem. Prvky seznamu totiž mohou být roztroušeny na různých místech v paměti a často se tak do cache procesoru musí nahrát nový blok dat. I když použijeme pole prvků, které jen různým způsobem provazujeme, stále je výkon nedostatečný.



Obrázek 8: Vlevo je případ s částicemi uloženými různě po paměti. Vpravo jsou částice v poli, přesto však náhodně provázané.

Je tedy třeba zvolit jinou datovou strukturu pro uchovávání stavu částic, nebo omezit způsob generování a odstraňování částic. Pokud omezíme počet částic, můžeme pro uchování částic použít pole a částice rozšířit o atribut, zdali se jedná o aktivní částici. Pak sekvenčně procházíme prvky a počítáme pouze s aktivními částicemi. Neaktivní využíváme jako volné místo pro generování nových částic.



Obrázek 9: Zelené buňky označují volné místo, červené představují aktivní částice.

Pokud potřebujeme simulovat větší množství částic, vyplatí se přemýšlet nad akcelerací pomocí GPU. V takovém případě je použití seznamů nemožné. Nejen, že by se takový výpočet špatně prováděl, ale grafické karty neznají datový typ ukazatel a seznam je tak nerealizovatelný. Proto zde použijeme pole tak, jak je uvedeno v případě výše. Generování a odstraňování částic je ale sekvenční problém a proto se i nadále musí provádět na procesoru. Výpočet ale paralelizovat lze a proto jej přesuneme na GPU. Pole částic tak musíme pravidelně aktualizovat jak na straně procesoru (vznik a zánik částic), tak na straně grafické karty (aktualizace vlastností).

Na straně procesoru máme velké množství možností jak částice reprezentovat, ale u grafické karty tolik možností nemáme. Proto se implementace na straně procesoru musí přizpůsobit možnostem GPU. Zůstávají tak varianty, kdy výpočet provedeme pomocí vertex shaderů či pixel shaderů. V dřívějších dobách nebylo možné získat výstupní data z vertex shaderu a proto byly pro

udržování stavu částic diskvalifikovány. Sice byly možnosti, jak toto obejít úpravou výpočtu, ale více se přistupovalo k použití pixel shaderu. Dnes jsou však tyto varianty ekvivalentní a záleží tak na možnostech konkrétního API pro práci s grafickou kartou.

V případě použití pixel shaderu byly atributy ukládány do bufferů, které byly použity jako textury i render-targety, kde každý pixel bufferu odpovídal jedné částici. Na grafických kartách ale nelze z textury zároveň číst a zapisovat do ní a proto jsou tyto buffery zdvojené a v každém kroku simulace se přepne jejich úloha. Samotný výpočet probíhá stejným způsobem jako osvětlování v deferred shadingu. Jako render target se označí buffery obsahující atributy částic. Přes celou obrazovku se vykreslí quad se zdrojovými buffery atributů v podobě textur. Při rasterizaci se provede krok simulace a na obrazu tak dostáváme nový stav částic. Tato data pak při generování a odstraňování částic upravujeme na procesoru a využíváme v dalším kroku výpočtu.

Při vykreslování samotných částic pak byly kromě barevné textury použity i buffery obsahující aktuální data a pomocí odpovídajících texturovacích souřadnic se pro částici načetly korektní informace.

3.2 Stíny

Nepostradatelnou součástí her jsou dnes i vržené stíny. Přidáním stínů do scény často velmi výrazně přispějeme k realističnosti scény. Navíc pozorovateli dodáme další informaci o vzájemné poloze objektů. Během uplynulých let se používaly různé metody vrhání stínů a v průběhu vývoje se volily stále komplexnější a realističtější metody pro tvorbu stínů. Nejjednodušší formou je vrhání stínů na rovinné objekty. Pokročilejšími metodami jsou shadow-volumes a nakonec shadow-mapping.

3.2.1 Projekční stíny

První metodou je promítání stínů na rovinnou plochu. Zde se objekt vrhající stín zploští, naškáluje a umístí pomocí speciální transformační matice. Tento zploštělý objekt se poté vyrenderuje a díky provedené transformaci je promítnut na rovinu, na které má stín ležet.

Pokud se tato metoda implementuje naivním způsobem, může docházet k z-fightingu. Tento problém souvisí se z-bufferem a principem, na kterém funguje. Pokud dva objekty leží na stejném místě, měl by ten později vykreslený překrýt dřívější. Často se ale stane, že se fragmenty z obou objektů prolínají a obraz je pak zničený. Výsledný stín je potrhaný a při pohybu kamery se třese. Problém můžeme vyřešit mírným posunutím zploštělého objektu nad dopadovou rovinu či zakázáním testování hloubky při vykreslování stínu tak, jak je popsáno v [1].

Pokud chceme tuto metodu použít pro scénu s více objekty, vykreslíme nejdříve rovinu, na kterou budeme stín promítat. Poté vykreslíme zploštělé verze objektů vrhajících stíny se zakázaným osvětlením. Nakonec vykreslíme objekty scény. Pokud pro zabránění z-fightingu použijeme zákaz testování hloubky, je toto pořadí velmi důležité. Pokud bychom jej nedodrželi, mohly by stíny překreslit i objekty, které právě tyto stíny vrhají. V případě posunování stínů nad rovinu dopadu na pořadí nezáleží, ale tento posuv musíme volit opatrně. Pokud objekt leží na rovině dopadu, mohlo by se stát, že stín vykreslíme nad tento objekt a částečně jej zakryjeme jeho vlastním stínem.

Tato metoda je velmi jednoduchá, ale pro moderní hry se složitými scénami se nehodí, protože potřebujeme zobrazovat stíny na nejrůznějších podkladech. Navíc tato metoda už z principu neumožňuje tzv. self-shadowing, kdy objekt vrhá stín sám na sebe. Proto se musíme poohlédnout po komplexnějším způsobu získání stínů.

3.2.2 Shadow-volumes

Mnohem pokročilejší metodou jsou shadow-volumes. Pro výpočet stínů se používají modely reprezentující stín v prostoru, tedy stínová tělesa. Tato tělesa tvoří jakousi obálku vrženého stínu

a počátek mají na obrysu objektu, který stín vrhá. Při vykreslování scény a stínů v ní pak používáme informaci vytvořenou pomocí těchto stínových těles.

Samotné stínové těleso vytvoříme tak, že nejdříve nalezneme siluetu objektu. Tu tvoří hrany sdílené mezi přivrácenými a odvrácenými trojúhelníky. Poté tuto siluetu protáhneme směrem od světla až k nekonečnu.

Pro vykreslení scény se používá několik variant této metody. Všechny varianty ale využívají stencil-buffer. To je dvourozměrný obraz s rozměry výstupního obrazu, který obsahuje určité celočíselné informace. Při renderování objektů je pak možné upravovat informace ve stencil-bufferu na pozici pixelů rasterizovaného objektu. Při renderování dalších objektů lze tyto informace využít k maskování a vykreslit tak pouze část objektu. Stencil-buffer tak lze přirovnat k šabloně, přes kterou nastříkáme obraz na zeď.

Z-pass

Při vykreslování scény včetně stínů pomocí varianty z-pass nejdříve celou scénu vykreslíme tak, jako by byla kompletně ve stínu. Poté naplníme stencil-buffer podle následujícího postupu:

- Vyprázdníme stencil-buffer a zakážeme zápis do z-bufferu.
- Zapneme back-face culling, čímž zakážeme renderování odvrácených polygonů.
- Operaci pro úpravu stencil-bufferu nastavíme na inkrementování při z-pass.
- Vykreslíme stínová tělesa a naplníme tak stencil-buffer.
- Přepneme se z back-face cullingu na front-face culling (tedy zakážeme renderování přivrácených polygonů a povolíme renderování těch odvrácených)
- Operaci pro úpravu stencil-bufferu nyní nastavíme na dekrementování při z-pass.
- Znovu vykreslíme stínová tělesa.

Po tomto procesu vykreslíme celou scénu se zapnutým osvětlením a stencil-buffer použijeme jako masku. Oblasti označeny nulou budou vykresleny, ostatní se vymaskují a zůstanou tak zastíněné.

Nedostatkem této varianty je situace, kdy se kamera ocitne uvnitř stínového tělesa. Nedojde tak ke korektnímu naplnění stencil-bufferu, protože stínovému tělesu, ve kterém se kamera nachází, chybí stěna, která by těleso uzavřela. Celý stencil-buffer má tak odchylku +-1 podle konkrétní situace. Tuto chybu lze vyřešit spočítáním uzávěru a jeho přidáním ke stínovému tělesu. Tento výpočet je ale velmi náročný a proto se přistupuje k jinému řešení. Tím může být detekování, zdali je kamera uvnitř nějakého stínového tělesa a úpravou hodnot stencil-bufferu odpovídající tomuto tělesu. Detekce polohy kamery může být taktéž velmi náročným výpočtem.

Z-fail

Východiskem ze situace s kamerou uvnitř stínového tělesa je robustnější varianta z-fail. Postupuje se podobně jako u z-pass. Opět nejdříve vykreslíme zastíněnou scénu. Stencil-buffer plníme následujícím postupem:

- Vyprázdníme stencil-buffer a zakážeme zápis do z-bufferu.
- Zapneme front-face culling, čímž zakážeme renderování přivrácených polygonů.
- Operaci pro úpravu stencil-bufferu nastavíme na inkrementování při z-fail.
- Vykreslíme stínová tělesa a naplníme tak stencil-buffer.
- Přepneme se z front-face cullingu na back-face culling (tedy zakážeme renderování odvrácených polygonů a povolíme renderování těch přivrácených)
- Operaci pro úpravu stencil-bufferu nyní nastavíme na dekrementování při z-fail.
- Znovu vykreslíme stínová tělesa.

Nepočítáme tedy stínová tělesa nacházející se před vykreslovanými objekty, ale za nimi. Výsledek je stejný jako u předchozí varianty. Díky tomu, že nepočítáme stínová tělesa mezi kamerou a objekty se ale elegantně zbavíme problému zmizení či invertování stínů v situaci, kdy se kamera nachází uvnitř stínového tělesa. Na druhé straně je ale z-fail metoda náročnější. Situací, kdy z-test selže a tedy bude třeba upravit stencil-buffer, je typicky více.

Shrnutí

Technika stínových těles umožňuje zobrazení dynamických stínů na libovolném povrchu. Stíny jsou dokonale ostré a navíc objekt vrhá stíny i sám na sebe. Díky hardwarové podpoře stencil-bufferu je tato metoda i poměrně rychlá a ve své době byla i oblíbená a používaná (např. v enginu pro Doom 3).

Na druhé straně má ale první varianta v některých situacích problémy, které se jen obtížně řeší. Druhá varianta, která je vůči těmto problémům celkem odolná, je ale patentovaná a její použití se musí licencovat. Také je náročnější než varianta z-pass. Přistupuje se tedy ke kombinaci, kdy se implicitně používá z-pass metoda a jen v kritických situacích se použije z-fail.

Další nevýhodou je náročnost metody, protože je potřeba rasterizovat stínová tělesa, která často zasahují do velké části scény. Naráží se tak na výkonnostní limit fill-rate grafických karet, tedy schopnosti vykreslit určitý počet pixelů za sekundu. Při použití více zdrojů světla, pro které stíny počítáme, se tento problém násobí.

Posledním výrazným nedostatkem je náročnost vytváření samotných stínových těles. Zobrazované objekty jsou totiž čím dál složitější a s tím narůstá i složitost vytvářených stínových těles. Zde se naráží na výpočetní limity procesoru, který není schopný v rozumném čase spočítat stínová tělesa.

Vzhledem k nedostatkům, které by se u moderních her projevovaly čím dál více, se od této metody upouští ve prospěch shadow-mappingu.

3.2.3 Shadow-mapping

Posledním a dnes nejrozšířenějším způsobem zobrazování vržených stínů je shadow-mapping. Princip této metody spočívá v tom, že se na scénu podíváme z pohledu světla. Každý zobrazený pixel scény má od zdroje světla určitou vzdálenost a představuje počátek stínu vrženého právě tímto pixelem. Pixel viditelný z pohledu světla je tedy osvětlený a cokoliv vzdálenějšího od světla než tento pixel je zastíněno. Princip viditelnosti za nás řeší z-buffer a je pro tyto účely využíván.

Při vykreslování scény nejdříve vykreslíme scénu neosvětlenou. Poté si vykreslíme scénu z pohledu světla a uložíme si informace ze z-bufferu, čímž získáme tzv. shadow-mapu. Nakonec vykreslujeme osvětlenou scénu. Při vykreslování pak na úrovni pixelů porovnáváme hodnotu z uložené shadow-mapy a spočtené vzdálenosti rasterizovaného pixelu od světla.

Už z principu je zřejmé, že lze tuto metodu použít pro zobrazování dynamických stínů na libovolném povrchu. Základní varianta této metody ale trpí aliasingem. To z důvodu omezené velikosti shadow-mapy a také omezené přesnosti z-bufferu. Navíc jsou stíny tvořeny pouze v jednom směru a pokud potřebujeme stíny vrhané bodovým zdrojem světla, musíme vytvořit více shadow-map pro pokrytí celého prostoru. I přes tyto nedostatky se metoda používá, protože různými úpravami základního principu lze tyto nedostatky obejít. Navíc tato metoda umožňuje poměrně jednoduché vytváření měkkých stínů.

Aliasing

Vznikající aliasing lze nejjednodušeji řešit zvýšením velikosti shadow-mapy. Tím ale využíváme více paměti grafické karty a pokud je světlo i se stínícím objektem daleko, aliasingu se nevyhneme. Navíc rozměry nemůžeme zvětšovat donekonečna.

Dalším možným řešením je upravením principu. Způsobů existuje velké množství, lze jmenovat například Cascade shadow-mapping [13], Light-space perspective shadow-mapping [15], Perspective shadow-mapping či Trapezoid shadow-mapping [14]. Každá varianta řeší problém aliasingu jiným způsobem a výběr vhodné metody záleží na situaci.

Všesměrová světla

Jak již bylo napsáno, metoda shadow-mappingu je primárně určená pro světla založená na principu kuželového světla s úhlem rozsahu menším než 180°. Idea je vcelku jednoduchá. Potřebujeme

minimalizovat počet shadow-map, pokrýt celé pole viditelnosti světla a zachovat pokud možno rovnoměrné rozložení přesnosti shadow-mapy pro celou scénu. Stačí nám tak použít některou metodu parametrizace projekce ze 3D do 2D používaných například pro environment-mapping, kdy se na plochu nanáší obraz prostředí obklopující určité místo ve scéně. Takových způsobů je několik, ale pro shadow-mapping nejsou vhodné všechny.

Setkáváme se tak s řešením, kdy použijeme tzv. cube-mapping. Zde do místa zdroje světla vložíme krychli a na každou stěnu promítneme příslušnou část okolního prostředí. Zachytíme veškerý prostor scény a kvalita shadow-map zůstává stejná jako u směrového světla. Počet shadow-map je ale vysoký, protože pro světlo dostáváme šest textur a vykreslovat scénu šestkrát je příliš náročné.

Lepším řešením je dle [16] použití paraboloidu, respektive dvou paraboloidů přiložených k sobě. Na každý zobrazíme celý poloprostor a pro pokrytí celého prostoru okolo světla stačí shadow-mapy pouze dvě. Ušetříme tak výkon díky mnohem menšímu počtu průchodů i přesto, že režie pro mapování ze standardní projekce na projekci na paraboloid není zcela zanedbatelná.

Měkké stíny

Shadow-mapping poskytuje poměrně jednoduché řešení zobrazení měkkých stínů. S dokonale ostrými stíny, které poskytuje třeba raytracing či u rasterizace metoda stínových těles, se v realitě nikdy nesetkáme, protože žádný zdroj světla nemá povahu nekonečně malého bodu. Určité rozmazání stínu v jeho okrajích se tak vyskytuje vždy. Měkké stíny se také objevují u plošných zdrojů světel a v rámci zvyšování uvěřitelnosti v zobrazování scén je třeba použít nějaký způsob, jak měkké stíny alespoň aproximovat.

Možností je posouvat zdroj světla kolem jeho pozice a vytvořit tak několik vržených stínů, které se nakonec zkombinují. Tento způsob je ale velice náročný, protože výpočet stínů je náročný sám o sobě a v tomto případě jej provedeme několikrát. Pro věrohodné výsledky je navíc potřeba velké množství těchto podružných stínů.

Je tedy vhodné použít přístup, který opět staví na ošálení pozorovatele, ale dodává dostatečný pocit reálnosti. Jedna z takových technik, zvaná Percentage-Closer soft shadows, je popsána v [17]. Díky programovatelnosti moderních grafických karet metoda není odkázána na pouhé porovnání hloubky z shadow-mapy a vzdálenosti rasterovaného pixelu, ale může chytrým způsobem prohledat i okolí a určit tak úroveň zastínění tohoto pixelu. Přesnost metody závisí na velikosti prohledávaného okolí a počtu vzorků, které se z něj vybírají, a závisí tak na umělcích, jaké parametry nastaví a jak bude výsledný stín vypadat.

3.3 Post-processing

Dalším druhem efektů jsou různé post-processingové úpravy obrazu. Zpracovává se výstupní obraz renderu scény a pracuje se tak pouze s 2D obrázkem. Nejsme ale omezeni pouze na barevný výstup, během renderování scény si lze uložit řadu dalších informací, které se pak při aplikaci různých efektů použijí.

Všech efektů je dosaženo pomocí dříve zmíněných render-targetů. Scénu vykreslíme do jednoho z nich, pak vykreslíme full-screen quad a render-target použijeme jako texturu. V pixel shaderu pak realizujeme samotný efekt, který načítá data z textury, zpracovává je a vytváří výsledný obraz.

3.3.1 Příklady různých efektů

Bloom efekt

Příkladem efektu, který používá pouze barevnou informaci o scéně, je bloom. Jedná se o přesvícení oblastí s vysokým jasem a simuluje se tak jakási záře kolem velmi jasných částí obrazu.

Tento efekt je proveden v několika průchodech. V prvním je snížen jas celého obrazu, takže oblasti s nízkou hodnotou jasu jsou staženy na nulu a oblasti s vysokým jasem jsou mírně redukovány. Tento obraz je poté zvlášť horizontálně a vertikálně (ne však nutně v tomto pořadí) rozmazán. Virtuálně tak obraz rozmažeme v obou směrech zároveň a ušetříme trochu výkonu. Tento rozmazaný obraz nakonec aditivně smícháme s původním barevným obrazem. Často se ještě před snižováním jasu celý obraz zmenší a s původním barevným obrazem se míchá tento zmenšený obraz. Šetří se tak další výkon a navíc je tím dosaženo dalšího rozmazání.

Obrázek 10: Postup vytváření bloom efektu. Vlevo je původní scéna, napravo od ní horní propust. Dále je obraz rozmazán a nakonec je sloučen s originálem.

Edge-detection filtr

Dalším důležitým efektem je hledání hran v obraze. Pro tento účel lze použít i barevnou informaci o scéně. Tento efekt ale většinou chceme za účelem získání dojmu kresleného obrazu a chceme zvýraznit pouze obrysy objektu. Mnohem lepších výsledků pak dosáhneme při použití hloubky scény či normálových vektorů. Samotný edge-detection filtr pracuje na známých principech, například Sobelův operátor či pouze diference mezi sousedními pixely. Takto detekované hrany u barevného obrazu pouze ztmavíme.

Tento efekt lze provést již v jediném průchodu, ale na druhé straně potřebujeme ze scény získat informaci o normálách a o hloubkách. Je zřejmé, že v případě použití deferred shadingu získáme tyto informace zdarma, protože jsou součástí G-bufferu, ale v případě forward shadingu musíme scénu vykreslit znovu za účelem získání těchto hodnot.

Obrázek 11: Postup zvýraznění hran. Vlevo jsou detekované hrany podle normály a hloubky. Vpravo pak obraz se ztmavenými hranami.

U deferred shadingu je tento efekt základním kamenem pro realizaci antialiasingu. Po detekci hran je na oblasti pod hranami aplikováno rozmazání a základní antialiasing je tak úspěšně vytvořen.

Sun-shafts

Tento efekt se na první pohled jeví jako velmi složitý a málokoho napadne, že jej lze řešit v post-processing fázi. Jediným potřebným atributem je hloubka scény. Tak jako u předchozího je deferred shading výhodou.

Obrázek 12: Zcela vlevo je hloubka scény. Uprostřed je na hloubku aplikováno rozmazání ve směru šíření světla. Vpravo je pak výsledek složený z rozmazané hloubky a barev scény.

Pro samotný výpočet ještě potřebujeme znát pozici světla, které má vrhat paprsky. Tuto pozici transformujeme do prostoru obrazu a poté provedeme rozmazání hloubky scény ve směru od tohoto bodu k pixelu, který se zpracovává. Tím dosáhneme kruhového rozmazání. Tento výsledný obraz už jen patřičně utlumíme a aditivně smícháme s barevným obrazem scény.

Distortion

Výsledný obraz můžeme různým způsobem deformovat. K tomu se využívá buď goniometrických funkcí nebo nějaké šumové funkce (Perlinův šum). Podle hodnoty příslušející zpracovávanému pixelu se upraví texturovací souřadnice pro tento pixel a při mapování je pak obraz zdeformován.

Tento efekt není příliš často používán pro deformování celého obrazu, často jej ale lze s úspěchem použít pro simulaci ohybu světla procházejícím rozhraním mezi horkým a studeným vzduchem nad hořícím ohněm. V takovém případě je ale nutné během renderování scény vytvořit masku, kde má k deformaci dojít. Při simulaci ohně tak tuto masku upravujeme během renderování částicového systému reprezentujícího oheň.

Aby iluze lomu světla vypadala realističtěji, je zdeformována i maska. Světlo se pak láme i kolem plamenů a ne jenom přímo na nich. Při výpočtu efektu pak použijeme šumovou funkci závislou na čase a upravujeme pouze texturovací souřadnice těch pixelů, které spadají do deformované masky. Tím je obraz zdeformovaný pouze v oblasti kolem plamenů.

Barevné korekce

Různé korekce barev, jasu či kontrastu jsou velmi používané a do post-processing efektů samozřejmě patří. Pokud chtějí tvůrci dosáhnout ponuré atmosféry, snižují sytost barev. V některých vojenských hrách může být hráč vybaven noktovizorem, pak se celý obraz zabarvuje do zelena. Některých úprav by se dalo dosáhnout změnou textur, ale často je třeba obraz pouze jemně doladit a úprava všech textur a nasvícení je příliš pracná.

3.3.2 Shrnutí

Existuje spousta dalších efektů. Některé se snaží simulovat přehrávání starých filmů a přidávají šum a různé rozmazání či deformace. Jako post-processing se dělají i například zatmívačky v různých in-game video sekvencích. Existují i postupy, jakými navodit dojem, že celý obraz je namalovaný vodovými barvami, či nakreslený perem. Efektů je obrovské množství a možnosti jsou téměř neomezené.

Většina efektů si vystačí pouze s barevnou složkou, ale některé velmi zajímavé efekty potřebují znát normály či hloubky ve scéně. U forward shadingu je pro jejich získání nutný další průchod scénou, případně musí být upraveny všechny shadery pro ukládání těchto informací během renderování scény. U deferred shadingu tyto atributy dostáváme zdarma díky principu, na jakém funguje. Osvětlení scény totiž není v principu nic jiného než další zajímavý post-processing efekt aplikovaný na několik vstupních obrazů.

4 XNA

XNA vzniklo jako následovník projektu Managed DirectX (MDX), které mělo poskytovat běžně používané principy herních enginů v jednoduché formě. Příliš se ale neujalo a jeho místo tak obsadilo novější XNA. S MDX má společný především svůj cíl usnadnit vývoj herních projektů.

XNA je založeno na .NET Frameworku a rozšiřuje jej o řadu struktur a funkcí, které se objevují ve většině herních enginů. Využívá funkcionality DirectX 9.0c a je možné jej používat na Windows XP, Windows Vista, Windows 7, Windows Phone 7 a X-Box 360. To vše s žádnými nebo jen minimálními úpravami kódu aplikace. Primárně je určené pro práci s jazykem C#, ale neoficiálně jej lze používat i s jinými .NET vyhovujícími jazyky.

XNA Framework vhodným způsobem zapouzdřuje low-level funkcionalitu DirectX a poskytuje programátorům dostatečnou míru abstrakce a také jim umožňuje věnovat se důležitějším aspektům hry, protože samo řeší rozdíly mezi platformami.

Základní třídou při tvorbě aplikace v XNA je Game, která inicializuje vše potřebné pro běh hry. Tedy vytvoří a nainicializuje se grafické zařízení, načte se grafický obsah a spustí se nekonečná smyčka zajišťující soustavný běh aplikace. Ten probíhá ve dvou oddělených částech, Update a Draw. Při zavolání Update se na základě uběhnutého času počítají animace a různé další změny ve hře. Draw pak vykreslí veškerý obsah. Tyto metody automaticky volají Update a Draw pro zaregistrované komponenty, které mohou být rovněž vytvořeny programátorem. Lze je využít pro vytvoření grafu scény a nechat pak na XNA jejich pravidelné aktualizace a renderování. V případě, že je třeba sdílet některé prvky v různých částech kódu (například grafické zařízení), lze s výhodou využít takzvané Service. Do nich se registrují instance objektů implementující určité rozhraní a lze je pak podle tohoto rozhraní identifikovat a používat skrze třídu Game. Pokud je aplikace nastavovaná určitou sadou parametrů, nemusíme uvnitř kódu do všech různých míst předávat referenci na vlastní objekt, ale tento objekt zaregistrujeme jako Service a ihned je přístupný ve všech komponentách hry.

Další automaticky řízenou částí je Content Pipeline, která se stará o načítání a udržování obsahu. Hry bývají z velké části obsahem definovány, protože obsahem jsou veškeré objekty, textury, efekty a další data, která nakonec ve hře vidíme. I když je podstatou hry její princip, na obsahu velice záleží. Existuje ale velké množství různých formátů a efektivní načítání a správa obsahu tak představuje poměrně velkou výzvu. Je totiž třeba rozpoznat obsah a formát, v jakém je tento obsah uložený. Podle toho se zvolí importer, který je schopný data extrahovat. Poté je třeba tato surová data zpracovat a transformovat pro korektní použití uvnitř enginu. XNA celý tento proces řeší a opět tak umožňuje soustředit se na hru samotnou místo řešení podpůrných procesů. Content Pipeline je také možné rozšiřovat a přidat si tak například vlastní formát pro uložení některého typu dat. Přitom není nutné omezovat se jen na modely či zvuky, můžeme mít uložené informace o kompletní scéně. Zvolený soubor je načtený odpovídajícím Importerem. Tato data jsou pak uložena v Content DOM. Processor tato data zpracuje a vytvoří z nich požadovaný objekt. Tento objekt je poté uložen v jednotném formátu datového souboru XNA, který je nakonec použit za běhu v samotné hře. Processor může například zpracovat data z výškové mapy a vytvořit mřížku použitelnou pro model terénu. Záleží jen na nás a našich potřebách, jak naložíme s načtenými daty.

Přístup k různým oblastem je velmi intuitivní a přestože je low-level funkcionalita zapouzdřena, stále je možné ovládat velkou většinu probíhajících procesů. Snadno lze řídit různé možnosti vykreslování, ozvučení či uživatelského vstupu. XNA je velmi dobře vyváženým a mocným nástrojem pro rychlou a snadnou tvorbu her pro platformy postavených na technologiích firmy Microsoft.

5 Návrh

Tato kapitola se zabývá návrhem architektury pro vykreslování komplexní scény pomocí dříve zmíněných metod deferred shading a forward shading. V návrhu bude brán zřetel na možnost použití post-processing efektů na scénu. Také se předpokládá použití částicových systémů. Návrh tedy bude obsahovat i funkcionalitu nutnou pro jejich výpočet a pro případ jejich masivního nasazení bude tento výpočet navržen pro akceleraci pomocí moderních GPU. Celá architektura bude navrhována pro použití v XNA a bude využívat jeho schopností. V návrhu nebude zahrnut graf scény a tedy ani související optimalizace vykreslování založené na dělení prostoru.

5.1 Požadavky

Na návrh jsou kladeny různé požadavky. Především jde o fakt, že výsledná aplikace je týmovým projektem a je tedy nutné návrh vytvořit dostatečně obecně, aby jej bylo možné i nadále upravovat podle aktuálních potřeb.

Plánované scény budou obsahovat velké množství objektů, které budou v drtivé většině neprůhledné či budou mít zcela průhledné části. Alpha-blending se na objektech scény nebude téměř vyskytovat.

Rovněž je v plánu masivní využití částicových systémů a je tedy vyžadován plynulý běh i při obrovském množství částic. Tyto částice budou alpha-blending vyžadovat a renderer na toto musí být připraven. Některé částice mohou sloužit i jako lokální všesměrové zdroje světla.

Z předchozího požadavku plyne, že ve scéně se bude nacházet velké množství světel. Technika vykreslování tedy musí být schopna efektivně osvětlovat složitou scénu množstvím světelných zdrojů.

Post-processing efekty nejsou vyžadovány, ale na základě závěru z ostatních požadavků bude v návrhu zahrnuta možnost jejich využití.

5.2 Renderer

Vzhledem k požadavkům na osvětlení bude výhodné využít metodu deferred shadingu jako primární osvětlovací metodu. Kvůli použití částicového systému bude ale nutné vytvořit v návrhu prostor pro použití standardního forward shadingu a sloučit výstupy obou metod. Kvůli použití deferred shadingu a možnému budoucímu požadavku na řešení aliasingu bude ale nutné navrhnout i systém pro použití post-processing efektů.

5.2.1 Vykreslování

Deferred shading bude primární metodou osvětlování scény a proto bude rendering všech ostatních částí řízen odsud. Označme tedy vytvářený renderer jako DeferredShadingRenderer.

Proces vykreslování scény bude probíhat následujícím postupem:

- Provede se vyprázdnění G-bufferu.
- Vykreslí se veškeré neprůhledné objekty či objekty se zcela průhlednými částmi a tím se naplní G-buffer.
- Mimo renderer se aktualizuje seznam všech světel.
- Provede se osvětlení scény připravenými světly. Výstup bude směrován do připraveného bufferu.

- Prostor k vykreslení částic. Výstup bude směrován do téhož bufferu jako předchozí krok a dojde tak k automatickému sloučení obrazů obou použitých metod.
- Podle nastavení rendereru se provedou post-processing efekty využívající informace uložené v G-bufferu a barevného výstupu obsaženého v bufferu použitém v předchozích dvou krocích.

G-buffer bude obsahovat informace o pozici, normálový vektor, barvu materiálu a lesklost materiálu. Všechny tyto atributy budou v G-bufferu uloženy s respektováním omezené velikosti grafické paměti a některé hodnoty atributů nebudou ukládány, ale budou dopočítávány při osvětlování. Podoba G-bufferu bude tedy následující:

- 1x 32fp hodnota – hloubka
- 2x 16fp hodnota – X a Y souřadnice normálového vektoru
- 1x RGBA hodnota – RGB barva materiálu a lesklost materiálu

Plnění G-bufferu bude potřeba realizovat různými shadery, aby bylo možné používat různé materiály pro objekty. DeferredShadingRenderer bude tyto základní shadery implicitně načítat a poskytovat skrze své rozhraní.

Osvětlování bude prováděno pomocí několika druhů světel. V prvé řadě to bude ambientní světlo, které konstantně nasvítí celou scénu. Dále to budou lokální bodová světla, kuželová světla a nakonec také směrová světla. Pro každý druh světla tak bude vytvořen speciální pixel-shader, kterému budou dodány informace o světelném zdroji. Světelné zdroje budou popsány několika parametry, z nichž některé budou společné. Většina se ale bude lišit podle typu světla.

Ambientní	*Směrové*	*Bodové*	*Kuželové*
Barva	*Barva* *Směr šíření*	*Barva* *Pozice ve scéně* *Konstantní útlum* *Lineární útlum* *Kvadratický útlum*	*Barva* *Pozice ve scéně* *Směr šíření* *Konstantní útlum* *Lineární útlum* *Kvadratický útlum* *Úhel světelného kuželu* *Zaostření světla*

Tabulka 1: Druhy světelných zdrojů a jejich parametry.

Pro provedení osvětlení bude vytvořen obdélníkový polygon pro vykreslení přes celou obrazovku. Tento polygon bude využíván i pro aplikaci post-processing efektů.

Metodou forward shading budou vykreslované pouze částice a případně také objekty, které budou využívat alpha-blending. S jejich nasazením se ale příliš nepočítá. XNA využívá DirectX 9.0c a bez úprav je tak tato metoda implicitně používána. V návrhu je vyhrazen prostor pro použití této metody a není třeba se věnovat detailnějšímu návrhu.

5.2.2 Post-processing

Post-processing bude poslední fází renderingu. Bude využívat hloubky a normály uložené v G-bufferu a externě uložené barevné složky scény.

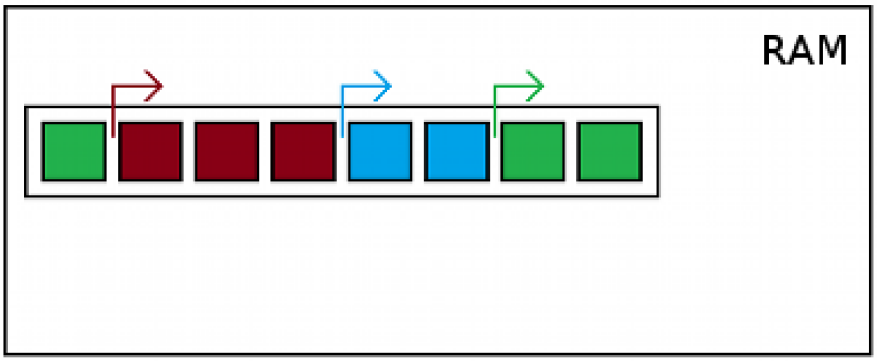
Každý efekt bude využívat schopnosti XNA provádět víceprůchodové shadery a každá část post-processing efektu tak bude implementovaná jako jeden shader program. Pro aplikaci efektu bude použit obdélníkový polygon.

Pro efektivní aplikaci efektů budou buffery použité pro post-processing zdvojené a podle potřeby se mezi nimi bude rychle přepínat.

5.3 Částicové systémy

Na částice bude kladen velký důraz a bude požadován vysoký výkon pro jejich výpočet. Proto bude výpočet probíhat na GPU. Různé částicové systémy budou oddělené, efekt kouře tak bude probíhat nezávisle na efektu ohně a podobně.

Každý systém bude obsahovat buffery s uloženými atributy částic. Ty se budou při práci na procesoru jevit jako fronty. Generování a zanikání částic bude probíhat na CPU a pro optimální výpočet budou mít všechny částice v jednom systému stejnou životnost. Fronty tak budou rozděleny na tři jasně ohraničené části, kdy v první části jsou aktivní částice, v další části jsou nově vygenerované částice a ve zbytku jsou volné pozice ve frontě. Díky stejné životnosti budou částice zanikat ve stejném pořadí, v jakém jsou uloženy ve frontě. Nedojde tak k situaci, kdy se musí komplikovaným a pomalým výpočtem zjišťovat první volná pozice či stejně komplikovaným výpočtem určovat, které aktivní částice zrovna zanikly. Uvěřitelnost simulace by měla být dostatečně zajištěna velkým množstvím částic a náhodností ostatních atributů.



Obrázek 13: Fronta částic. Červené jsou aktivní částice, modře označené jsou nově vygenerované částice, zelené buňky jsou volné.

Celá simulace bude probíhat podle následujícího postupu:

- Emitory vytvoří nové částice.
- Částice se přidají do seznamů nových částic odpovídajících systémů.
- Po naakumulování nových částic budou tyto hromadně přidány do bufferů.
- Provede se výpočet jednoho kroku simulace.
- Z bufferu obsahujícího dobu života se aktualizuje pozice první aktivní částice.
- Vykreslí se všechny aktivní částice, počínaje první aktivní částicí zjištěnou v předešlém kroku.

Použitými atributy budou pozice, rychlost pohybu a doba života. Velikost částic bude určena použitým systémem a případně životností. V některých případech totiž budou částice měnit svou velikost v závislosti na době života. To samé platí o průhlednosti. Částice nebudou mít barvu, ale bude na ně nanesena textura, která bude opět společná pro všechny částice v systému.

Buffery s atributy budou pro GPU texturami a pro výpočet simulace bude vykreslován obdélníkový polygon s aplikovaným odpovídajícím shaderem, který bude napsán pro konkrétní systém. Částice různých systémů se tak budou chovat odlišně a nebude třeba tyto rozdíly ukládat ve formě dalších atributů částic. Tak jako u post-processing efektů i tyto buffery budou zdvojené, kdy jeden buffer z páru bude použit jako textura se zdrojovými daty a druhý buffer bude použit jako render-target pro uložení nově vypočtených dat.

Díky simulaci prováděné na GPU lze předpokládat, že bude možné počítat velké množství částic a úzkým hrdlem tak bude nejspíše vykreslování částic. Tomu se pokusíme zabránit

instancingem. Při obyčejném renderování se vykresluje vždy po jednom objektu. Nachystají se transformační matice, shader, textury a zavolá se funkce na vykreslení. V případě instancingu se nachystá celé pole transformačních matic odpovídajících jednotlivým instancím stejného objektu, shader a textury a jediným voláním vykreslovací funkce se vyrenderuje určitý počet instancí objektu. Díky dodaným transformačním maticím se každá instance vykreslí na správném místě.

V této implementaci ale nebudeme chystat celé matice, protože částic bude velké množství a každá matice je složena z šestnácti hodnot. Do grafické karty bychom museli předávat velké množství dat, které by trvalo dlouho vytvořit. Namísto toho využijeme bufferu s uloženými pozicemi a každé instanci předáme pouze texturovací koordináty do tohoto bufferu. V shaderu pro vykreslování se pro každou instanci načte její pozice, z ní se vytvoří transformační matice a ta se nakonec použije pro vykreslení.

6 Implementace

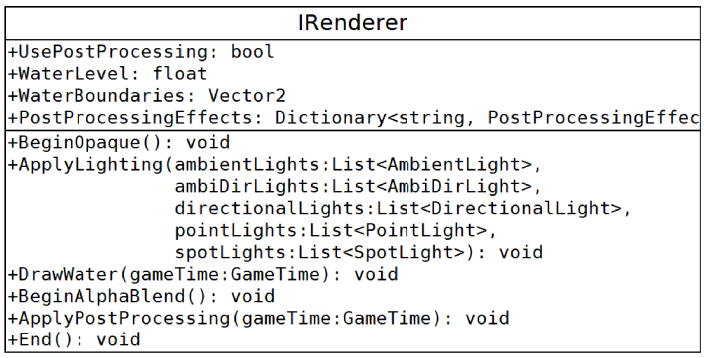
Tato kapitola bude věnována implementačním detailům všech navržených součástí a jejich integraci do hry. Nejprve bude rozebrána implementace vykreslování pomocí technik deferred a forward shading a problémy, které se objevily v souvislosti s použitím XNA frameworku. V další části je popsán způsob, jakým byly implementovány post-processing efekty. Rovněž jsou zde krátce popsány efekty použité ve hře. V předposlední části je rozebrána implementace částicových systémů za použití hardwarové akcelerace výpočtu na GPU. V závěru je pak pospáno, jak byly všechny části integrovány do hry a jakým způsobem jsou používány.

6.1 Renderování

Podle původních požadavků bylo v návrhu specifikováno, že hlavní metodou pro vykreslování bude deferred shading. Renderer obstarávající tuto funkčnost byl implementován nejdříve. Avšak kvůli celkové náročnosti herních scén a tedy i nízkému výkonu, byly požadavky omezeny. Omezení se týkalo především počtu světel, konkrétně se počet zredukoval na pouhý jeden směrový světelný zdroj – sluneční svit. Dále se změnil požadavek na způsob výpočtu osvětlení, kdy bylo u mnoha objektů upřednostněno osvětlení po vrcholech. Na druhé straně bylo do požadavků přidáno alespoň jednoduché vykreslování vodních ploch.

Z výše uvedených důvodů byl deferred shading vyloučen jako primární renderovací technika a byl naimplementován jednoduchý forward shading s jediným světlem. Deferred shading v aplikaci zůstal a je tedy možné přepínat mezi technikami vykreslování.

Kvůli jednoduššímu a transparentnímu použití obou rendererů byl navržen společný interface nazvaný `IRenderer`, který pak oba renderery implementují.



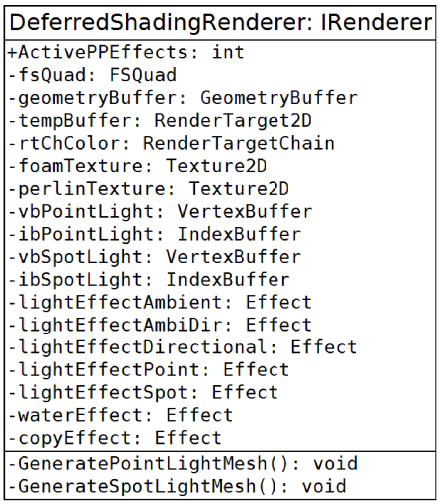
Obrázek 14: Atributy a metody společného interfacu `IRenderer`.

Dalším důsledkem použití dvou vykreslovacích metod jsou úpravy shaderů pro vykreslování objektů. Použité shadery tedy obsahují techniky jak pro forward shading, tak pro deferred shading. U forward shadingu se v shaderech provádí veškeré výpočty včetně osvětlení. U techniky pro deferred shading se v shaderu počítají výstupní data pro Geometry-buffer.

Díky dodržení interfacu nebylo potřeba radikálnějších změn ve zbylém kódu hry a například použití částic nebylo ovlivněno vůbec. Post-processing byl zahrnut jako součást rendererů a přestože jsou u každé varianty rozdíly, princip a většina kódu pro aplikaci post-processing efektů jsou identické.

6.1.1 Deferred shading renderer

Renderer zodpovědný za korektní vykreslování metodou deferred shading je reprezentovaný třídou `DeferredShadingRenderer` a implementuje výše popsané rozhraní `Irenderer`. Kromě toho má spoustu dalších atributů či pomocných metod.



Obrázek 15: Atributy a metody třídy `DeferredShadingRenderer`.

Geometry-buffer

V návrhu je sice zahrnuta specifikace atributů G-bufferu, ale pro účely hry bylo nutné ukládat jeden atribut navíc – intenzitu odlesků. Bez tohoto atributu se totiž objekty ve scéně velmi leskly a jen pomocí lesklosti materiálu nebylo možné odlesky řádně kontrolovat. G-buffer tedy obsahuje pozici, normálový vektor, intenzitu odlesku, lesklost materiálu a barvu materiálu. Aby G-bufferu nepřibyl další render-target, byla snížena přesnost normál a jeho přesná podoba je tedy následující:

- 1x 32fp hodnota – hloubka
- 1x RGBA hodnota – X, Y a Z souřadnice normálového vektoru, intenzita odlesků
- 1x RGBA hodnota – RGB barva materiálu a lesklost materiálu

V programu je G-buffer reprezentován třídou `GeometryBuffer`. V konstruktoru se pouze specifikují jeho rozměry. Jednotlivé render-targety pro atributy jsou vytvořeny právě zde a jsou přístupné přes veřejné metody `get`.

Osvětlení a optimalizace

Pro výpočet osvětlení jsou vytvořeny speciální shadery. Při aplikaci osvětlení se procházejí jednotlivé světelné zdroje a podle typu se vybere příslušný shader. Podle základního principu stačí vždy vykreslit full-screen quad s odpovídajícím shaderem. Tento způsob lze vylepšit.

Pouze ambientní a směrové světlo vždy zasáhnou celou scénu a projeví se tak na celé obrazovce. Bodová a kuželová světla jsou ale omezena útlumem. Logicky tedy bodové světlo osvítí jen prostor mezi svou pozicí a body ve vzdálenosti, kdy je již intenzita vyzařovaného světla utlumena. Tento prostor má tvar koule. U kuželového světla je situace obdobná, pouze se jedná o tvar připomínající kužel. Přesněji jde o průnik koule definované vzdáleností, do které má světlo nějaký vliv, a kuželu s vrcholem na pozici světelného zdroje a s nekonečnou výškou. Nadále však budeme pro jednoduchost mluvit o kuželu. Pokud jsou tyto zdroje dostatečně daleko od kamery, projeví se pouze na relativně malé ploše obrazovky a ušetří se tak značná část výkonu, protože na nepokryté části obrazovky nedochází k žádným zbytečným výpočtům.

Jsou tedy implementovány dvě pomocné metody: `GeneratePointLightMesh(...)` a `GenerateSpotLightMesh(...)`. Tyto dvě metody procedurálně vytvoří modely koule a kuželu a uloží je do indexbufferů a vertexbufferů. Při aplikaci osvětlení bodovým či kuželovým světlem se pak namísto full-screen quadu vybere jiný model a vykreslí se s shaderem odpovídajícím typu zdroje.

Tyto dva modely jsou vytvořeny pouze jednou a nijak se v programu neupravují. Různé bodové resp. kuželové zdroje ale mohou mít různé parametry. Do shaderu jsou tyto informace předávány, protože jsou nezbytné pro korektní výpočet osvětlení. V shaderu tak lze provést i nezbytnou úpravu modelu. Podle útlumu se spočítá vzdálenost, kdy je intenzita dopadajícího světla již zanedbatelná. Tato vzdálenost je použita pro celkové naškálování modelu. Model kuželu je nadále upraven podle úhlu světelného kuželu.

Shadery počítající osvětlení využívají různé atributy G-bufferu. Ambientní světlo nezpůsobuje žádné odlesky, proto je pro tento světelný zdroj relevantní pouze barva materiálu. Pro ostatní zdroje světla jsou však potřeba všechny atributy. Z hloubky se pomocí inverze projekční matice spočítá pozice pixelu v prostoru kamery. Normála je reprezentována v RGB složkách, které nabývají pouze kladných hodnot. Proto jsou normály zakódovány stejně jako bývají v normal-mapách. Je tedy nutné je přepočítat z navzorkovaných hodnot n_s na skutečné normály pixelu n_p podle následujícího vzorce:

$$n_f = n_s * 2 - 1 \tag{4}$$

Dále je použito podobně jednoduché kódování pro lesklost materiálu. Spekulární složka osvětlení se totiž počítá podle vzorce:

$$L_S = L_C \cdot S_I \cdot (\vec{C} \times \vec{R})^{S_S} \tag{5}$$

L_S je intenzita spekulární složky světla dopadající na povrch pixelu, L_C je barva světla, S_I je intenzita odlesku daná materiálem, $\vec{C}$ představuje vektor směřující od kamery k pozici pixelu obrazu. $\vec{R}$ pak představuje odražený vektor a S_S lesklost materiálu.

V tomto případě by hodnoty lesklosti v rozmezí od nuly do jedné neměly smysl – materiál by byl příliš lesklý. Spíše se hodí lesklost tlumit a hodnota lesklosti je tedy přepočítána podle následujícího vzorce:

$$S_S = sample_S \cdot 255 + 1 \tag{6}$$

S_S opět představuje lesklost materiálu. $sample_S$ pak představuje hodnotu načtenou z G-bufferu.

Ostatní hodnoty již kódovány nejsou a jsou použity přímo. Pro výpočet osvětlení je pak použit následující vzorec:

$$C = C_D \cdot L_D + L_S \tag{7}$$

C je výsledná barva pixelu, C_D je barva materiálu, L_D představuje intenzitu difůzní složky světla a L_S intenzitu spekulární složky světla.

Spekulární složka je počítána podle výše uvedeného vzorce (5) a difůzní podle vzorce (8).

$$L_D = \vec{L} \times \vec{N} \tag{8}$$

L_D představuje intenzitu difůzní složky světla, $\vec{L}$ je vektor směřující od pozice pixelu ke světlu, $\vec{N}$ je normálový vektor pixelu.

Spočtené osvětlení je ukládáno a akumulováno a celkový výsledek je pak použit při další činnosti rendereru.

Princip fungování rendereru

Celý proces vykreslování začíná metodou `BeginOpaque(...)`. V ní se jako cíle vykreslování nastaví G-buffer. Buffer se vyprázdní a poté může začít samotné vykreslování neprůhledných objektů.

Po vykreslení objektů je G-buffer naplněn daty, která jsou potřebná k výpočtu osvětlení. Voláním metody `ApplyLighting(...)` se toto provede postupem popsaným výše.

V této chvíli je nutné si uvědomit zásadní věc ve funkčnosti XNA. Z-buffer i stencil-buffer jsou totiž vázané na použitý render-target a pokud chceme hloubku či informace ze stencil-bufferu využít i později, musíme poté znovu kreslit do toho samého render-targetu. Což znamená, že v případě deferred shadingu musíme objekty kreslené po aplikaci osvětlení znovu renderovat do G-bufferu, což v případě implementované aplikace znamená render-target obsahující barvu a lesklost materiálu. Vzhledem k tomu, že se G-buffer používá pouze pro aplikaci osvětlení a v pozdějších fázích již tato data nejsou potřebná, můžeme toto provést.

Při výpočtu osvětlení tedy potřebujeme využívat data z G-bufferu, ale zároveň potřebujeme zapisovat do jednoho z render-targetů, jež je součástí G-bufferu. Toto ale na současném hardwaru není možné a osvětlení tak musíme ukládat do dočasného bufferu. V aplikaci je pro tento účel vytvořen tempBuffer. Po výpočtu osvětlení je pak obsah tohoto bufferu zkopírován do G-bufferu a můžeme pak i nadále využívat hardwarového z-bufferu i stencil-bufferu. Toto je sice funkční řešení, ale spíše by se od tvůrců XNA hodilo, kdyby v tomto ohledu umožnili více flexibility. Propojení z-bufferu a stencil-bufferu s render-targetem je totiž poměrně svazující a nutí programátora plýtvat výkonem na kopírování.

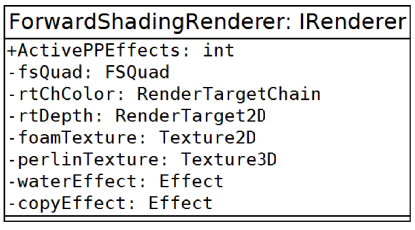
V každém případě ale díky předešlému postupu můžeme vykreslovat i nadále. Cílem renderování je stále G-buffer, který ale v tuto chvíli ztrácí svůj původní význam. Pokud je to vyžadováno, pak je v tomto okamžiku metodou `DrawWater(...)` vykreslena voda. Detaily o vykreslování vodních ploch budou popsány později.

V dalším kroku je metodou `BeginAlphaBlend(...)` zahájeno vykreslování objektů, které využívají alpha-blending. Poté jsou tyto objekty vykresleny a celý postup se tak chýlí ke konci.

V závěru jsou aplikovány post-processing efekty metodou `ApplyPostProcessing(...)`. Detaily budou opět uvedeny později. Na úplný konec se volá metoda `End(...)`, která uzavře celý proces renderování. Její význam se projeví v případě, kdy byl zakázaný post-processing. Ten je totiž implementovaný tak, že při aplikaci posledního efektu se vykresluje do back-bufferu. Pokud ale post-processing neproběhl, máme scénu stále jenom v G-bufferu. Metoda `End(...)` tedy v případě nutnosti překopíruje data do back-bufferu.

6.1.2 Forward shading renderer

Forward shading renderer byl vytvořen až dodatečně po zjištění, že hra je poměrně náročná na výkon grafické karty. S tímto vědomím se tento renderer i implementoval a jedná se tedy o jednoduchou variantu forward shadingu, kdy je použito pouze jedno světlo. Není tak třeba řešit žádné priority mezi světelnými zdroji a skládání mezivýsledků pomocí akumulačního bufferu.



Obrázek 16: Atributy třídy `ForwardShadingRenderer`.

Podobně jako u předchozího případu, i zde renderer představovaný třídou `ForwardShadingRenderer` implementuje stanovené rozhraní `IRenderer`. Zde je však metoda `ApplyLighting(...)` bez jakékoliv funkčnosti, protože veškeré osvětlení se počítá už při vykreslování.

Osvětlení

Na rozdíl od deferred shadingu, zde není osvětlení jednotné. V rámci šetření výkonu se u některých objektů použilo osvětlování po vrcholech a pouze u vybraných objektů se provádí výpočet na úrovni pixelů. V obou případech je ale výpočet prováděn stejným způsobem jako u deferred shadingu a i zde tedy platí vzorce 5, 6, 7 a 8.

Princip funkčnosti

I zde je celý proces zahájen metodou `BeginOpaque(...)`. Pokud je zakázaný post-processing i vykreslování vody, pak se vykresluje přímo do back-bufferu. V opačném případě je ale postup komplikovanější a musí se využívat dalších bufferů. Neplýtvá se pamětí grafické karty a pro tento účel je využit buffer pro provádění post-processing efektů.

V dalším okamžiku jsou vykresleny objekty bez alpha-blendingu a zde je proveden i výpočet osvětlení. Volání metody `ApplyLighting(...)` tedy ponechá vše beze změn.

Dalším krokem je kreslení vody metodou `DrawWater(...)`. Nato je zahájeno vykreslování objektů s alpha-blendingem metodou `BeginAlphaBlend(...)`. Opět platí dříve zmíněná spojitost z-bufferu a stencil-bufferu s render-targetem a vykreslování těchto objektů se musí provádět do stejného bufferu jako vykreslování neprůhledných objektů.

Na samý závěr jsou aplikovány post-processing efekty, k čemuž opět slouží metoda `ApplyPostProcessing(...)`, a opět je renderování uzavřeno voláním metody `End(...)`. Pro obě metody platí ta samá pravidla jako u `DeferredShadingRendereru`.

6.1.3 Vykreslování vodních ploch

Ve hře bylo plánováno, že vodní plochy budou hrát poměrně důležitou roli v logice herní mechaniky. Přestože k tomu nakonec nedošlo, vykreslování vodních ploch bylo implementováno a voda je použita pro zvýšení vizuální kvality. Toto vykreslování mají na starost renderery a jak u deferred shadingu, tak u forward shadingu funguje naprosto stejně.

Vykreslování vody je ve hrách poměrně rozsáhlé téma a volba vhodné metody vždy záleží na konkrétní situaci. V akčních hrách, kdy se lze ke všemu přiblížit na vzdálenost několika centimetrů či se lze dokonce dostat pod vodní hladinu, je potřeba velice kvalitní zpracování. Často se tak řeší vlnění vody, kdy se používají různé šumové funkce či dokonce Fourierova transformace. Toto vlnění se pak aplikuje na model představující vodní plochu a lze tak pozorovat jak se voda valí na břeh a za krátkou chvíli opět zmizí.

Obrázek 17: Zobrazení vodní hladiny ve hře Crysis 2

Dále je důležitá věrohodná simulace optických jevů na rozhraní vody a okolního prostředí. Mezi tyto jevy patří reflexe a refrakce a výběr metody opět závisí na situaci. V dnešních hrách se již poměrně často setkáváme s iluzí rozkladu světla u refrakce, kdy lze pozorovat barevnou duhu na hranách objektů. Zrovna tak je několik možností jak realisticky zobrazit odraz na hladině. Zde lze použít například environment-mapping či se objekty nad hladinou vykreslí na místě, kde by měl být zobrazen jejich odraz a výsledek se ořízne pomocí stencil-bufferu.

Ve výsledné aplikaci je voda velmi zjednodušená. Refrakce je implementována poměrně naivním způsobem a reflexe je zanedbána úplně. I přesto, že výsledek není matematicky přesný a bylo jej dosaženo pouze opakovaným testováním a laděním různých parametrů, vypadá velice přesvědčivě.

Vlnění

Model vody je pro jednoduchost pouhou rovinou a je reprezentován dvěma trojúhelníky, které tvoří obdélník zadaných rozměrů. Veškeré vlnění je tak pouze iluzí, která vzniká díky výpočtu normálových vektorů na pixelech této roviny.

K výpočtu vlnění byl použit Perlinův šum. Ten byl nejdříve počítán za běhu aplikace v pixel shaderu. Po přidání dalších částí výpočtu však nastal problém s překladem shaderu, kdy XNA hlásilo problém s počtem instrukcí. Z toho důvodu byl naprogramován externí generátor Perlinova šumu, který generuje 3D texturu. Tato textura má rozměry 256x256x256, což je maximum, které XNA dovolí. Textura je pak v aplikaci načtena a předána do shaderu, který je použit pro vykreslování vody. V něm už se pouze načítají a dále zpracovávají data. Výpočet tak byl významně urychlen a bylo možné přidat i další efekty na vodní hladině.

Generátor je postaven na XNA a využívá pixel shaderu pro generování textury. Programově se zadají rozměry textury a při spuštění pak dojde k vygenerování bezešvé textury ve všech třech rozměrech. V každém snímku se generuje jedna vrstva textury, která je pak uložena do souboru ve formátu PNG. Tyto obrázky je pak nutné libovolným nástrojem převést na 3D texturu ve formátu DDS, který je XNA schopné načíst.

Samotné vlnění je pak vytvořeno pomocí několika vzorků z textury s šumem. První sada vzorků texturu vzorkuje zhruba s šestinásobnou frekvencí než druhá sada. Pro lepší dojem z hladiny vody byly tyto sady rozpohybovány v závislosti na čase v protichůdném směru. Tato dvě vlnění jsou pak ováhována a sečtena, čímž dají jedno výsledné vlnění.

Výsledné vlnění reprezentuje výškovou mapu pro celou plochu vody. Z těchto výšek jsou v pixel shaderu spočítány normálové vektory, které se nadále používají pro výpočet osvětlení a také reflexi a refrakci.

Refrakce a reflexe

Jak bylo napsáno výše, reflexe je zanedbána a je tak reprezentována pouze konstantní barvou, která evokuje odraz oblohy na vodní hladině.

Refrakce naproti tomu byla zapracována do shader programu. Pro vytvoření refrakce však byl potřeba podklad, tedy obraz objektů pod vodní hladinou. K tomuto účelu byl použit dočasný buffer, který u deferred shadingu fungoval jako cíl pro výpočet osvětlení a z kterého byla data následně zkopírována zpět do G-bufferu. U forward shadingu takový buffer ale není a proto se v metodě `DrawWater(...)` vytváří a naplní se kopií aktuálního obrazu.

Samotná refrakce je pak založená na úpravě texturovacích souřadnic podle normálového vektoru v pixelu hladiny vody. Tyto texturovací souřadnice se pak použijí pro vzorkování podkladu. Je tak simulován raytracing, kdy se vyšle paprsek z kamery skrz pixel na hladině vody a ten dopadá na podklad. Použitý způsob není matematicky přesný a zanedbává například ovlivnění hloubkou vody, do které paprsek dopadá. Zrovna tak nelze zobrazit cokoliv, co je zakryté již v podkladu díky používání z-bufferu. Pro ošálení pozorovatele to však stačí, protože vodní hladina je díky výše popsanému vlnění po celý čas v pohybu.

Zabarvení

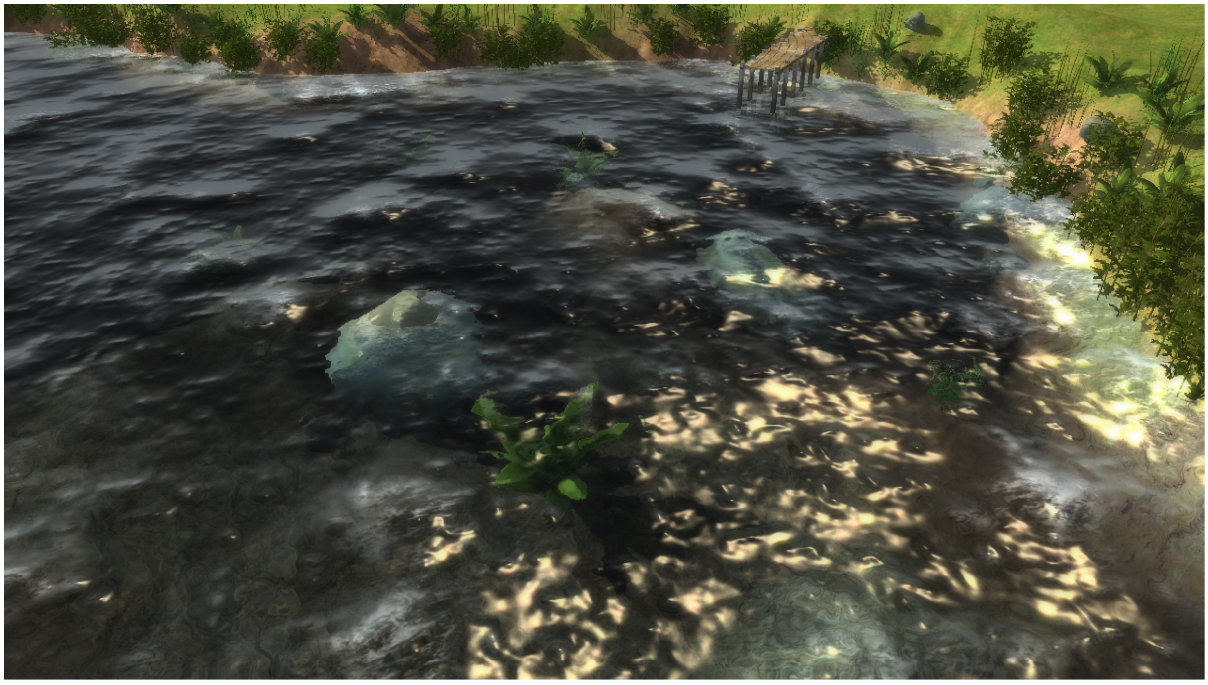
Celkové zbarvení vody je složené z několika částí. V první řadě se využívá z-bufferu, resp. jeho kopie, která u deferred shadingu vzniká ukládáním hloubky pro zpětnou rekonstrukci pozice. U forward shadingu je tohoto dosaženo ukládáním hloubky během rasterizace objektů. Při vzorkování podkladu se vzorkuje i hloubka a podle ní se pak ovlivňuje i zabarvení vodou.

Pro tento účel jsou v shaderu definovány tři barvy. Výsledná barva zobrazená na hladině vody je lineární interpolací mezi barvou podkladu a barvami vody. Váhy interpolace jsou tak jako parametry vlnění určeny experimentálně a jsou závislé na hloubce. Podklad ve větší hloubce je tak tmavší než podklad na mělčině.

Do zabarvení je zahrnuto i vytváření pěny na hladině. Tato pěna je načtena jako další textura a podobně jako vlnění je i pěna rozpohybována. Není ale zobrazena po celé ploše vody, ale pouze v oblastech, kde vlny dosahují svého vrcholu nebo kde je voda velmi mělká a je tedy pravděpodobné, že vlny narážejí do nějakého pevného objektu.

Výsledek

Přestože je simulace optických jevů na hladině vody velmi zjednodušená, je složená z několika částí, kdy každá i přes svou nedokonalost přidává na věrohodnosti a celkový výsledek tak vypadá zdařile. Zde se více než kdekoliv jinde v aplikaci projevuje, že lidské oko lze ošálit i jednoduchou aproximací.



Obrázek 18: Voda implementovaná ve hře. Lze si všimnout odlesků světla, odrazu oblohy, lomu světla či pěny u kamenů.

6.2 Post-processing efekty

Oba renderery umožňují použít různé post-processing efekty. Tyto efekty mohou využít informace o barvě a hloubce, které jsou během rasterizace získány. U deferred shadingu by bylo možné využívat i normálové vektory, ale pak by bylo potřeba pro konzistenci mezi renderery rasterizovat normály

i u forward shadingu. Výstupem efektů je ale pouze barva a informaci o hloubce tak nelze měnit. V praxi se ale hloubka či normály mění málokdy a tak tento fakt nijak nevadí.

Každý efekt je reprezentován několika shadery, které se postupně provádějí. Tyto shadery obstarávají funkčnost jednotlivých kroků, ze kterých se skládá výsledný efekt. Efekt přepálení (bloom) je složený ze čtyř kroků: snížení jasu, vertikální rozmazání, horizontální rozmazání a zkombinování s originálním obrazem. Každý shader tak představuje jeden z těchto kroků.

Efekt zastoupený třídou `PostProcessingEffect` tedy obsahuje seznam kroků, kdy každý je reprezentován třídou `PostProcessingEffectPart`. Pro účely aplikace je každý efekt opatřen atributem, zdali se má efekt použít či nikoliv a lze tedy efekty dle potřeby zapínat a vypínat.

Pro aplikaci efektů je implementována třída `RenderTargetChain`. Tato třída obsahuje sadu render-targetů, mezi kterými se pak vhodným způsobem přepíná a mění se tak jejich úloha. V jeden okamžik tak render-target funguje jako textura a po přepnutí úloh funguje jako cíl vykreslování. `RenderTargetChain` obsahuje takovéto buffery čtyři: dva uložené v poli `tmpBuffers` a dva v poli `origBuffers`. Mezi buffery prvního pole se přepíná po provedení každého z kroků efektu. Tím je zajištěna návaznost těchto kroků. Mezi buffery druhého pole se přepíná mezi jednotlivými efekty a díky tomu se upravuje obraz, který je následujícím post-processing efektem považován jako originální. Plynule tak na sebe navazují i celé efekty.

efekt	Bloom				Halo		
krok	Horní Propust	Rozmazání V	Rozmazání H	Kombinace	Rozmazání V	Rozmazání H	Kombinace
origBuffer[0]							
origBuffer[1]							
tmpBuffer[0]							
tmpBuffer[1]							

Obrázek 19: Postup aplikace post-processing efektů a použití různých render-targetů. Ze zeleně označených se v aktuálním kroku čtou data. Do červených se vykresluje. Bílé jsou v kroku nevyužité.

Implementované post-processing efekty

V aplikaci byl implementován například Bloom efekt, který je podrobně popsaný v kapitole 3.3 a výsledek je na obrázku 20.

Obrázek 20: Efekt přepálení (bloom) na střeše požárního vozu.

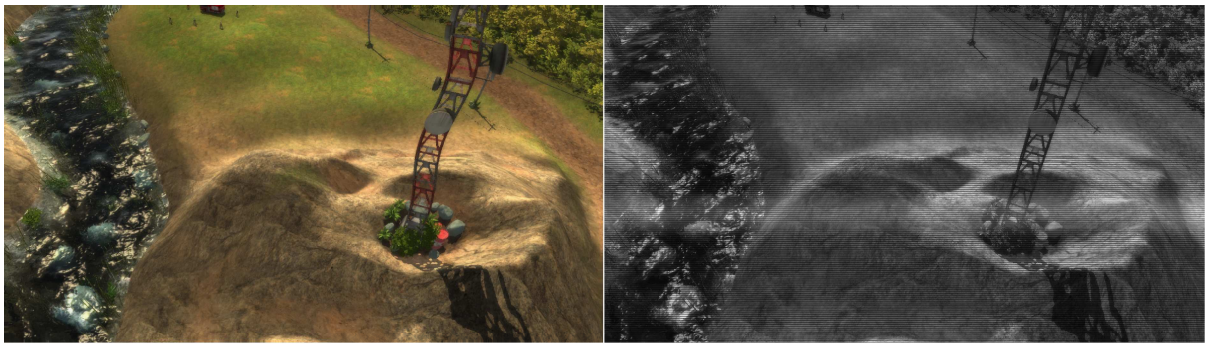
Dalším efektem je tzv. Halo efekt, který celý obraz jakoby zahalí do mlhy. Tohoto efektu je dosaženo pouhým rozmazáním obrazu a zkombinováním s originálním obrazem. Je tedy využito dvou kroků.

Implementován byl i efekt zvlnění obrazu. Tohoto zvlnění je dosaženo pomocí funkce sinus a úpravou texturovacích souřadnic podle celkového času.

Předposledním efektem je invertování obrazu. Tento efekt funguje v jediném kroku, v němž invertuje barvy i texturovací souřadnice.

Na závěr byl implementován efekt staré televize. Celý obraz je převeden do odstínů šedi. Přes obrazovku jsou zobrazeny tenké černé proužky a v závislosti na čase přes ni přejíždí jeden velký bílý pruh. Efekt je společně s efektem vlnění zachycen na obrázku 21.

Bloom a Halo efekty jsou v činnosti po celou dobu zobrazování 3D obsahu. Zbylé efekty se zapínají a vypínají podle potřeby.



Obrázek 21: Efekt vlny a staré televize v činnosti.

6.3 Částicové systémy

Ve hře hraje významnou roli oheň. Jeho zpracování tak bylo velmi důležité. Standardním způsobem pro jeho zobrazování je použití částicových systémů. Ve hře jsou tak částicové systémy implementované a navíc akcelerované na GPU, což umožňuje masivní nasazení. Kromě ohně je částicovými systémy simulován i kouř, výbuchy a částečně také voda.

6.3.1 Částice

Částice v aplikaci jsou reprezentovány několika atributy. Konkrétně se jedná o pozici, dobu do konce života částice (Time To Live, zkráceně TTL), čas vytvoření částice, rychlost pohybu a rotace. Kvůli využití GPU pro akceleraci výpočtu je většina těchto atributů umístěna v bufferech, které se používají stejně jako u post-processingu. Buffery jsou tedy zdvojené, kdy se jeden buffer z dvojice používá jako cíl renderování a druhý jako textura se zdrojovými daty. Po provedení výpočtu se jejich úlohy prohodí. Každý pixel bufferu odpovídá jedné částici a kapacita částicového systému je tak omezená rozměry těchto bufferů.

Veškeré atributy částic jsou reprezentovány 32fp hodnotami a všechny render-targety obsahující jejich atributy by se analogicky k G-bufferu daly označit za jakýsi Particle-buffer, jehož podoba je následující:

- 4x 32fp hodnoty – pozice a TTL
- 4x 32fp hodnoty – rychlost pohybu a rotace

Atribut doby vytvoření na GPU uložen není, protože nemá pro výpočet kroku simulace význam. Tento atribut se projeví až při odstraňování mrtvých částic a byl zaveden, přestože lze částice odstraňovat i na základě hodnoty TTL. Důvod použití tohoto atributu bude popsán později.

Vzhledem k tomu, že u částic není potřeba uchovávat žádná originální data a je potřeba přepínat pouze mezi dvěma buffery, není využito třídy `RenderTargetChain` z post-processingu, ale byla vytvořena třída `ParticleRenderTargetChain`, která obsahuje pouze potřebný počet bufferů.

Pro vykreslování částic je použit čtvercový polygon, na který je nanesena textura. Touto texturou může být standardní obrázek či dokonce 3D textura, kdy je každá vrstva použita v určitý čas a částice je tak v průběhu animovaná.

6.3.2 Částicový systém a manažer částic

Každý částicový systém je reprezentovaný třídou `ParticleSystem`, která obsahuje veškerá data o částicích patřících do tohoto systému. Data nutná pro GPU a pro výpočet simulace jsou uložená ve výše popsaných bufferech. Kromě nich jsou pro výpočet důležité další parametry, které jsou ale pro všechny částice systému společné. Jedná se o gravitaci, vítr a velikost částic.

Celková funkčnost je rozdělená na CPU část a GPU část, kdy na CPU probíhá generování a odstraňování částic a na GPU se provádí výpočet simulace. Procesorová část si vkládá data nově vygenerovaných částic do polí a ve vhodný čas jsou všechny tyto částice přidány do render-targetu. Teoreticky by bylo možné přidávat každou částici zvlášť, ale z pohledu celkové rychlosti je výhodnější tento proces provádět po dávkách a přenášet z paměti RAM do paměti grafické karty co možná nejvíce dat najednou.

Při přidávání nových částic se však vyskytl problém s výkonem, který nedosahoval předpokládaných výsledků. Pro úspěšné přidání částic je totiž nutné aktualizovat data v grafické kartě a toho bylo dosaženo pomocí metod `GetData(...)` a `SetData(...)`, které poskytuje XNA u použité třídy `RenderTarget2D`. Tyto metody fungují velice transparentně, ale přenášejí mezi grafickou kartou a procesorem velké množství dat a proto docházelo k propadu výkonu. Prvním pokusem o řešení bylo, že se z grafické karty nestahovala všechna data, ale pouze obdélník, který obsahoval oblast zájmu. I přesto ale byly propady výkonu značné a z toho důvodu bylo použito řešení, které sice není programově tak jasné, ale funguje velice rychle a stejně spolehlivě.

Data částic jsou uložená v render-targetech a není nic jednoduššího než tyto objekty využít k jejich primárnímu účelu – vykreslování. Nové částice jsou do nich tedy přidány jejich vykreslením do těchto render-targetů. Pro každou novou částici se tak použije malý čtverec, který se umístí na pixel render-targetu představující volnou pozici a v pixel-shaderu se vykreslí data této částice. Díky tomu není třeba přenášet velké množství dat na procesor, zde je upravit a pak je přenést zpět do grafické karty. Je zde také využito instancingu, kdy se nachystají data všech nových částic a jediným voláním vykreslovací funkce se přidají všechny tyto částice. Úzké hrdlo, které se u přidávání částic projevilo, bylo úspěšně odstraněno.

Odstraňování částic bylo stejně problematické jako přidávání. V první verzi implementace bylo prováděné na základě TTL. To se ale aktualizuje v každém kroku výpočtu a tak bylo nutné si vytáhnout aktuální data z grafické karty. I k tomuto byla použita metoda `GetData(...)`, ale tak jako u přidávání částic, i zde docházelo k velkému propadu výkonu. Navíc je zde šance, že během jednoho snímku zemřou všechny částice a nelze tak použít ani optimalizaci s využitím stahování pouze oblasti zájmu, protože touto oblastí je celý render-target.

Řešením bylo použití dalšího atributu částic, který ale není na grafické kartě, ale pouze u procesoru. Tímto atributem je výše zmíněný čas vytvoření částice. Díky tomu, že všechny částice jednoho systému mají pevně stanovenou dobu života, lze určit stav částice na základě času jejího vytvoření, životnosti částic a aktuálního času. V částicovém systému je udržována řada ukazatelů, mezi nimiž je i ukazatel na první živou částici. Díky postupnému vytváření a přidávání částic máme tedy jistotu, že pokud některé částice zahynou, bude první z nich ležet právě pod tímto ukazatelem. Z principu přidávání částic taktéž plyne, že když při odstraňování mrtvých částic narazíme na první částici, která stále ještě žije, další částice budou také stále ve stavu žijících částic. Díky tomu se

```
if (čas_vytvoření_částice + doba_života_částic < aktuální_čas)
      stav = mrtvá_částice
else
      stav = živá_částice
```

Text 3: Pseudokód rozhodování o stavu částice.

proces odstraňování částic provádí jen po minimální možnou dobu, protože je daná startovní pozice a při první příležitosti se proces přeruší.

Kvůli dalšímu atributu se sice používá více paměti, protože je nutné udržovat pro každou částici další hodnotu, ale neztratíme tím výkon jako při použití metody `GetData(...)` a využívání aktuálních hodnot TTL konkrétních částic. Je tak odstraněno další úzké hrdlo.

Díky oddělení různých částicových systémů je možné provádět u každého systému výpočet kroku simulace trochu odlišným způsobem. Pokud ale není určen jiný způsob, je použita implicitní metoda výpočtu. V shaderu zodpovědném za tento výpočet se pro aktuálně zpracovávanou částici nejdříve načtou vstupní data z textur. Tato data jsou poté rozdělena na samostatné jednotky a poté je proveden výpočet. Ten je realizován podle následujících vzorců:

$$P_{new} = P_{old} + V_{old} \cdot \Delta T \tag{9}$$

$$TTL_{new} = TTL_{old} - \Delta T \tag{10}$$

$$V_{new} = V_{old} + (W + G) \cdot \Delta T \tag{11}$$

Dolní indexy „new“ značí nové hodnoty, „old“ značí hodnoty staré. P *pak představuje pozici,* TTL *dobu do uplynutí života částice,* V *rychlost pochybu,* ΔT *uplynulý časový úsek,* W *sílu větru a* G *gravitační zrychlení.*

Po spočtení nových hodnot jsou tyto uloženy do připravených render-targetů. Kromě nezávislého výpočtu kroku lze také určit i specifický způsob vykreslování. Oheň se tak vykresluje jiným způsobem než kouř.

V aplikaci je třídou `ParticleManager` implementován jednoduchý manažer, který zapouzdřuje činnost spojenou se všemi částicovými systémy a usnadňuje tak práci s nimi. Metoda `AddParticleSystem(...)` přidá nový částicový systém, `Update(...)` provede odstranění mrtvých částic nad všemi systémy, `Calculate(...)` provede výpočet jednoho kroku simulace a `DrawParticles(...)` vykreslí všechny částice. Mimo to poskytuje také statistické informace o částicích a využití systémů.

V metodě `Calculate(...)` se u všech částicových systémů volají metody `Move(...)` a `AddNewParticlesToGpu(...)`. První metoda provede samotný výpočet kroku simulace, druhá pak vloží nově přidané částice do bufferů uložených na grafické kartě.

6.3.3 Emitory částic

Ve hře se částicové systémy používají k vytváření různých efektů. Každý z nich potřebuje odlišný způsob generování částic a proto bylo implementováno více druhů emitorů částic. Emitorem se chápe určitý bod v prostoru, ze kterého jsou do scény chrleny nové částice.

Emitory jsou implementovány jako uzly grafu scény a dědí tak z obecné třídy `Node`. Díky tomu mají základní vlastnosti jako je pozice ve scéně, rotace, velikost a mnoho dalších. Emitor tak lze jednoduše umístit či dokonce navázat na jiné objekty. Každý emitor je navázaný na určitý částicový systém a nové částice jsou do systému přidávány pomocí metody `AddParticle(...)`, kterou implementuje třída `ParticleSystem`. Každý emitor má také specifikovaný počet částic, které má vygenerovat během jedné sekundy.

Prvním typem emitoru je základní generátor částic, který je v aplikaci implementován pomocí třídy `ParticleEmitterNode`. Tento emitor generuje částice podle několika vstupních parametrů. Podle zadaného úhlu se pomocí náhodného generátoru určí pozice na výseči jednotkové koule. Tuto pozici lze poté upravit pomocí dalších parametrů. Osa generování je určena rotací celého uzlu grafu scény a částice tedy lze generovat na libovolném místě. Počáteční rychlost částic se určuje stejným způsobem jako jejich pozice. Základ je tedy určen úhlem na kulové výseči a poté je upraven dalšími

parametry. Díky těmto parametrům lze částice generovat na kouli, elipsoidu či jejich rovinných variantách. Tento typ emitoru je použit například pro generování kouře.

Pro oheň se předchozí emitor příliš nehodil, protože každá nová částice je umístěna nezávisle na předchozích. Oheň je ale složen z plamenů a každý z nich se chvíli pohybuje přibližně stejným směrem. Bylo tak potřeba zajistit určitou soudržnost několika částic aby svým pohybem simulovaly plameny. K tomu byl vytvořen druhý typ emitoru, který je v kódu reprezentovaný třídou `BatchParticleEmitterNode`. Ten negeneruje částice zcela náhodně, ale po dávkách o zadané velikosti. Princip generování je podobný předchozímu, ale v první fázi se vygeneruje směr a pozice celé dávky. Poté se pro tuto dávku postupně generují částice, které jsou vždy mírně odchýlené od pozice a rychlosti pro celou dávku. Po dosažení velikosti dávky jsou vygenerována nová čísla pro celou sérii nových částic a proces se spouští nanovo. Díky dávkovému generování lze oheň vytvořit jako řadu plamenů, které stále vznikají a zanikají a je tak simulováno jejich šlehání do prostoru. A díky mírnému vychylování jednotlivých částic tyto plameny vypadají náhodně a živelně. Oheň se tak chová poměrně věrohodně.

Posledním typem emitoru je `ExplosionParticleEmitter`, který slouží k simulaci explozí. Tento emitor by šel složit z několika obyčejných emitorů, ale pak by bylo nutné při každé explozi vytvořit řadu uzlů grafu scény a vhodně je spojit. Snazším řešením se tak jevilo vytvořit speciální typ emitoru, který se o toto postará sám. Tento emitor má proto několik částí, které fungují jako virtuální uzly grafu scény. Prvním je střed výbuchu, který zůstává po celou dobu nehybný. Dalšími je předem určený počet letících částí, které imitují hořící šrapnely vymrštěné při výbuchu. Každý ze šrapnelů se pohybuje od místa výbuchu a je ovlivněn gravitací. Všechny pak po cestě generují částice a nechávají za sebou ohnivou stopu.

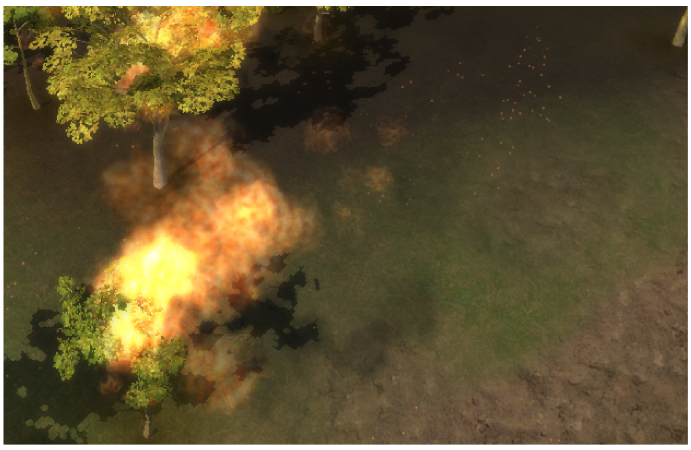
6.3.4 Implementované částicové systémy

Oheň

Pro simulaci ohně byly použity tři různé částicové systémy, které se liší velikostí částic a rychlostí, jakou jsou částice generovány. Emitory generující dávky částic jsou navázané na hořící objekty a podle žáru těchto objektů je určen i částicový systém. Objekty, které vydávají více žáru, hoří silnějším ohněm.

Pro vykreslování částic ohně je použito aditivního alpha-blendingu, kdy se nově vykreslená barva přičte k barvě, která je již ve frame-bufferu. K tomu je použita animovaná textura pomocí 3D textury. Několik prvních vrstev představuje samotný oheň, zbylé vrstvy pak jiskry létající z ohně.

Celý proces vykreslování je navíc ovlivněn shaderem, který upravuje velikost částice v závislosti na její zbývající životnosti. Podobným způsobem je upravována i průhlednost částice. Je tím dosaženo plynulého objevování a mizení částic.

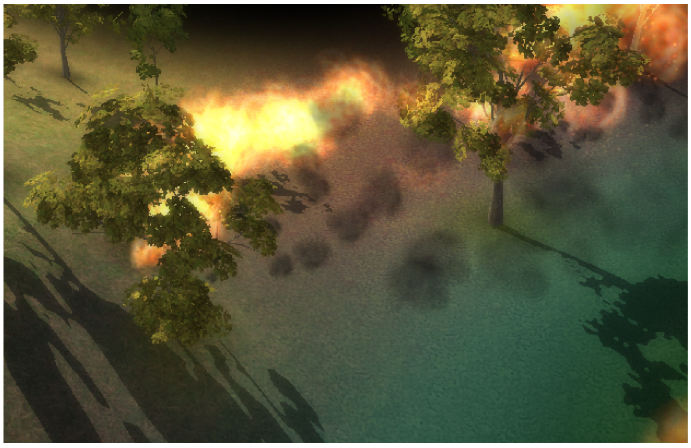


Obrázek 22: Částicový systém ohně. V pravém horním rohu si lze všimnout malých jisker.

Kouř

Pro kouř byl použitý obyčejný emitor a tak jako emitory ohně jsou i tyto navázány na hořící objekty. Jsou ale umístěny mírně nad tyto objekty, aby kouř vznikal nad plameny.

Pro vykreslení je v případě kouře použita jediná 2D textura a standardní alpha-blending, kdy se podle průhlednosti přepočítává příspěvek vykreslovaného pixelu. Tak jako u ohně, i zde je vytvořen speciální shader, který upravuje průhlednost a velikost částice. V tomto případě se však průhlednost postupně snižuje a velikost naopak zvětšuje. Je tak simulováno rozptylování kouře ve vzduchu.



Obrázek 23: Částicový systém pro zobrazení kouře. Ve hře je úzce svázaný s ohněm.

Voda

Ve hře lze oheň hasit a pro simulaci stříkané vody je připraven speciální částicový systém. Pro výpočet kroku je zde použit shader, který nepostupuje podle vzorců 9, 10 a 11, ale upraví pouze TTL částice. Aktuální pozice je spočtena až v shaderu zodpovědném za vykreslování. Výpočet má k dispozici počáteční a koncovou pozici a z nich na základě TTL částice spočte vzdálenost mezi těmito dvěma body. Aby se částice pohybovala podobně jako kdyby na ni působila gravitace, je upravena ještě výška v jaké se částice nachází. Částice tak sleduje parabolu, která je mírně deformovaná podle převýšení počáteční a koncové pozice.

Vykreslení je pak provedeno podobně jako u kouře. Velikost i průhlednost se tak postupně upravují a je použit standardní alpha-blending.

Obrázek 24: Voda stříkaná z hadic je vytvořena upraveným částicovým systémem.

Exploze

Pro explozi byl použit speciálně vytvořený emitor a pro výpočet kroku simulace je použit implicitní způsob. Vykreslování pak probíhá podobným způsobem jako oheň. Při vykreslení se aplikuje aditivní alpha-blending. Velikost a průhlednost jsou opět upravovány pro plynulé objevování a mizení částic.

Obrázek 25: Výbuch vytvořený speciálním emitorem simulujícím letící trosky.

6.4 Integrace do hry

Všechny implementované součásti byly postupně integrovány do výsledné aplikace. Její návrh ale také musel respektovat použití všech implementovaných částí.

Třída `SceneManager` obsahuje graf scény a je zodpovědná za aktualizaci všech uzlů v průběhu hry metodou `Update(...)` a také za jejich vykreslení metodou `Draw(...)`. Při aktualizaci se generují nové částice u emitorů. Těmto uzlům se předá uplynulý časový okamžik a podle něj se spočítá počet generovaných částic v tomto snímku. Poté jsou tyto částice vygenerovány a přidány do částicových systémů jako nové částice.

```
ParticleManager.Calculate();

Renderer.BeginOpaque();
DrawList(opaqueNodes, lightInformation);

Renderer.ApplyLighting(lightInformation);

Renderer.DrawWater();

Renderer.BeginAlphaBlend();
DrawList(blendedNodes, lightInformation);

ParticleManager.DrawParticles();

Renderer.ApplyPostProcessing();

Renderer.End();
```

Text 4: Princip funkčnosti vykreslování jednoho snímku.

Pro snadné manipulování se světly jsou tato implementována jako další uzly grafu scény. Při vykreslení jsou pak každému uzlu grafu scény předány informace o těchto světlech. Tyto informace

však mají význam pouze v případě forward shadingu, protože právě u této metody se osvětlení počítá během vykreslování. U deferred shadingu jsou tyto informace ignorovány a využity až o chvíli později. Metoda `Draw(...)` je implementována podle pseudokódu v textu 4.
Nejprve se provede výpočet kroku simulace částicových systémů. Logicky by tento krok měl být proveden v metodě `Update(...)`, ale vzhledem k tomu, že tento výpočet probíhá vykreslováním do render-targetů, je vhodnější použít metodu `Draw(...)`. Poté započne samotné vykreslování a právě zde se využívá metod rendereru. Nejprve se vykreslí neprůhledné objekty. Poté se aplikuje osvětlení, vykreslí vodní plochy a pak průhledné objekty. Nato se pomocí manažeru částic vykreslí všechny částice. Nakonec jsou aplikovány post-processing efekty a ukončeno vykreslování.

7 Experimenty

Díky implementaci obou metod renderování je možné alespoň jednoduché porovnání. ForwardShadingRenderer umožňuje použít pouze jedno a proto je toto porovnání provedeno právě s jedním světelným zdrojem. Z principu deferred shadingu se očekává, že u jediného světla se neprojeví jeho síla a z důvodů vyšší režie bude spíše pomalejší než forward shading. Na druhé straně lze použít více světel a ověřit lineární složitost výpočtu osvětlení.

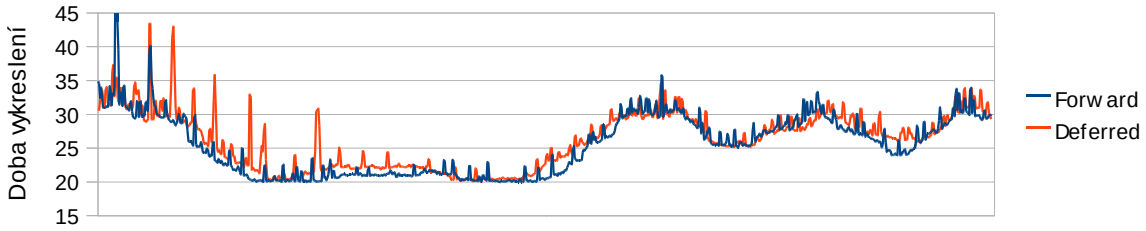
Do druhé sady testů jsou zařazeny výpočty simulace částicových systémů. V rámci zvyšování výkonu verze s akcelerací bylo vhodné porovnání s částicovým systémem, který využívá pouze procesor. Proto byla implementována třída `ParticleSystemCPU`, která je v principu stejná jako třída `ParticleSystem`, ale grafická karta se používá pouze pro vykreslování částic.

Všechny testy jsou prováděny na stejné konfiguraci: Pentium Dual Core E5200, 2GB RAM, AMD Radeon 5750.

7.1 Porovnání metod vykreslování

Ve hře jsou tři hratelné mise, které jsou si kvůli úzkému zaměření hry poměrně podobné. Přesto se však v několika ohledech liší. Rozdíly jsou v množství objektů, jejich komplexitě a také je zde různě zvlněný terén, což se v kombinaci s technikou dynamicky se měnící úrovně detailů také projevuje na rychlosti vykreslování. Rovněž se liší v objemu vykreslovaných částic.

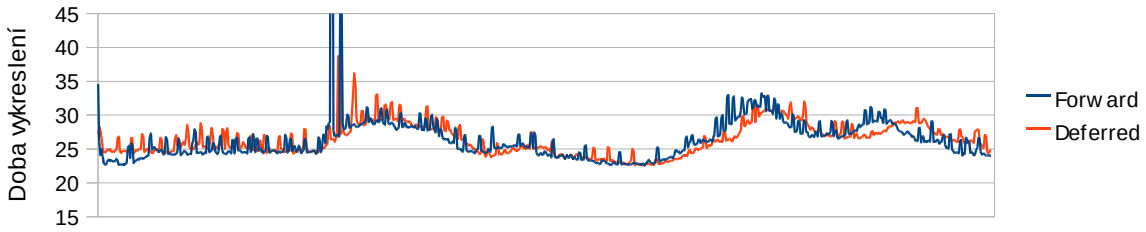
Pro účely měření bylo v aplikaci doimplementováno vypisování několika údajů do výstupního souboru. V experimentech zaměřených na porovnávání vykreslovacích metod se do tohoto souboru zapisují údaje o době potřebné k vykreslení snímku. Během samotného měření pak byly u každé mise ručně provedeny přelety nad terénem. Konkrétní výsledky se tak v čase mohou lišit, charakter by ale měl být zachován.



Graf 1: Porovnání metod, scéna 1

Z grafu 1 lze vyčíst, že průběh v čase je skutečně podobný a i přes ruční manipulaci kamery a drobné odchylky lze výsledky použít k porovnání metod. Obě metody podávají velice podobné výsledky a jen těžko se odhaduje, která si vede lépe. Lze si však všimnout, že metoda forward shading se v některých situacích dostává na nižší hodnoty času potřebného k vykreslení snímku.

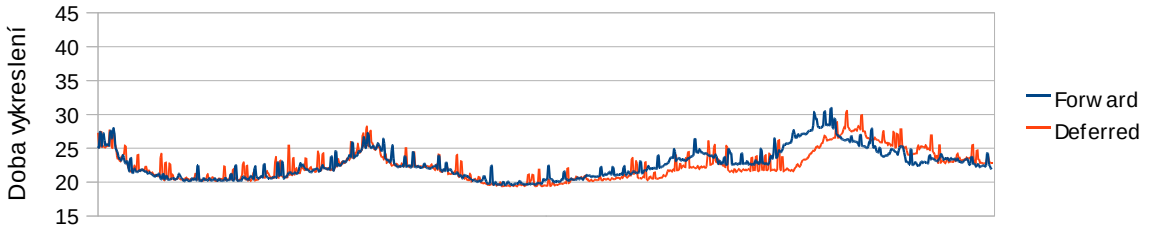
Graf 2 ukazuje výsledky naměřené ze druhé scény. I zde je průběh v čase velice podobný a akorát ke konci měření se rozcházejí. Rovněž je velmi podobný výkon obou metod. U první scény



Graf 2: Porovnání metod, scéna 2

se forward shading v několika časových okamžicích choval lépe. V této misi se forward shading také chová o trochu lépe.

Graf 3 z poslední mise jen potvrzuje, že mezi výkony metod není větší rozdíl. Průběh se opět liší až na konci měření. Právě zde si můžeme všimnout, že metoda deferred shading podává mírně lepší výsledky.



Graf 3: Porovnání metod, scéna 3

Podle výsledků ze všech tří misí je velmi obtížné rozhodnout, která metoda je výhodnější. Předpoklad byl, že metoda deferred shading bude mít kvůli své režii na přípravu G-bufferu a světelných zdrojů nižší výkon. To se ale v provedených experimentech nijak výrazně neprojevilo a obě metody lze spíše považovat za srovnatelné.

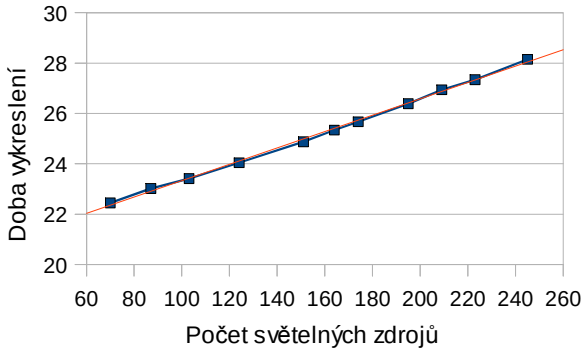
7.2 Osvětlení metodou deferred shading

Implementace metody deferred shading dovoluje použití značného množství světel. Podle původních plánů měly být světelnými zdroji některé částice představující oheň. Těchto částic však bývá obrovské množství (v závislosti na scéně a průběhu hry až čtvrt milionu) a navíc žijí po relativně krátkou dobu. Bylo by tedy poměrně obtížné a náročné stále vytvářet a rušit světelné zdroje a navíc zajišťovat jejich pohyb s částicemi.

Světelné zdroje tedy nejsou vázané na samotné částice, ale na emitory těchto částic. Dalo by se tedy říci, že světlo je umístěné ve středu ohně. Vzhledem k tomu, že každý strom má svůj emitor, pak by v extrémním případě mohl počet světel narůst až k hodnotě přesahující tisíc zdrojů. V implementaci jsou ale jednoduché optimalizace pro ořezávání scény pohledovým tělesem. V kombinaci s omezeným dosahem světel je tedy reálný počet kolem tří set zdrojů.

Pro otestování byla použita první mise, kde zpočátku hoří malý počet stromů. Jak se oheň rozhořívá, stoupá i počet světel. Měření bylo prováděno po částech. Hra se nechala po malou chvíli běžet a poté se pozastavilo plynutí času. V tento okamžik započalo měření doby vykreslování. Po nasbírání dat se hra opět spustila aby narostl počet světel. Takto se získala data pro několik situací.

Z naměřených dat se poté spočítaly průměrné hodnoty pro daný počet světel. Výsledky jsou vynesené do grafu 4. Z něj lze vyčíst, že náročnost výpočtu osvětlení odpovídá lineární křivce a graf tak dokazuje velmi dobrou efektivitu metody při vysokém počtu světelných zdrojů.



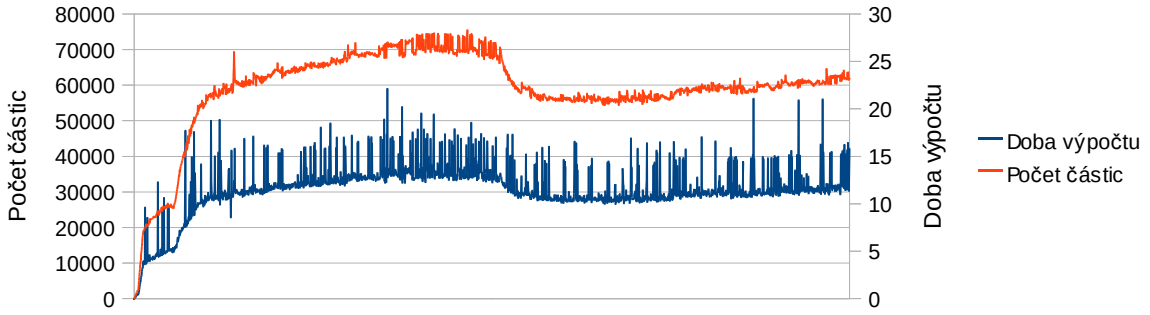
Graf 4: Závislost doby vykreslení na počtu světelných zdrojů.

7.3 Částicové systémy na CPU a GPU

Ve všech třech hratelných misích po většinu času hoří značná část lesa. Rozloha a průběh hoření se však liší a pro test částicových systémů jsou tedy mise ideální. Experimenty tedy byly provedeny na všech scénách a to jak s verzí běžící na procesoru, tak s verzí akcelerovanou na grafické kartě. V každém snímku se do výstupního souboru vypíše počet částic a doba trvání výpočtu kroku simulace. Měření je zastaveno po skončení mise.

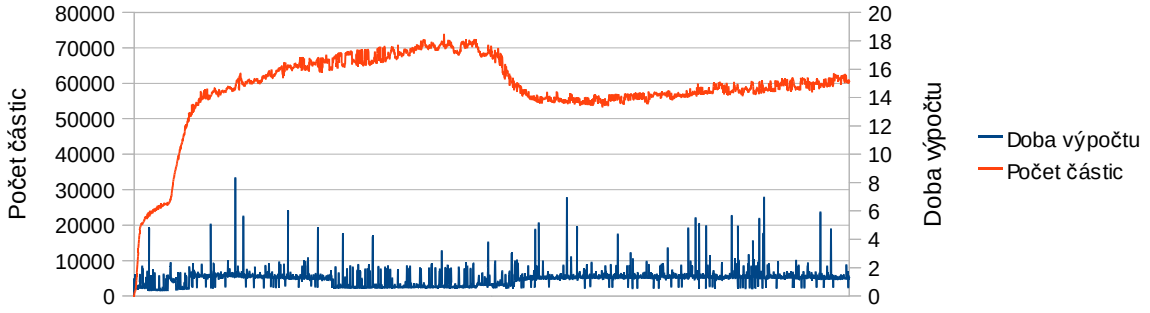
Verze běžící na procesoru vždy počítá krok simulace pouze pro aktivní částice. U neaktivních by toto bylo zbytečné a akorát by se plýtvalo výkonem. Na druhé straně u GPU verze se vždy přepočítává celý systém, protože data jsou uložená v render-targetech a grafická karta nerozezná aktivní částici od neaktivní. V shaderu se sice testuje TTL částice a podle toho se buď provede výpočet, nebo pouze vynulují všechny hodnoty. To má ale nejspíše pouze zanedbatelný vliv na celkový výsledek.

Je také třeba uvést, že při testech bylo použito celkem sedm částicových systémů. Rozměry render-targetů pak byly 512*512 pro tři ze systémů a 128*128 pro zbylé čtyři systémy. Celkově je tak možné simulovat až 851 968 částic, k čemuž na grafické kartě skutečně dochází.



Graf 5: CPU verze, scéna 1

Grafy 5 a 6 ukazují výsledky CPU resp. GPU verze naměřené v první misi. Je vidět, že výkon CPU verze závisí na počtu částic v systému, protože se krok počítá jen pro aktivní částice. Na grafické kartě ale žádná souvislost mezi počtem aktivních částic a dobou výpočtu vidět není.

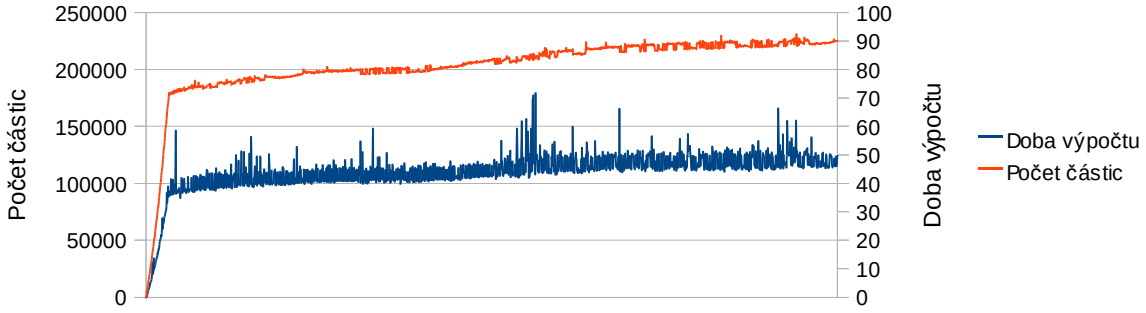


Graf 6: GPU verze, scéna 1

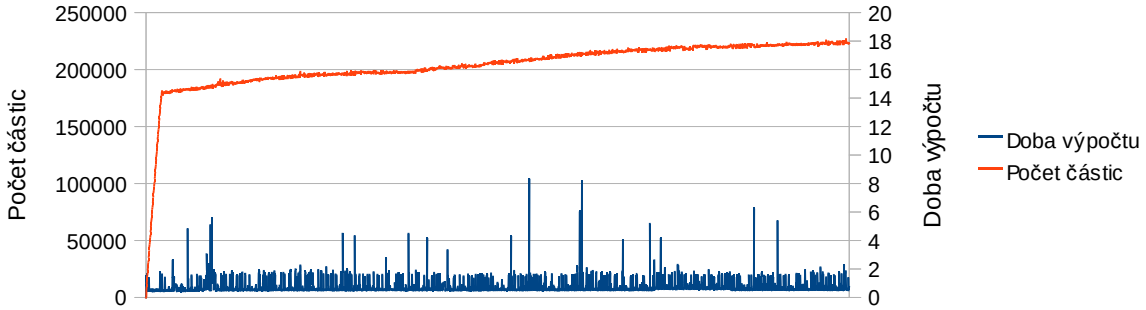
Z grafů lze také vyčíst, že v první misi bylo použito maximálně necelých osmdesát tisíc částic a u verze běžící na procesoru trval výpočet tohoto množství částic zhruba patnáct milisekund. To odpovídá asi šedesáti snímkům za sekundu. Je však třeba si uvědomit, že ve hře se nepočítají jenom částicové systémy, ale spousta dalších věcí, a každý ušetřený zlomek sekundy se počítá.

Grafy 7 a 8 zobrazují výsledky naměřené v misi druhé. CPU verze opět ukazuje závislost doby výpočtu na počtu částic, zatímco GPU verze má spíše konstantní charakter. Během této mise počet částic dosahuje téměř čtvrt milionu. Doba výpočtu na procesoru se při tomto množství částic pohybovala kolem padesáti milisekund, což odpovídá asi dvaceti snímkům za sekundu. Za plynulou animaci se běžně považuje třicet snímků za sekundu, procesor je tedy v tomto ohledu už

nedostatečný. Naproti tomu grafická karta se, podobně jako u první mise, stále drží na čase pohybujícím se kolem jedné milisekundy.

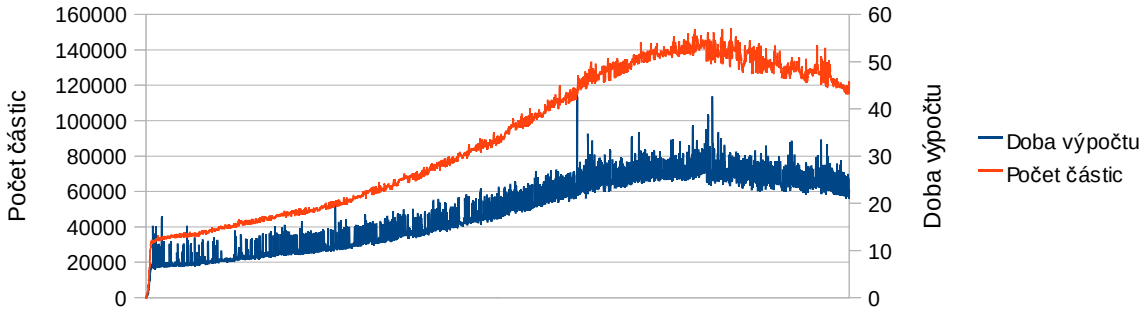


Graf 7: CPU verze, scéna 2

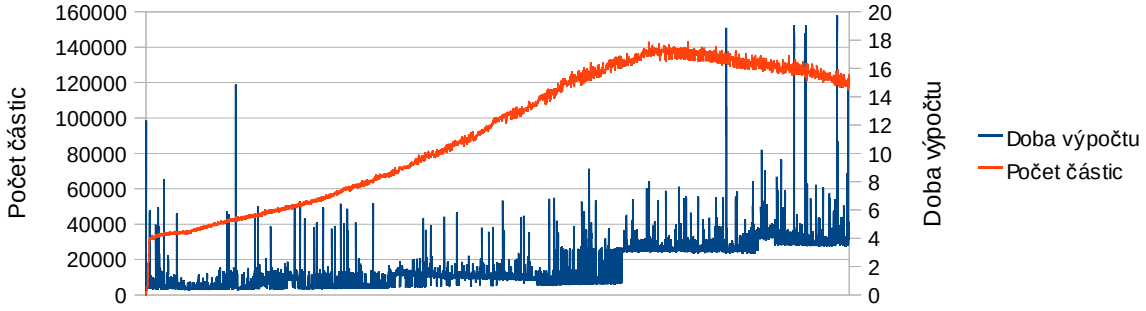


Graf 8: GPU verze, scéna 2

Grafy 9 a 10 ze třetí mise už jen potvrzují doposud naměřené výsledky. Pouze u GPU verze se zhruba od dvou třetin herní doby objevuje navýšení doby výpočtu. Počet částic a výsledky z dosavadního průběhu či předchozích misí ale nenasvědčují, že by příčinou byl právě počet částic. Je možné, že došlo k nějaké nezjištěné události na grafické kartě a to následně způsobilo pokles výkonu při výpočtu.



Graf 9: CPU verze, scéna 3



Graf 10: GPU verze, scéna 3

Z naměřených výsledků plyne, že akcelerace na grafické kartě funguje dobře a pro hru se úsilí vložené do její implementace vyplatila. Díky vysokému výkonu je možné provádět výpočet obrovského množství částic. Na procesoru by takový výpočet pravděpodobně nebyl možný. Rovněž je nutné připomenout, že výsledky obsahují pouze výpočet kroku simulace. Doba potřebná k vykreslení částic zde uvedená není. Zde by se rozdíl mezi GPU a CPU verzemi ještě prohloubil, protože akcelerovaná verze má již všechna data na grafické kartě, kdežto CPU verze musí tato data při vykreslování dodat.

8 Závěr

Cílem práce bylo nastudovat v současnosti používané techniky vykreslování komplexních scén v počítačových hrách. Byly prozkoumány tři odlišné metody pro zobrazování v reálném čase, které se aktuálně používají či se testují a dosahují poměrně příznivých výsledků. Byly taktéž popsány různé speciální efekty, kterých se ve hrách využívá. Výstupem měla být aplikace, která by demonstrovala použití vhodné techniky.

V rámci práce byla implementována hra, ve které lze použít jak metodu forward shading, tak metodu deferred shading. Ve hře byly použity masivní částicové systémy pro simulaci ohně a dalších efektů. Výpočet částic je akcelerovaný na grafické kartě. Pro porovnání byl implementován i výpočet na procesoru, ale kvůli velice nízkému výkonu není použit. Dále bylo implementováno použití post-processing efektů, které dodatečně upravují vykreslený obraz.

Díky implementaci různých variant vykreslování či výpočtů bylo možné provést porovnání. Byly změřeny a zhodnoceny výsledky obou implementovaných metod vykreslování a také byla ověřena časová složitost výpočtu osvětlení u metody deferred shading. U částicových systémů bylo měřením zjištěno, že výpočet na procesoru je příliš pomalý a používání grafické karty k akceleraci tedy není bez důvodu.

Výsledná hra má několik hratelných úrovní, hojně využívá částicové systémy a post-processing efekty. Lze přepínat mezi technikami vykreslování, kdy se při použití deferred shadingu používá velké množství světelných zdrojů bez velkého dopadu na výsledný výkon.

Celá aplikace je naprogramována v jazyce C# za použití frameworku XNA. V průběhu implementačních prací bylo zjištěno a mnohokrát oceněno, že XNA poskytuje velmi mnoho různých struktur a metod, které by jinak programátor musel psát sám. Na druhé straně se ale objevily některé problémy spojené s vlastnostmi frameworku, které práci spíš komplikují a musely být různě obcházeny.

Na aplikaci by se dal vylepšit především výsledný výkon. Na průměrném stroji běží hra po většinu času na hranici plynulosti a prostoru pro optimalizace je pravděpodobně dost. Dále by se dala rozšířit metoda forward shadingu, která zatím používá pouze jediný zdroj světla. Poté by se daly obě použité metody vykreslování porovnat i při velkém množství světel.

Literatura

1. WRIGHT, Richard S.; LIPCHAK, Benjamin; HAEMEL, Nicholas. OpenGL SuperBible – Fourth Edition. Addison – Wesley, 2007.
2. SAITO, Takafumi; TAKAHASHI Tokiichiro. SIGGRAPH '90. 1990, [cit. 2011-01-06]. Dostupné z WWW: <http://portal.acm.org/citation.cfm?id=97901>.
3. HARGREAVES, Shawn. Deferred Shading. Game Developers Conference. 2004, [cit. 2011-01-06]. Dostupné z WWW: <http://read.pudn.com/downloads160/sourcecode/game/724029/DeferredShading.pdf>.
4. CALVER, Dean. Beyond3D [online]. 2003-07-31, [cit. 2011-01-06]. Photo-realistic Deferred Lighting – Page 1. Dostupné z WWW: <http://www.beyond3d.com/content/articles/19/1>.
5. PERSSON, Emil. Humus – 3D [online]. 2007-10-23, [cit. 2011-01-06]. Deferred shading. Dostupné z WWW: <http://www.humus.name/index.php?page=3D&ID=74>.
6. POLICARPO, Fabio; FONSECA, Francisco. Deferred Shading Tutorial [online]. 2007. Dostupné z WWW: <http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf>.
7. STARÝ, Petr. Deferred Shading. Brno, 2009. Diplomová práce. FIT, VUT v Brně.
8. SHISHKOVTSOV, Oles. GPU Gems 2. Addison – Wesley, 2005. Deferred Shading in S.T.A.L.K.E.R, s. 345-366.
9. SPILLE, Carsten. PC Games Hardware [online]. 2009-01-28, [cit. 2011-01-06]. Starcraft 2: Technology and engine dissected. Dostupné z WWW: <http://www.pcgameshardware.com/aid,674502/Starcraft-2-Technology-and-engine-dissected/News/>.
10. HOWARD, Jeffrey. Intel Software Network [online]. 2007-10-10, [cit. 2011-01-06]. Real Time Ray-Tracing: The End of Rasterization?. Dostupné z WWW: <http://blogs.intel.com/research/2007/10/real_time_raytracing_the_end_o.php>.
11. POHL, Daniel. Intel Software Network [online]. 2009-08-12, [cit. 2011-01-06]. Quake Wars Gets Ray Traced. Dostupné z WWW: <http://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/>.
12. POHL, Daniel. Intel Software Network [online]. 2010-09-12, [cit. 2011-01-06]. Wolfenstein gets ray traced – on your laptop!. Dostupné z WWW: <http://blogs.intel.com/research/2010/09/wolfenstein_gets_ray_traced_-.php>.
13. DIMITROV, Rouslan. Cascaded Shadow Maps [online]. NVIDIA Corporation, 2007, [cit. 2011-01-06]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf>.
14. MARTIN, Tobias; TAN, Tiow-Seng. Anti-aliasing and Continuity with Trapezoidal Shadow Maps [online]. The Eurographics Association, 2004. Referát. School of Computing, National University of Singapore. Dostupné z WWW: <http://www.comp.nus.edu.sg/~tants/tsm/tsm.pdf>.
15. WIMMER, Michael; SCHERZER, Daniel; PURGATHOFER, Werner. Light Space Perspective Shadow Maps [online]. The Eurographics Association, 2004. Referát. Vienna University of Technology. Dostupné z WWW: <http://www.cg.tuwien.ac.at/research/vr/lispsm/shadows_egsr2004_revised.pdf>.
16. BRABEC, Stefan; ANNEN Thomas; SEIDEL, Hans-Peter. Shadow Mapping for Hemispherical and Omnidirectional Light Sources. 2002, [cit. 2011-01-06]. Dostupné z WWW: <http://www.thomasannen.com/pub/cgi2002.pdf>.
17. FERNANDO, Randima. Percentage-Closer soft shadows [online]. NVIDIA Corporation, 2005, [cit. 2011-01-06]. Dostupné z WWW: <http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf>.

18. KLUCHER, Michael. XNA Game Studio Team Blog [online]. 2006-08-29, [cit. 2011-01-06]. The XNA Framework Content Pipeline. Dostupné z WWW: <http://blogs.msdn.com/b/xna/archive/2006/08/29/730168.aspx>.
19. MSDN [online]. [cit. 2011-01-09]. XNA Game Studio 4.0. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb200104.aspx>.
20. HARGREAVES, Shawn. Shawn Hargreaves Blog [online]. 2008-11-24 [cit. 2011-01-06]. Content Pipeline assemblies. Dostupné z WWW: <http://blogs.msdn.com/b/shawnhar/archive/2008/11/24/content-pipeline-assemblies.aspx>.

Přílohy

Příloha 1: DVD nosič obsahující:

- instalační balík se hrou
- game-play instrukce
- zdrojové kódy
- plakát a video prezentující dosažené výsledky