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

Bakalářská práce

Možnosti enginu OGRE pro stereoskopické zobrazování

Plzeň, 2009 Jiří Fatka

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne.....

Abstrakt

Ogre abilities in stereoscopy

Rendering complex scenes requires a lot of work if it's based only on pure rendering APIs (like DirectX and OpenGL). This is a reason for creating graphics engines that come with support of making complex scenes easier because they integrate some functions that are same in every graphic application e.g. scene managment.

This bachelor thesis describes graphics engine Ogre (Object-Oriented Graphics Rendering Engine), it's design and internal structure, positives and negatives and abilities to render some effects (and how).

Second part of this thesis is about stereoscopy (what stereoscopy is) and how implement stereo in Ogre, problems with implementation and how to solve those problems.

Obsah

Kapitola 1

Úvod

Grafické enginy jsou jádra, která poskytují nástroje pro základní operace renderování scény. Programátor se proto nemusí starat o některé základní věci spojené s renderováním scény jako je např. způsob renderování modelu, transformace a stínování. Některé enginy umožňují za sebe schovat použitý renderovací systém jako je Direct3D nebo OpenGL a jiné jsou navrženy i pro více herních platforem najednou, jako je PC, XBox či PlayStation.

Dříve si jednotlivé enginy firmy schovávali jako duchovní vlastnictví a měli k němu přístup jen tvůrci enginu a samotná firma. Změna přišla až s enginem Doom, který byl licencován a ostatní firmy tak měli přístup k již hotovému enginu, čímž nemuseli vyvíjet vlastní a ušetřili tak čas i peníze za tvorbu vlastního enginu. Tak se mohli věnovat více tvorbě obsahu do hry.

Dnes je většina enginu je napsána v nižším programovacím jazyku (např. C/C++), který poskytuje velkou rychlost. Pro psaní obsahu se využívá vyšších programovacích jazyků či skriptovacích jazyků (Lua, Python atd.), jelikož zde už není potřeba takové rychlosti. Pro popis herních enginů bylo čerpáno z [1].

V této práci budu popisovat grafický engine[1] Ogre 3D, jež patří mezi velice oblíbené a jedná se o dobrý nástroj k tvorbě her a aplikací využívající 2D a 3D grafiku. Samotný popis a vlastnosti enginu lze nalézt v kapitole 2 a v následující kapitole 3 je vytvořeno několik příkladů schopností enginu, včetně toho jak jich docílit.

V druhé části práce (kapitola 4) popisuji stereoskopické zobrazování a jak lze využít engine Ogre ke stereoskopickému zobrazování scény za použití „stereo vision managera". Stereoskopické zobrazování se snaží navodit pocit vnímání hloubky i u plochých obrazů a využívá k tomu schopnost lidského mozku. Stereoskopické zobrazování dnes proniká do všech možných oblastí našeho života, vznikají filmy s podporou promítání v moderních kinech (jako je IMAX), na trhu jsou dostupné 3D televizory, tvůrci her implementují stereoskopické zobrazování do svých her (či pomocí integrace podpory do grafických ovladačů v případě firmy NVIDIA) a nedávno se dokonce v Americe objevila první 3D reklamní výloha[28]. Detailnější popis stereoskopie je uveden v 4.1.

V kapitole 5 je popsána složitější scéna za použití několika doplňků, které nepatří přímo do Ogre, a popis problémů se spojením těchto doplňků a stereoskopického zobrazování.

[1] Grafický engine poskytuje jen nástroje na tvorbu grafické části programu, kdežto herní enginy pro tvorbu celých her. Grafický engine je vlastně jedna z částí herního enginu.

Kapitola 2

Ogre3D

2.1 Obecné

Ogre je navržen jako plně objektově-orientovaný engine, což umožňuje jednoduché rozšiřování objektů přes dědičnost a snadnou tvorbu pluginů (rozšíření). Při jeho návrhu se počítalo i s možností použití Direct3D, OpenGL, Glide a dalších (lze přidat v podobě pluginů). Nespornou výhodou pro programátora je, že některé části renderování, např. problémy s průhledností objektů, jsou řešeny na pozadí enginu a programátor se nemusí o ně starat. Samozřejmostí je kompletní online dokumentace veškerých tříd enginu i ostatních věcí.

Pro definici materiálů používá vlastní formát souborů (viz 2.3.3). Umožňuje použití vertex a fragment shaderů napsaných v assembleru i ve vyšších jazycích jako jsou Cg, DirectX9 HLSL nebo OpenGL GLSL. Dovede načítat textury z formátů PNG, JPEG, TGA, BMP, nebo DDS a používat různé druhy typů textur (1D, 2D i 3D).

Pro ukládání modelů využívá vlastního formátu MESH, který podporuje animaci kostry, LODy a další. Existuje plno programů popř. pluginů pro export do formátu MESH.

Pro správu scény používá několik různých správců (samozřejmě s možností rozšiřitelnosti), díky čemuž si může programátor zvolit nejvhodnější pro jeho aplikaci. Struktura scény je navržena v podobě stromu, který podporuje závislosti mezi objekty scény.

Mezi speciální podporované efekty patří podpora částicových systémů, zobrazení nebe za pomoci SkyBoxu, SkyPlane nebo SkyDome, podpora billboardingu a další. Mezi neméně důležité funkce se řadí jednoduchá správa zdrojů, která je schopná načtení používaných zdrojů v případě potřeby do paměti a to i z archivů (ZIP, PK3), rozšiřitelnost o pluginy bez nutnosti překladu enginu a mnoho dalších.

Ogre je vydán pod licencí LGPL. Pro ty co chtějí použít Ogre jako základ pro svůj software a nechtějí zveřejňovat zdrojové kódy, je možné získat Ogre pod licencí OUL (OGRE „Unrestricted" License). Pod touto licencí je už nutno za Ogre zaplatit.

2.1.1 Alternativy

Celkové porovnání schopností jednotlivých alternativních enginů vůči Ogre lze nalézt v tabulce 2.1. Následující texty u enginů jsou z většiny převzaty z oficiálních stránek Irrlichtu[16] a delta3d[17].

Irrlicht

Irrlicht je multiplatformní 3D engine napsaný v jazyce C++. Hodí se ke kompletní tvorbě 3D a 2D aplikací jako jsou hry nebo vědecké vizualizace. Engine je velice dobře dokumentovaný. Podporuje mnoho funkcí jako jsou např. dynamické stíny, částicové systémy, animace postav a detekce kolizí. Dle autorů je programování přístupné přes velice dobře navržené C++ rozhraní, které se velice jednoduše používá.

Velikou výhodu oproti Ogre má nejspíše v podpoře načtení velkého množství druhů modelů (Ogre umí načíst jen svůj vlastní formát MESH). Taktéž má oficiální podporu pro psaní v .NETu (dokumentace je pro C++ a C#).

Delta3d

Delta3D je open-source engine, který může být použit pro hry, simulace nebo jiné grafické aplikace. Jeho modulární návrh integruje velice dobře známe open-source projekty jako je Open Scene Graph (OSG), Open Dynamics Engine (ODE), Character Animation Library (CAL3D) a OpenAL. Pro renderování využívá OpenSceneGraph a OpenGL. Hlavním cílem Delta3D je poskytnutí jednoduchého a flexibilního API se základními elementy potřebnými pro všechny vizualizační aplikace.

Stejně jako Irrlicht (viz 2.1.1) podporuje více druhů modelů. Existuje několik rozšíření, které upravují engine na lepší podporu např. pro psaní v jiném programovacím jazyce, nebo lepší podporu pro vytváření her („dtGame").

Vlastnost	Ogre	Irrlicht	delta3d
Renderer	DirectX, OpenGL	DirectX, OpenGL	OSG, OpenGL
Multiplatformnost	Ano	Ano	Ano
Detekce kolizí	Částečná	Ano	Ano
Dynamická světla	Ano	Ano	Ano
Dynamické stíny	Ano	Ano	Ne
Billboardy	Ano	Ano	Ano
SkyBox	Ano	Ano	Ano
Částicové systémy	Ano	Ano	Ano
Shadery	Ano	Ano	Ano
Pluginy	Ano	Ne	jen pro dtGame
Licence	LGPL / OUL	ZLib	LGPL

Tabulka 2.1: Tabulka schopností některých grafických enginů.

2.2 Architektura

Ogre je jen grafický engine a ne herní engine, takže sám o sobě podporuje jen grafické operace. Pro další typy funkcí, jako jsou fyzikální výpočty, grafické uživatelské rozhraní, zvuky, umělá inteligence, síťování, vstupy a detekce kolizí, používá externí knihovny (většinou přístupné přes wrappery, viz 2.4).

Následující popis enginu je převzat z [4].

2.2.1 Jednoduchý pohled

V diagramu na obrázku A.12 je vidět, že celý engine je postaven na třídě Root, která má přístup k třídám `Manager`, jež ji umožňují přístup do různých podsystémů enginu.

Abstraktní třída `RenderSystem` dovoluje enginu použít specifickou implementaci pro renderování - Direct3D nebo OpenGL.

Třída `ResourceManager` je taktéž abstraktní a dědí se od ní další třídy, které poskytují enginu přístup k rozdílným typům zdrojů jako jsou textury, materiály a písma.

Všechny třídy, kromě `RenderSystem`, `SceneManager` a `RenderWindow` používají návrhový vzor (template) `Singleton`, která zajišťuje, že existuje jen jedna instance dané třídy a tím lze přistupovat k této jediné instanci odkudkoliv přes statickou metodu.

2.2.2 Renderovací cyklus

Na obrázku A.13 je vidět jak probíhá renderování jednoho snímku v Ogre. Renderování snímku začíná na straně samotné aplikace, která žádá třídu `Root` o rendering aktuální scény (pravidelné renderování lze přenechat i na třídě `Root` zavoláním metody `startRendering`). Třída `Root` následně aktualizuje všechny „Render Target" za pomoci třídy `RenderSystem` (což je DirectX nebo OpenGL). Podle nastaveného viewportu a aktivní kamery se provede do „Render Target" vykreslení scény. Samotný proces vykreslování jednotlivých objektů provádí třída `SceneManager`, která uchovává veškeré informace o scéně a jejích objektech. Při vykreslování objektu, jež je potomek třídy `Renderable`, se získá struktura `RenderOperation`, která uchovává samotná data potřebná pro vykreslení objektu, jako je seznam vertexů, indexů a způsob jakým mají být dané vertexy vykresleny.

2.2.3 Rozšíření o pluginy

Jelikož Ogre podporuje v základu jen grafické funkce, tak je nucen ostatní řešit pomocí pluginů. Pluginy můžou být k enginu linkovány buď staticky nebo dynamicky. Každý dynamicky linkovaný plugin musí být potomkem třídy `Plugin`[6], jehož metody využívá k instalaci pluginu do enginu. Plugin se ale musí sám zaregistrovat enginu co vlastně představuje, zda renderovací systém či správce scény apod.

2.3 Schopnosti

2.3.1 Stromová struktura scény

Pro uchování objektů ve scéně je v Ogre využívána stromová struktura. Ta umožňuje vytvořit závislosti některých objektů na druhých. Je možno tak posunout, či otočit kořen stromu a celá operace se provede i na celý podstrom. Hlavní kořen lze získat ze správce scény a hlouběji do stromu je nutno procházet ručně. Jednotlivé objekty lze připojovat k požadovanému kořeni a podle daného kořene se na objekt aplikují transformace.

Pro správu scény lze použít i jiné správce, než je defaultní, kteří umožňují využití dalších algoritmů pro zpracování scény jako je např. OctTree, Terrain (načtení terénu z externího souboru) nebo BSP (speciální formát vytvořený firmou Id software).

2.3.2 Listenery

Listenery jsou zajímavé řešení reakcí na události v enginu. Dovolují externí reakci na události uvnitř enginu bez nutnosti přístupu do části, kde se daná událost stala. Listener je vlastně třída, která má specifické metody (je potomkem nějaké třídy typu Listener[1]), které jsou při dané události volány. Pro programátora je asi nejdůležitější třída `Ogre::FrameListener`, která reaguje na renderování každého snímku. Pomocí této třídy lze provádět operace závislé na čase při každém snímku. Další typy listenerů jsou spíše použity uvnitř enginu či na složitější operace.

2.3.3 Materiály

Ogre využívá vlastní formát definice materiálů. Tento formát je velice podobný formátu shader programu napsaném v DirectX HLSL. Každý materiál má vlastní techniku a ta zas minimálně jeden průchod, kde se definují jednotlivé operace materiálu. Následující kód ukazuje jak se definuje jednoduchý materiál pro namapování textury ze souboru.

```
1  // definice materiálu
2  material TextureMap
3  {
4    // definice techniky
5    technique
6    {
7      // definice průchodu
8      pass
9      {
10       // nastavení texturovací jednotky
11       texture_unit
12       {
13         // použitá textura
14         texture image.png
15         // meřítko textury
16         // v tomto případě na stejné ploše bude
17         // textura zobrazena 10x
18         scale 0.1 0.1
19       }
20     }
21   }
22 }
```

Jak je z kódu poznat, je definování materiálu jen pro namapování textury vcelku složité, ale složitější materiály se lze tímto způsobem napsat mnohem snáze.

Materiálu lze nastavit i chování vůči odrazu a odlesku a to hodnotami uvedenými v části pass.

```
1  // ambientní světlo
2  ambient 1.0 1.0 1.0
```

[1] O implementaci rozhraní v C++ nelze hovořit, jelikož v něm neexistují rozhraní, jen abstraktní třídy.

```
3
4  // difůzní barva
5  diffuse 1.0 1.0 1.0
6
7  // barva a síla odlesku
8  // 3 hodnoty pro určení barvy
9  // a čtvrtá pro sílu odlesku
10 specular 1.0 1.0 1.0 9800.0
11
12 // vyzařovaná barva
13 // první 3 hodnoty jsou pro složky barvy
14 // a čtvrta nepovinná je pro průhlednost
15 emissive 0.0 0.0 0.0 1
```

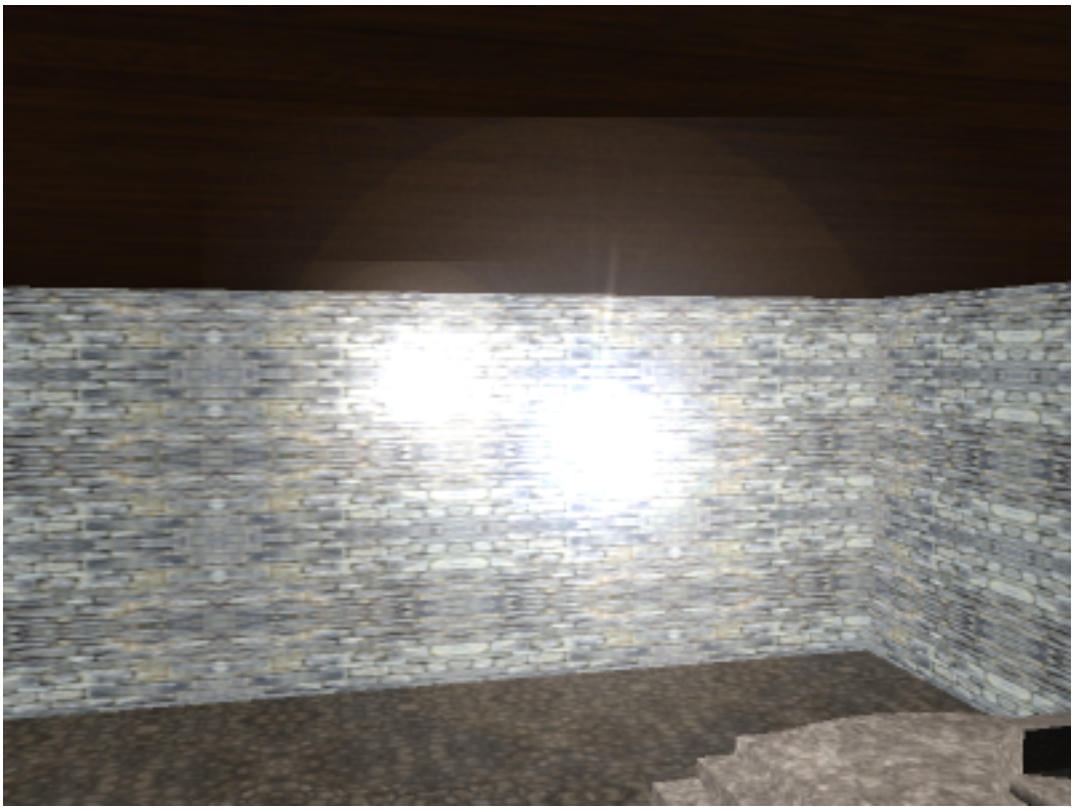
Přidání shaderů pro materiál se provádí následujícím kódem.

```
1  // definice vertex shaderu pro použití v materiálech
2  // tento shader je napsan v CG
3  vertex_program VertexShader cg
4  {
5      // zdrojový soubor shaderu
6      source VertexShader.cg
7      // vstupní bod pro program shaderu
8      // (hlavní spouštěná funkce)
9      entry_point main_vp
10     // nastavení pro překlad
11     profiles vs_1_1 arbvp1
12 }
13
14 // tato čast se nachází v části pass u materiálu
15 // umožnuje zvolenému shaderu nastavit požadované parametry
16 vertex_program_ref VertexShader
17 {
18     // zde je videt nastavení transformačních matic
19     param_named_auto worldViewProj worldviewproj_matrix
20     // a ambientního světla
21     param_named_auto ambient ambient_light_colour
22 }
```

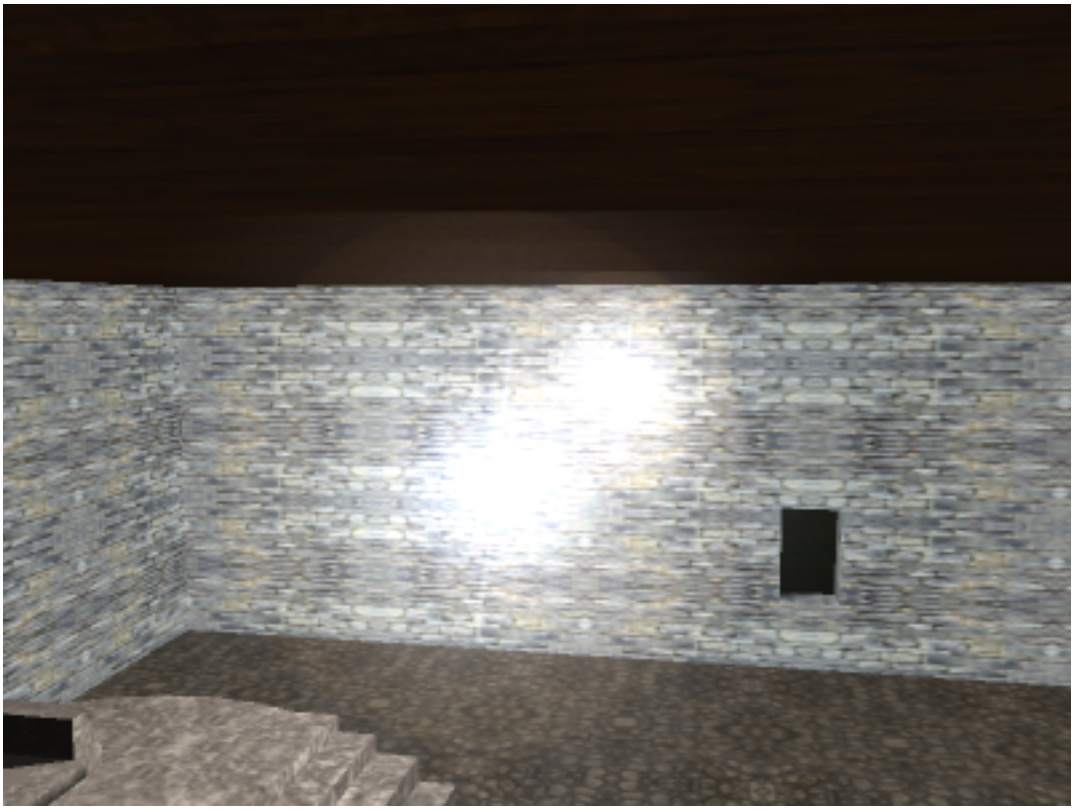
Popis jednoduchých i složitějších materiálů je uveden v [15] a výsledek použití některých složitějších materiálů lze nalézt v 3.2.4, 3.2.5 a 3.2.6.

2.3.4 Automatická správa průhledných objektů

O tuto část se Ogre stará úplně automaticky a není třeba nic zapínat. Lze to jen vypnout u materiálu. Na obrázcích 2.2 a 2.2 je vidět jak si Ogre poradí se dvěma billboardy. Na každém obrázku je pohled od jednoho billboardu na druhý a je na nich vidět, že nedochází k chybám při renderování (pořadí vykreslování objektů za použití z-bufferu, kdy později vykreslený vzdálenější objekt není za průhledným objektem vidět).



Obrázek 2.1: Pohled na 2 průhledné billboardy z jedné strany



Obrázek 2.2: Pohled na 2 průhledné billboardy z druhé strany

2.4 Wrappery

Jelikož Ogre je psáno v C++, což může být pro někoho obtížný jazyk, který ale zajišťuje vysokou rychlost kódu (jedná se o nativní kód), objevuje se snaha o možnost programování i v jednodušších programovacích jazycích. Například interpretované jazyky jsou mnohem jednoduší a umožňují automatickou správu paměti. Existují dva způsoby jak programovat Ogre pod těmito jazyky a jedním z nich je kompletní přepsání Ogre do daného jazyka a nebo možnost využití tzv. wrapperů. Tyto wrappery jsou mezivrstvou mezi daným jazykem a C++ jádrem Ogre. Wrapper v podstatě volá jen metody Ogre, které jsou napsány v C++.

Pro popis uvedených wrapperů byly použity informace z [7, 8, 9, 10, 11, 12, 13].

2.4.1 Jazyky

OgreDotNet: .NET

Wrapper pro .NET jazyky (C# a Visual Basic). V současné době je jeho vývoj pozastaven a je nahrazen níže zmíněným MOGRE.

MOGRE: .NET

Nebo-li Managed Ogre, je wrapper pro .NET jazyky verze 2.0. Mezivrstva mezi C++ a .NET je vytvořena za pomoci C++/CLI (interpretované C++ s trochu rozdílnou syntaxí). Upravuje některé zdrojové kódy Ogre, aby mohl poskytnout co nejmenší rozdíl mezi psaním v C++ a .NET.

Python-Ogre: Python

Wrapper pro Python. Jeho hlavní výhoda spočívá v tom, že není nutné překládat program, je dobře přenositelný a má spoustu dalších výhod spojených se skriptovacím jazykem. Nevýhodami je velká paměťová náročnost a nemožnost použití vláken.

Ogre4j: Java

Wrapper pro Javu. Pro přístup k rozhraním Ogre využívá Java Native Interface (JNI). Java je taktéž jako .NET interpretovaný jazyk s automatickou správou paměti a Garbage Collectorem.

Lugre: Lua

Wrapper pro skriptovací jazyk Lua. Citace z oficiálních stránek Lua[14]: "Lua kombinuje jednoduchou procedurální syntaxi s mocným popisem dat založeném na asociativních polích a rozšiřitelné sémantice. Lua dynamicky typuje proměnné, běží interpretována v byte kódu na virtuálním stroji, má automatickou správu paměti s garbage collectorem a je stvořena pro jednoduchou konfiguraci, skriptování a rychlé prototypování."[2]

[2]Citace je volným překladem, proto nemusí některé části být správně přeloženy.

2.4.2 Fyzika

Tyto i další zmíněné nejsou wrappery přímo pro Ogre, ale umožňují použití externích knihoven pro operace, které Ogre jako grafický engine neumí.

NxOgre: PhysX

Wrapper pro použití NVIDIA PhysX instrukcí přímo v Ogre. Dovoluje přístup ke složitým fyzikálním výpočtům s možností akcelerace na grafických kartách NVIDIA (přes rozhraní CUDA) i na fyzikálních akcelerátorech Ageia PhysX.

OgreODE: ODE

Velice dobře použitelný fyzikální engine pro Ogre. Poskytuje jednoduché fyzikální výpočty i některé složitější jako je např. fyzika postav. Jedná se o nejoblíbenější nástroj pro programování fyziky pod Ogre. Bohužel v současné době není v moc použitelném stavu, jelikož poslední stabilní verze je určena pro starší verzi OGRE.

OgreBullet: Bullet

Bullet jsou knihovny pro 3D detekci kolizí a dynamiku pevných těles pro hry a animace. Zdarma pro komerční použití pod licencí ZLib. Je integrován do Blender 3D a COLLADA 1.4. Byl použit mnoha profesionálními společnostmi v AAA titulech pro Playstation3, XBox 360, Wii a PC.

OgreNewt: Newton

Newton Game Dynamics je jednoduchá fyzikální knihovna. Obsahuje jenom malé množství funkcí, ale díky tomu je rychlá, stabilní (málo kódu, kde lze udělat chybu) a jednoduše se používá.

MogreNewt: Newton

C++/CLI port OgreNewt, což v podstatě umožňuje využití funkcí knihoven Newton v .NET programovacích jazycích pod MOGRE.

2.4.3 Vstup

OIS (Object Oriented Input System)

Pro správu vstupů používá Ogre od verze 1.4 (Eihort) defaultně OIS, které je dodáváno přímo v SDK. Pracuje na principu registrace Listenerů pro zvolený typ vstupu (myš, klávesnice atd.) a následném voláním jeho metod při události na straně zařízení. Podporuje oba režimy reakcí na vstupy - okamžitý (aktuální stav vstupního zařízení) i bufferovaný (ukládá do bufferu změny stavu na vstupním zařízení).

MOIS

Jedná se o MOGRE wrapper pro OIS.

2.4.4 GUI

CEGUI

S enginem dodávaný wrapper pro správu grafických rozhraní - autoři Ogre napsali vlastní renderer pro použití v Ogre. Psaní vlastních rozhraní je řešeno přes velice dobře strukturované XML soubory.

2.5 Instalace SDK

Jako multi-platformní engine umožňuje Ogre instalaci na různé operační systémy a použitý v různých vývojových prostředích. Zde uvedu jen instalaci pod Windows pro programování pod Microsoft Visual Studio 2008. Návod na instalaci pro ostatní operační systémy a vývojová prostředí lze nalézt na Ogre Wiki[3].

1. Na stránkách http://www.ogre3d.org v sekci "Download" vybrat "Download a Prebuilt SDKs."
2. Stáhnout požadované SDK.
3. Spustit instalátor a následovat instrukce.
4. Dále není potřeba nic nastavovat. Jen je možné, že se do systémových cest nezapíše proměnná OGRE_HOME, která obsahuje adresu k SDK.

Sestavení ze zdrojových kódů.

Jelikož je Ogre open-source, tak je možnost i vlastního sestavení za pomoci aktuálních zdrojových kódů. Ogre umožňuje překlad pod mnoha vývojovými prostředími už s předpřipravenými skripty na překlad, čímž odpadá nutnost nastavování skriptu pro překlad. Jak se provádí překlad a sestavení zde uvádět nebudu, jelikož díky SDK není nutný. Jak provést překlad pro další podporované platformy je uvedeno na Ogre Wiki[3].

Kapitola 3

Ukázky schopností enginu

3.1 Návrh aplikace

Návrh ukázkové aplikace vychází v základu ze standardního návrhu ukázkových aplikací Ogre. V těchto případech celý program ovládá jedna třída, která se stará o správu enginu a dalších věcí potřebných k běhu aplikace.

Ve funkci `main` se vytváří jen jedna statická instance třídy `BP::Application`, která provede nastavení a spuštění aplikace. Dále se zde odchytávají výjimky vyhozené Ogre v případě chyby. Téměř celý kód funkce `main` je převzat z tutoriálů Ogre, jelikož se jedná o pořád stejnou část kódu.

Kvůli načítání scény z XML souboru jsem si vytvořil třídu `BP::SceneLoader`. Tato třída využívá XML parser TinyXml[18].

Dále jsou v programu použity další rozšíření pro podporu stereoskopie (stereo vision manager) a dalších částí potřebných pro použité scény.

3.1.1 Objektový návrh

Diagram tříd je zobrazen na obrázku A.14 (Vygenerovaný pomocí Microsoft Visual Studio 2008).

Třída Application

Hlavní třída, která se stará o vše v programu. Její jediná instance je vytvořena v metodě main, odkud je taky spuštěn celý proces přípravy enginu a spuštění renderování scény.

V následujících 11 bodech je popsán postup jak třída připravuje engine až do spuštění renderování scény.

1. Vytvoření instance Ogre::Root
2. Načtení seznamu zdrojů ze souboru zdrojů
3. Načtení informací o scénách ze souboru scén
4. Načtení konfigurace pro stereo
5. Zobrazení konfiguračního dialogu (pokud není konfigurační soubor předán přímo přes příkazovou řádku)

6. Načtení zdrojů
7. Načtení XML souboru se scénou a její vytvoření
8. Inicializace stereoskopického zobrazování na základě volby v konfiguraci
9. Inicializace vstupních zařízení
10. Registrace listenerů
11. Spuštění renderingu scény

Třída ControlListener

Potomek třídy `Ogre::FrameListener` a `OIS::KeyListener` a má na starosti téměř veškeré ovládání programu (kromě pohybu kamery). Díky ní lze přepínat mezi typy stínů, měnit parametry stereoskopického zobrazování atd.

Třída InputManager

Nadstavba OIS (viz 2.4.3) vstupu kvůli jednotnému a snazšímu ovládání vstupů.

Třída LoadingBar

Třída vycházející ze třídy `Ogre::ExampleLoadingBar`, která byla upravena pro lepší použití v programu (ve výsledku se liší jen minimálně).

Třída MoveListener

O pohyb kamery (pozorovatele) se stará tato třída. Navíc ještě počítá kolize s okolními objekty za pomoci MOC[29] (Minimal Ogre Collision, `MOC::CollisionTools`).

Třída MultiheadListener

Tato třída slouží jako přepínač mezi okny programu u dualhead výstupu. Důvod jejího použití je v tom, že bez ní by se neaktivní okno neaktualizovalo (obnovuje se jen aktivní okno). Celá třída je převzata z ukázkového dema od „stereo vision managera" (viz 4.3.1).

Třída ObjectMoveListener

Za účelem možnosti animovat a pohybovat s některými objekty na scéně vznikla tato třída. Podobně jako třída `BP::MoveListener` se stará o pohyb a kolizi jen s tím rozdílem, že směr pohybu je generován náhodně při nárazu do nějaké překážky. Taktéž se stará o aktualizace animace objektu, pokud je objektu nějaká zadána.

Třída SceneLoader

Největší třída celého projektu. Dovede načíst celou scénu z XML souboru se scénou. Je velice elegantně navržena, proto není tak velký problém doprogramovat reakci na nově přidaný element do XML. To je řešeno mapou ukazatelů na metodu, která element zpracuje, a klíčem v mapě je název elementu. Při zpracovávání XML souboru se prochází struktura rekurzivně a pokud se narazí na nějaký element, tak se podle jeho názvu zavolá odpovídající metoda, jejíž ukazatel se získá z mapy ukazatelů. Proto tedy stačí jen napsat novou metodu na zpracování elementu a v konstruktoru třídy přidat ukazatel na danou metodu do mapy ukazatelů a třída se o ostatní sama postará. Tento způsob má ale i nevýhodu v tom, že metody musí mít určitý tvar a pro předávané objekty neexistuje žádná společná rodičovská třída. Proto je jediným možným způsobem předávat rodičovský objekt jako `void*` a v metodě ho ručně přetypovávat s kontrolou, zda jde vůbec o správný objekt.

Formát XML souboru je inspirován formátem prvních verzí editoru Ogitor[5] (v té obě se ještě jmenoval „scene loader").

3.1.2 Použitá data

Ve scénách jsou použity standardní modely, textury, shadery apod. dodávané spolu s enginem Ogre. Jen některé modely a materiály jsem vytvořil na demonstraci schopností enginu. Model druhé místnosti je optimalizovaná verze místnosti vytvořené kolegou Přemyslem Jarošem. Pro virtuální svět (viz kapitola 5) byly použity zdroje z výborného editoru Ogitor[5].

3.1.3 Vlastní scény

Program byl navržen tak, aby se dali scény do něj jednoduše vytvářet pomocí XML souborů. Návod na vytvoření vlastní scény a na to jak jí přidat do programu lze nalézt v dodatku D.

3.2 Použité efekty

V programu jsem použil jen některé efekty, které Ogre umí zobrazit. Většinou jde jen o jednoduché efekty, jelikož některé složitější se dají hůře vytvořit.

Ze zde uvedených je asi největší problém vytvoření efektu normal mapping (viz 3.2.5) a parallax mapping (viz 3.2.6), které potřebují ke svému správnému fungování mít u modelu správně spočítané tangenty (případně binormály), jinak se efekty vůbec neprojeví.

3.2.1 Vrhané stíny

Ogre dovoluje zapnutí několika druhů stínů a u každého vykreslovaného objektu lze nastavit, zda bude vrhat stíny, či ne. Příjemcem stínů je jakýkoliv objekt. V případě použití texture stínu je příjemce jen objekt, který nevrhá stín. V ukázkové aplikaci je možnost přepínání mezi jednotlivými druhy stínů. Některé stíny se nemusí ve scéně správně zobrazit, jelikož jsou částečně závislé i na použitém materiálu, či rozvržení scény.

Následující kód ukazuje jak se nastaví správci scény, že ve scéně nebudou žádné stíny. Typ stínu lze nastavit pomocí metody `setShadowTechnique` nějaké instance třídy `Ogre::SceneManager`. Zadávaným parametrem je hodnota výčtového typu `Ogre::ShadowTechnique`. Na obrázku A.1 je vidět výsledek použití stínu typu Stencil Additive, který podporuje vrhání stínů z více světelných zdrojů.

```
1 Ogre::SceneManager* sceneManager = ...;
2
3 // provede nastavení zvolené techniky stínu
4 sceneManager -> setShadowTechnique(Ogre::SHADOWTYPE_NONE);
```

V následujícím kódu je uveden příklad jak povolit vrhání stínu objektem. Umožňuje to metoda `setCastShadows` třídy `Ogre::MovableObject`. Zapnutí stínu u objektu má vliv jen u některých potomků této třídy (např. třída `Ogre::Entity`, která v sobě uchovává samotný model, který může vrhat nějaký stín). V případě, že objekt má nastaveno, že nevrhá stíny, tak je automaticky označen za příjemce stínů (toto je hlavně vidět u stínů typu texture, kde nelze mapovat texturu stínu na objekt, který daný stín vrhá).

```
1 Ogre::MovableObject* movableObject = ...;
2
3 // objektu nastaví, že bude vrhat stíny
4 movableObject -> setCastShadows(true);
```

3.2.2 Billboard

Billboard je v aplikaci použit pro určení pozice světel ve scéně. Zobrazení billboardu na pozici světla lze provést nastavením hodnoty atributu „flare" na „true" u elementu „Light".

Tento kód předvádí jak lze vytvořit billboard pro světlo. Výsledek je vidět na obrázku A.2.

```
1  // Definice některých proměnných
2  Ogre::SceneManager* sceneManager = ...;
3  Ogre::SceneNode* node = ...;
4
5  // Vytvoří novou kolekci pro billboardy
6  // billboardy z této kolekce mají některé hodnoty společné
7  Ogre::BillboardSet* bbs = sceneManager -> createBillboardSet("
       BillboardFlare", 1);
8  // vytvoření billboardu na pozici (0, 0, 0)
9  bbs -> createBillboard(Ogre::Vector3::ZERO);
10 // nastavíme billboardu materiál
11 bbs -> setMaterialName("Examples/Flare");
12 // připojíme billboard. node určuje, kde se billboard zobrazí
13 node -> attachObject(bbs);
```

3.2.3 SkyBox

SkyBox je v podstatě krychle, ve které se nachází celá scéna a je na ní mapovaná textura s nebesy. Nevýhoda tohoto řešení je, že občas je poznat přechod mezi texturami na hraně krychle. Na obrázku A.3 je vidět SkyBox, který byl namapován

CubeMap texturou (kterou si Ogre sám vytvořil z šesti různých textur). SkyBox lze v programu přidat následujícím kódem.

```
// definice proměnných
Ogre::SceneManager* sceneManager = ...;

// zapnutí SkyBoxu
// prvním parametrem je to, ze SkyBox zapínáme
// druhý parametr je název materiálu skyboxu
// třetí je vzdálenost od středu (tj. velikost)
sceneManager -> setSkyBox(true, "SkyBox", 1000);
```

3.2.4 Multitexturing

Multitexturing lze provést nezávisle na programu za pomoci materiálu. Kód v B.1 reprezentuje vytvoření jedné výsledné textury ze dvou za použití třetí jako alfa mapy. Kód je s mírnou úpravou převzat z [15]. Jak je na uvedené stránce napsáno, tak je potřeba, aby byla textura pro alfa mapu byla uložena v 8-bit režimu (odstíny šedi), jinak nebude multitexturing fungovat. Výsledek je možno vidět na obrázku A.4.

3.2.5 Normal Mapping

Normal mapping je v Ogre prováděn za pomoci nastavení materiálu. Aby se byl správně zobrazen je potřeba mít u modelu spočítány správně tangenty a binormály, které se při normal mappingu využívají. Ty lze spočítat přímo programově následujícím kódem či je uložit přímo u Ogre modelu mesh za pomoci nástroje OgreMeshUpgrade s argumentem „-t".

```
unsigned short src, dest;

if (!pMesh -> suggestTangentVectorBuildParams(src, dest))
{
    pMesh -> buildTangentVectors(src, dest);
}
```

V programu je normal mapping použit na modelu Atény (viz obrázek A.5). Správný výsledek je vidět jen v některém případě, jelikož materiál dovede pracovat jen s jedním světlem a v případě, že ve scéně je více světel najednou dochází k přepínání mezi světly. Kód materiálu zde není nutné příliš zmiňovat, jelikož jde jen o načtení a nastavení externího shaderu.

3.2.6 Parallax Mapping

Jedná se o další způsob vytvoření dojmu větší detailnosti u jednoduchého modelu. Na rozdíl od normal mappingu dokáže vyvolat větší dojem prostorovosti. Kameny na stěně na obrázku A.6 vypadají jako kdyby vystupovaly do prostoru (hlavně v pravé části obrázku). Takového efektu se s normal mappingem docílit nedá. Nevýhodou této metody je trochu větší náročnost. Jak přesně funguje parallax mapping zde uvádět nebudu, není to cílem této práce. Pro ukázku efektu byl použit ukázkový materiál Ogre s názvem materiálu „Examples/OffsetMapping/Specular". Kód materiálu je jako u normal mappingu o načítání a nastavování externích shaderů.

3.2.7 Animace

Ogre má přímo v sobě integrovanou podporu animací modelů. Model musí mít jen vytvořenou kostru (další soubor s příponou „skeleton"). Tento soubor v sobě uchovává informace o kostře modelu a animacích. K jednotlivým animacím lze přistupovat podle názvu a dále jim nastavovat další parametry jako je nekonečné opakování animace, zapnutí samotné animace atd. Aby bylo možné pozorovat změnu animace, je nutné jí měnit podle času. To lze provádět pomocí nějaké instance třídy `Ogre::FrameListener`, kde se v jedné z metod přidává čas, který uběhl od posledního snímku, dané animaci. Tím vlastně dojde ke změně animace podle uběhnutého času.

Následující kód popisuje nastavení animace a její následné aktualizace podle uběhnutého času. Podle tohoto kódu je animován i robot na na obrázku A.7.

```
1  // entita obsahující animaci
2  Ogre::Entity* entity = ...;
3
4  // získání stavu požadované animace, v našem případě "Walk"
5  AnimationState* state = entity -> getAnimationState("Walk");
6  // animace se bude neustále opakovat
7  state -> setLoop(true);
8  // provedeme zapnutí animace
9  state -> setEnabled(true);
10
11 // další část kódu je nutno volat v metodě frameStarted
12 // nějakého FrameListeneru
13
14 // struktura evt je jediný parametr metody frameStarted
15 // tato metoda provede změnu animace vzhledem ke změne času
16 state -> addTime(evt.timeSinceLastFrame);
```

Kapitola 4

Možnosti enginu pro stereoskopické zobrazování

4.1 Co je to stereoskopie

Člověk je od přírody obdarován párem očí, které vnímají svět kolem. Každé oko vnímá svět z jiného úhlu a tím poskytuje mozku trochu jiný obraz než oko druhé a obrazy z obou očí si mozek spojí a člověk tak dovede vnímat hloubku (jak vzdálený je pozorovaný objekt). Samotné slovo „stereo" pochází z řečtiny a znamená „prostorový". Stereoskopie se vlastně zabývá tím, jak poskytnout člověku pocit prostorovosti i u plochých obrazů. Pro správné vnímání hloubky někdy postačí i monokulární podměty, podle kterých dokáže mozek rozeznat hloubku. Mezi tyto podměty patří např. perspektiva, relativní velikost objektů, vzájemná poloha objektů, osvětlení a stín, atmosférický vliv či pohybová paralaxa (při pohybu pozorovatele se u bližších objektů zdá, že se pohybují rychleji než vzdálenější).

Prostorové vnímání umožňuje přinést lepší zážitek při pozorování virtuálních světů, při zábavě (sledování filmů), ale i ve vědě, kdy člověk může lépe ovládat stroj na dálku s velikou přesností. Stereoskopie byla využita i za války, kdy pomocí kombinace špionážních snímků mohly být odhaleny maskované nepřátelské pozice. Pro vysvětlení stereoskopie bylo čerpáno z [19].

Dnes většina standardních zobrazovacích zařízení (jako je např. monitor u PC, televize atd.) zobrazuje obraz do roviny a neposkytuje pozorovateli možnost vnímání hloubky. Stereoskopické zobrazování umožňuje pomocí těchto zařízení poskytnout pozorovateli možnost vnímání prostoru a to tím, že poskytne každému oku trochu jiný obraz podle toho jak by oko vnímalo svět kolem. Když takto dostane každé oko jiný obraz, mozek obrazy zkombinuje a člověk vnímá obraz prostorově (tj. vnímá hloubku).

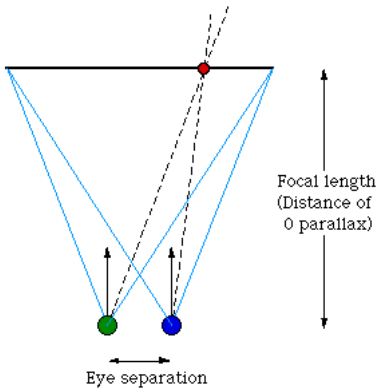
Největším problémem stereoskopického zobrazování je jak dostat ke každému oku ten správný obraz. Některé z technik, které toto řeší jsou popsány v 4.2. U některých technik může docházet k tzv. crosstalk, což znamená, že dochází k vzájemnému ovlivňování pravého a levého obrazu (člověk částečně vnímá pravým okem obraz určený pro levé oko).

Aby bylo možné vnímat hloubku obrazu, musí oba obrazy obsahovat tzv. paralaxu. Existuje několik druhů paralaxy podle níž bude pozorovaný objekt vnímán před, na nebo za rovinou displaye. Dále jsou definovány tzv. horizontální a vertikální paralaxa, ale lidské oči jsou stavěny jen na horizontální paralaxu, proto vše

následující bude spojováno jen s horizontální paralaxou.

Nulová paralaxa

Pozorovaný objekt s nulovou paralaxou bude pozorovatelem vnímán na rovině displaye (obr. 4.1).

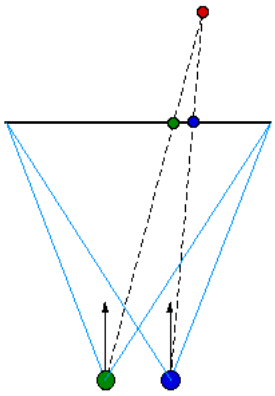


Obrázek 4.1: Nulová paralaxa (převzato z [27])

Pozitivní paralaxa

Při pozitivní paralaxe je objekt vnímán za rovinou displaye. Pro samotného pozorovatele je nejlepší, když většina objektů je zobrazována v pozitivní paralaxe (viz obrázek 4.2).

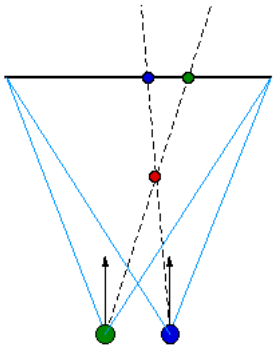
Existuje ještě divergentní pozitivní paralaxa, kdy se osy očí rozbíhají. To není kvůli stavbě samotným očí příliš dobré, jelikož na to nejsou zvyklé.



Obrázek 4.2: Pozitivní paralaxa(převzato z [27])

Negativní paralaxa

Opak pozitivní, kdy jsou objekty vnímány před rovinou displaye. Při této paralaxe často dochází k bolení očí, proto není dobré zobrazovat mnoho objektů pomocí této paralaxy. Velice efektivní je pomocí této paralaxy zobrazovat rychle se pohybující objekty, které budou na scéně zobrazeny jen krátkou dobu.



Obrázek 4.3: Negativní paralaxa (převzato z [27])

4.2 Techniky stereoskopického zobrazování

Pro popis následujících technik bylo převážně čerpáno z [20].

4.2.1 Anaglyf

Tato technika je založená na principu oddělování barev. Ke vnímání obrazu se využívá jednoduchých brýlí s barevnými filtry, kde je pro každé oko použit jiný barevný filtr, který propouští jen specifické barevné spektrum světla. Obrazy speciálně vytvořené pro tuto techniku se nazývají anaglyfy. Anaglyfy se vytvářejí spojením dvou obrazů, kdy je z každého obrazu použita jen část barevného spektra světla, která odpovídá použitému barevnému filtru na brýlích pro dané oko. Díky tomu bude první obraz propuštěn jen do prvního oka a druhý jen do druhého.

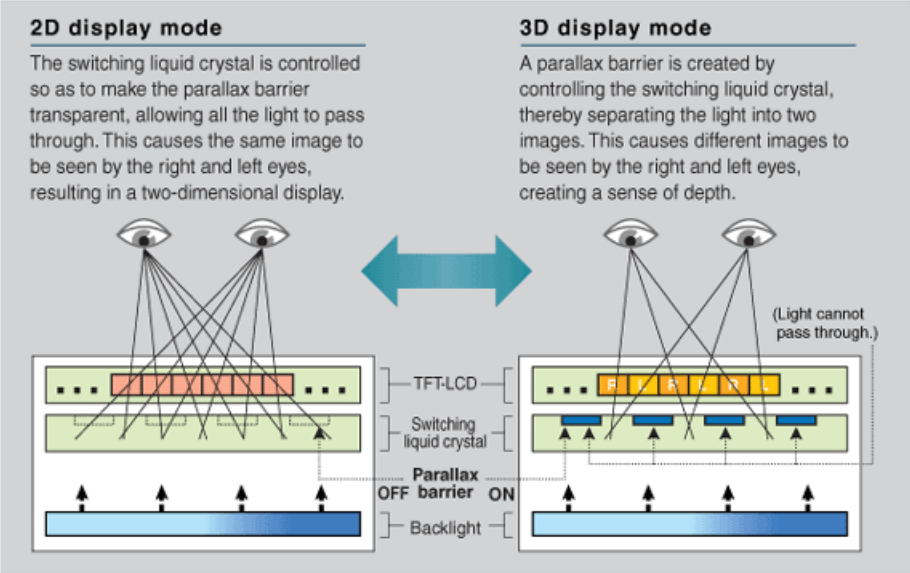
První anaglyfy využívali modré a červené barvy a byly určeny pro zobrazování obrazu ve stupních šedi. Pro zobrazení barevného obrazu se využívá i zelené barvy. Tím může vzniknout několik kombinací filtrů. Mezi nejčastěji používané patří červeno-azurový a žluto-modrý anaglyf (oba lze vyzkoušet v ukázkovém programu).

Z textu je patrné, že dochází u této techniky ke ztrátě barevné informace, což je jediná její nevýhoda. Na druhou stranu používané brýle jsou velice levné a výsledný obrázek lze vytisknout na papír a z něj lze také pomocí brýlí vnímat hloubku (technika tedy není závislá na zobrazovacím zařízení).

4.2.2 Polarizace

Světlo jakožto elektromagnetické vlnění kmitá ve všech směrech. Polarizací dostaneme jen světlo kmitající v jedné rovině. K tomu se právě využívají tzv. polarizační filtry. Této vlastnosti využíváme u této techniky, kdy obrazy pro každé oko necháme promítnout přes polarizační filtry, které jsou povětšinou posunuty navzájem o 90° (horizontální a vertikální) a promítány na stejné místo na jednom plátně. Ke vnímání stačí jen brýle s polarizačními filtry, které propustí jen ten obraz určený pro dané oko.

Oproti anaglyfu má tu výhodu, že nedochází ke ztrátě barevné informace. Bohužel toto řešení může být velice drahé, neboť je potřeba mít dva projektory s polarizačními filtry a jelikož je potřeba promítat na stejné místo, je nutné použití zrcadel. Aby nedocházelo k chybám v promítaném obrazu je nutno použít drahá zrcadla, která nemají odrazovou plochu překrytu sklem, protože by docházelo k lomu



Obrázek 4.4: Jak funguje Sharp 3D LCD (převzato z [21])

paprsku světla na přechodech vzduch-sklo a sklo-odrazová plocha a tím by vlastně daný paprsek světla dopadl na jiné místo promítacího plátna než by měl.

4.2.3 Quad-buffer

Podle popisu na stránce [24], je to schopnost OpenGL kreslit do pravého a levého předního a zadního bufferu nezávisle (celkem 4 buffery, proto tedy Quad). Jelikož jsou pravý a levý buffer oddělený, je možno přehazovat přední a zadní buffery v synchronizaci se zobrazovacím zařízením. Jak už jsem zmínil, je Quad-buffer doménou OpenGL. Pod DirectX je podobná technika nazvána Multihead a umožňuje renderovat do více než dvou nezávislých předních a zadních bufferů. Dle popisu multiheadu na [26] jde o vytvoření nezávislých swapchainů[1] se zobrazováním přes celou obrazovku.

4.2.4 Auto-stereoskopické zařízení - Sharp 3D LCD

Auto-stereoskopické mají tu vlastnost, že není potřeba žádných speciálních brýlí, či dalšího zařízení. Existuje mnoho typů auto-stereoskopických zařízení, ale zde popíši podrobněji jen tzv. Sharp 3D LCD[21].

Sharp 3D využívá tzv. parallax bariér, které propouštějí světlo jen přes určité body na obrazovce (obsahující střídavě pravý a levý obraz) do správného oka. Porovnání s 2D zobrazením je zobrazeno na obrázku 4.4.

[1]Swapchain je propojení zadního a předního bufferu (případně více bufferů) a stará se o jejich přehazování.

4.3 Implementace stereoskopického zobrazování

4.3.1 Stereo vision manager

Jelikož Ogre je známý a často používaný engine a stereoskopie není taky žádnou novinkou, proto tedy už někdo podporu stereoskopie do Ogre vytvořil. Tento „plugin" umožňuje být připojen do programu dvěma způsoby. Prvním z nich je připojení jako DLL knihovna tj. jako plugin Ogre. Druhou možností je přímé připojení do programu jako zdrojové kódy. Tuto možnost využívám i ve svém programu. Více se lze dozvědět na [23].

4.3.2 Realizace jednotlivých technik

Celý princip správce stereo spočívá v ukládání jednotlivých obrazů pro každé oko do dvou textur pro techniky zobrazující výsledek v jednom obrazu a nebo v případě dualhead outputu přímo do zadních bufferů. Pokud jde o rendering do textur, tak se výsledné textury spojují pomocí tzv. kompozitoru, který využívá na spojování textur externí shader. Celý manager je velice chytře vymyšlen, jelikož pro renderování stačí jediná kamera a ta je pro vykreslování obrazu pro pravé nebo levé oko posunuta doprava nebo doleva na základě nastavených parametrů a po vykreslení obrazu zase vrácena zpět na výchozí pozici (tj. mezi očima).

Zdrojové kódy Stereo vision manageru jsem mírně upravil pro potřeby programu, protože jsem potřeboval např. získat seznam podporovaných stereoskopických technik a nebo odkazy na viewporty, do kterých se renderují jednotlivé obrazy pro obě oka.

Další stereoskopickou techniku lze jednoduše přidat v konstruktoru třídy pomocí následujícího kódu (v kódu je přidávána technika Sharp 3D) a ještě nadefinovat správný materiál, pokud je potřeba, pro danou techniku.

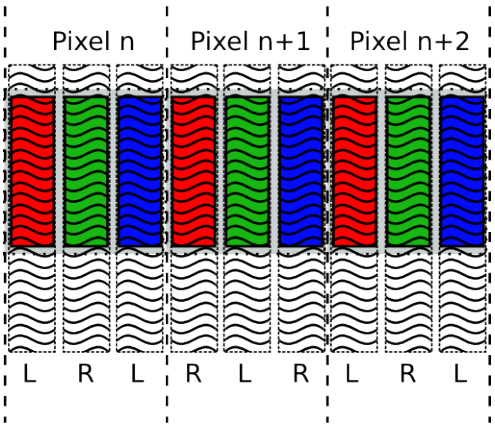
```
1  // Do seznamu dostupných technik přidáme popis nové techniky
2  // v konstruktoru StereoModeDescription se uvádí na prvním
3  // místě název techniky druhým parametrem je název materiálu
4  // použitého kompozitorem. Taktéž je dobré přidat danou
5  // techniku do výčtového typu StereoMode pro lepší přehled
6  // o dostupných technikách
7  // pokud nebude vyplněn druhý parametr, tak se ke kombinaci
8  // textur nebude využívat kompozitor.
9  mAvailableModes[SM_SHARP_3D] = StereoModeDescription(
10     "SHARP_3D", "Stereo/Sharp3D");
```

Anaglyf

Výsledný obraz je sestaven ze dvou textur za pomoci shaderu. Pro oba podporované typy anaglyfu je použit stejný shader a rozdíl je jen v parametrech (maska složek barvy pro každý z obou obrazů), se kterými byl shader přeložen. Celý kód shaderu lze nalézt v B.2 jako funkci „Anaglyph_fp".

Dual head output

Tato technika jako jediná využívá dvou viewportů (pro každé oko jeden). Na rozdíl od ostatních nepotřebuje žádný shader na sloučení textur, jelikož jednoduše obě



Obrázek 4.5: Rozdělení obrazů v technice Sharp 3D (převzato z [22])

textury zobrazí ve dvou viewportech.

Ogre samo o sobě nepodporuje zobrazování na více monitorech (zobrazovacích zařízeních) a bez toho nešlo uvést správně do chodu tuto techniku. Proto jsem musel upravit zdrojové kódy Ogre pomocí patche [25], který umožnil zobrazení na více monitorech. Patch byl bohužel určen pro starší vývojovou verzi Ogre, takže zde byl trochu problém s jeho aplikací. Nakonec se patch podařilo aplikovat na stable verzi 1.6.2. Upravené zdrojové kódy[2] jsou k nalezení na přiloženém datovém médiu.

Sharp 3D

Jak je vidět na obrázku 4.4, tak se obrazy pro pravé a levé oko pravidelně střídají. Bohužel zde není uvedeno, zda je to na úrovni pixelu či subpixelu. Jak přesně tuto techniku naimplementovat jsem čerpal z [22], kde je uvedeno, že se obrazy pro pravé a levé oko střídají na úrovni subpixelů.

Implementačně je tato technika velice podobná anaglyfu, kde se taktéž pomocí shaderu spojují dvě před-renderované textury do výsledného obrazu. Bohužel sousední pixely mají složení barvy z jiných textur, ale pixely ob jeden mají složení stejné. Proto stačí zjistit, zda v shaderu zpracovávaný pixel je sudý nebo lichý a podle toho určit výsledné složení pixelu. Výsledný shader je vlastně kombinací shaderu pro anaglyf (sloučení dvou pixelů do jednoho) a shaderu pro interlace (určení, zda jde o sudý nebo lichý pixel).

Tato technika není přímo zahrnuta ve stereo vision manageru, proto jsem jí musel doprogramovat. Materiál a shader použitý pro tuto techniku lze nalézt v dodatku pod B.2.2 a B.2.3.

[2]Do kódů je přidána i oprava chyby v patchi, kdy docházelo k vyhození výjimky i v případě, že vytvoření DirectX zařízení proběhlo v pořádku (a to jen u grafických karet bez podpory vertex shaderů). Chybu jsem nalezl při testování na svém notebooku s grafickou kartou Intel GMA950.

Kapitola 5

Virtuální svět

Návrh světa byl převzat z výborného editoru Ogitor[5], kde je scéna vytvořena jako ukázková. Jelikož je ale scéna v Ogitoru uložena v XML rozdílném formátu než je ten použit v ukázkovém programu (formát je inspirován první verzí Ogitoru, kdy se ještě jmenoval „scene loader"), tak jsem byl nucen přepsat tento formát do formátu použitelném v programu. Scéna Ogitoru využívá několika přídavných pluginů, z nichž využívám v programu jen některé (viz 5.1).

Výsledek viz obrázky A.8, A.9, A.10 a A.11.

5.1 Použitá rozšíření

Editable Terrain

Toto rozšíření slouží k načítání složitých terénů z textur. Dovoluje použití světelné mapy i mapy na mapování více textur na terén. Dokonce umožňuje programově upravovat terén, ale pro potřeby ukázkového programu nebylo nutné toto implementovat.

Kvůli tomuto rozšíření jsem byl nucen trochu upravit kolizní nástroje (třída `MOC::CollisionTools`), aby bylo možno získávat údaje o výšce terénu přímo v kolizním systému. Samotný MOC má podporu jen pro jednodušší verzi Editable Terrainu.

Hydrax

Velice výborný plugin na zobrazení reálné vodní hladiny. Bohužel při implementaci tohoto rozšíření jsem narazil na několik zásadních problémů, jež docela dost ovlivňovaly stereoskopické zobrazování. Hydrax sám o sobě podporuje jen zobrazení do jednoho viewportu a musí být aktualizován pomocí `Ogre::FrameListener`. Problém spočívá v tom, že stereo manažer posouvá kameru na požadovanou pozici jen při updatu viewportu a pak jí vrací zpět a při updatu Hydraxu už je kamera na původní pozici. Tento problém jsem částečně vyřešil tím, že jsem vytvořil jednu třídu, která je potomkem třídy `Ogre::FrameListener` a `Ogre::RenderTargetListener`. První umožňuje uložení času od posledního snímku a další je využit proto, že může být volán v době, kdy je kamera na správné pozici. Jelikož k registraci třídy jako `Ogre::RenderTargetListener` je nutno znát ukazatel na `Ogre::RenderTarget` nebo na `Ogre::Viewport`, tak jsem byl nucen upravit zdrojové kódy stereo vision

managera, abych mohl získat ukazatele na oba používané viewporty a přidat do nich daný listener.

Hydrax využívá k renderování vody několik druhů tzv. modulů. Já zvolil „ProjectedGrid", který renderuje vodu v závislosti na renderovací kameře a ořezává vodu podle pohledové kamery a tím eliminuje renderování zbytečných trojúhelníků, které nejsou vidět. Ke zjišťování využívá pohledovou matici kamery, kterou získá přímo od kamery. Bohužel při výpočtu matice se nezohledňuje tzv. frustum offset, který upravuje pohledovou pyramidu. Kvůli tomuto dochází k ořezávání i části vodní hladiny, která je vidět. Kvůli řešení jsem byl nucen upravit zdrojové kódy třídy `Hydrax::Module::ProjectedGrid`, kde jsem jen upravil hodnoty na získání čtyř souřadnic podle pohledové matice a to tím, že jsem posunul požadovaný bod mimo pohledovou matici. Touto opravou se sice renderuje více trojúhelníků, které ani nejsou vidět, ale při použití frustum offset nedochází k ořezávání viditelné hladiny vody. Bohužel při určité kombinaci ohniskové vzdálenosti a vzdálenosti očí může dojít k viditelnému ořezávání vodní hladiny. Toho se nelze zbavit, jen pomocí nastavení parametrů lze docílit, že to bude vidět jen ve výjimečném případě. Ten nastane jen, když bude ohnisková vzdálenost velice malá a vzdálenost očí velice veliká, ale to jsou hodnoty, kterých nelze při běžném užívání dosáhnout (jen při testování).

Dalším vážným problémem při použití Hydraxu a stereo vision manageru je ve špatném zobrazování odrazu na vodní hladině a části viditelné pod vodní hladinou při zapnuté stereo technice. Toto je způsobeno tím, že se textury pro odraz a lom světla (tzv. refraction) se počítají v době, kdy už stereo vision manager přesunul kameru zpět na původní pozici. Původně jsem si myslel, že vyřešit tento problém bude obtížné, ale nakonec to šlo jednoduše vyřešit přidáním ukazatelů na „Render Target" do stereo vision managera. Ten pak dané textury aktualizuje v době, kdy má kameru na správné pozici. Jak přidat závislost do stereo vision managera popisuje následující kód.

```
1  // vytvoření stereo vision managera
2  StereoManager* stereo_manager = ...;
3  // vytvoření render target, který bude stereo manager
       aktualizivat
4  Ogre::RenderTarget* render_target = ...;
5
6  // přidání závislosti
7  stereo_manager -> addRenderTargetDependency(render_target);
8  ...
9  // ještě je nutné závislost odebrat, jelikož by mohlo dojít
10 // v programu k chybě, kdy se bude stereo manager snažit volat
11 // neexistující render target
12 stereo_manager -> removeRenderTargetDependency(render_target);
```

Kapitola 6

Závěr

Ogre je velice dobrý grafický engine s množstvím funkcí, které ulehčují programátorovi práci. Při programování mě velice těšila automatická správa zdrojů, automatické uvolňování některých objektů enginu a integrovaná podpora pro stíny. Mezi velice zajímavé funkce bych zařadil dědičnost materiálů, což velice zjednodušuje práci, jen daný rodičovský materiál musí být uveden dříve než potomek, jinak dojde k chybě a Ogre materiál nenačte. Při tvorbě mi trochu vadilo, že Ogre dovede načíst modely jen formátu MESH, což ale kvůli návrhu a podporovaným funkcím chápu - naneštěstí existuje plno programů na převod do tohoto formátu. Při implementaci normal a parallax mappingu jsem narazil na podivné chování při zobrazování daného efektu. Nepodařilo se mi zjistit, proč se mi tyto efekty přestali správně zobrazovat, i když jsem provedl všechno správně pro jejich zobrazení (model musí mít vypočtený tangenty) a ani logovací soubor enginu nehlásil žádné chyby. Screenshoty uvedené u těchto efektů jsou proto z verze, kdy mi ještě tyto efekty fungovali správně. Mezi největší nevýhody bych zařadil rozsáhlost enginu, kdy prostudování veškerých možností enginu zabere mnoho času. Naneštěstí díky dobré online dokumentaci, přístupu ke zdrojovým kódům, veliké komunitě a rozsáhlému fóru, není problém nalézt odpověď na jakýkoliv problém.

Díky mnou vytvořené podpoře pro načítání scény z XML není problém vytvořit externě jakoukoliv scénu. Tímto lze načíst jen základní elementy scény, ale díky návrhu třídy není problém podporu na další elementy rozšířit.

Pro stereoskopické zobrazování není Ogre zrovna ideálně navrženo. Při vývoji se s touto možností ani nepočítalo a v nejbližší době ani nepočítá. Kvůli podpoře dualhead výstupu jsem byl nucen aplikovat na zdrojové kódy Ogre patch s podporou zobrazování na více zobrazovacích zařízeních, ale i tato samotná aplikace patche se nevyhnula problémům, jelikož patch je určen na v té době aktuální vývojovou verzi Ogre a já měl díky tomu problémy s nalezením vhodné stable verze, na kterou by šlo tento patch správně aplikovat. Díky „stereo vision manager" lze jednoduše přidat podporu pro stereoskopické zobrazování. Manager podporuje nejvíce známé techniky, jen podporu techniky Sharp 3D jsem musel dodělat. Jelikož je manager velice dobře navržen, tak přidání další techniky nebyl takový problém.

Literatura

[1] Jeff Ward. *What is a Game Engine?* [online]. 2008. <http://www.gamecareerguide.com/features/529/what_is_a_game_.php> [cit. 2009-02-21].

[2] *OGRE - Open Source 3D Graphics Engine* [online]. Ver. 1.6.2. <http://www.ogre3d.org> [cit. 2009-04-30].

[3] *Ogre Wiki* [online]. 2009. <http://www.ogre3d.org/wiki/> [cit. 2009-04-30].

[4] *Documentation Architecture - Ogre Wiki* [online]. Poslední změna 13.10.2008. <http://www.ogre3d.org/wiki/index.php/Documentation_Architecture> [cit. 2009-04-13].

[5] *Ogitor - Ogre Wiki* [online]. Poslední změna 1.5.2009. <http://www.ogre3d.org/wiki/index.php/Ogitor> [cit. 2009-05-01].

[6] *OGRE: Ogre::Plugin Class Reference* [online]. Poslední změna 5.2.2009. <http://www.ogre3d.org/docs/api/html/classOgre_1_1Plugin.html> [cit. 2009-03-16].

[7] *OgreDotNet - Ogre Wiki* [online]. Poslední změna 25.2.2009. http://www.ogre3d.org/wiki/index.php/OgreDotNet [cit. 2009-04-27].

[8] *MOGRE - Ogre Wiki* [online]. Poslední změna 21.4.2009. <http://www.ogre3d.org/wiki/index.php/MOGRE> [cit. 2009-04-26].

[9] *PyWiki* [online]. 2009. <http://wiki.python-ogre.org/index.php/Main_Page> [cit. 2009-03-11].

[10] *ogre4j - Java Native Interfaces for OGRE* [online]. 2008. <http://ogre4j.sourceforge.net/> [cit. 2009-03-12].

[11] *NxOgre - Ogre Wiki* [online]. Poslední změna 5.3.2009. http://www.ogre3d.org/wiki/index.php/NxOgre [cit. 2009-03-11].

[12] *OgreODE - Ogre Wiki* [online]. Poslední změna 24.10.2008. <http://www.ogre3d.org/wiki/index.php/OgreODE> [cit. 2009-03-06].

[13] *OgreBullet - Ogre Wiki* [online]. Poslední změna 18.10.2008. <http://www.ogre3d.org/wiki/index.php/OgreBullet> [cit. 2009-03-08].

[14] *The Programming Language Lua* [online]. Poslední změna 15.4.2009. <http://www.lua.org> [cit. 2009-04-29].

[15] *Materials - Ogre Wiki* [online]. Poslední změna 12.12.2008. <http://www.ogre3d.org/wiki/index.php/Materials> [cit. 2009-04-10].

[16] *Irrlicht Engine - A free open source 3d engine* [online]. Poslední změna 7.5.2009. <http://irrlicht.sourceforge.net> [cit. 2009-03-16].

[17] *Delta3D - Open source gaming & simulation engine* [online]. 2009. <http://www.delta3d.org/> [cit. 2009-03-15].

[18] *TinyXml* [online]. Poslední změna 24.1.2008. <http://www.grinninglizard.com/tinyxml/> [cit. 2009-04-20].

[19] P. Čížek. Prostorové zobrazování (DP ZČU). 2005.

[20] F. Mikšíček. Causes of Visual Fatigue and Its Improvements in Stereoscopy (DP ZČU). 2006.

[21] *About SHARP Technology: SHARP Electronic Components* [online]. Poslední změna 16.4.2008. <http://sharp-world.com/products/device/about/lcd/3d/index.html> [cit. 2009-04-30].

[22] Lukas Ahrenberg. *Programming graphics for Sharp LL-151-3D parallax barrier auto stereoscopic screens* [online]. Poslední změna 4.3.2009. <http://lukas.ahrenberg.se/archives/178> [cit. 2009-04-30].

[23] *Stereo vision manager* [online]. Poslední změna 2.3.2009. <http://www.ogre3d.org/forums/viewtopic.php?f=5&t=32284> [cit. 2009-04-22].

[24] *stereo geometry in OpenGL* [online]. Poslední změna 7.6.2008. <http://www.orthostereo.com/geometryopengl.html> [cit. 2009-05-01].

[25] *Multiple render windows patch - version 3* [online]. Poslední změna 10.3.2009. <http://sourceforge.net/tracker/?func=detail&aid=2677804&group_id=2997&atid=302997> [cit. 2009-04-30].

[26] Microsoft. *Multihead (Direct3D 9)* [online]. 2009. <http://msdn.microsoft.com/en-us/library/bb147217.aspx> [cit. 2009-05-05].

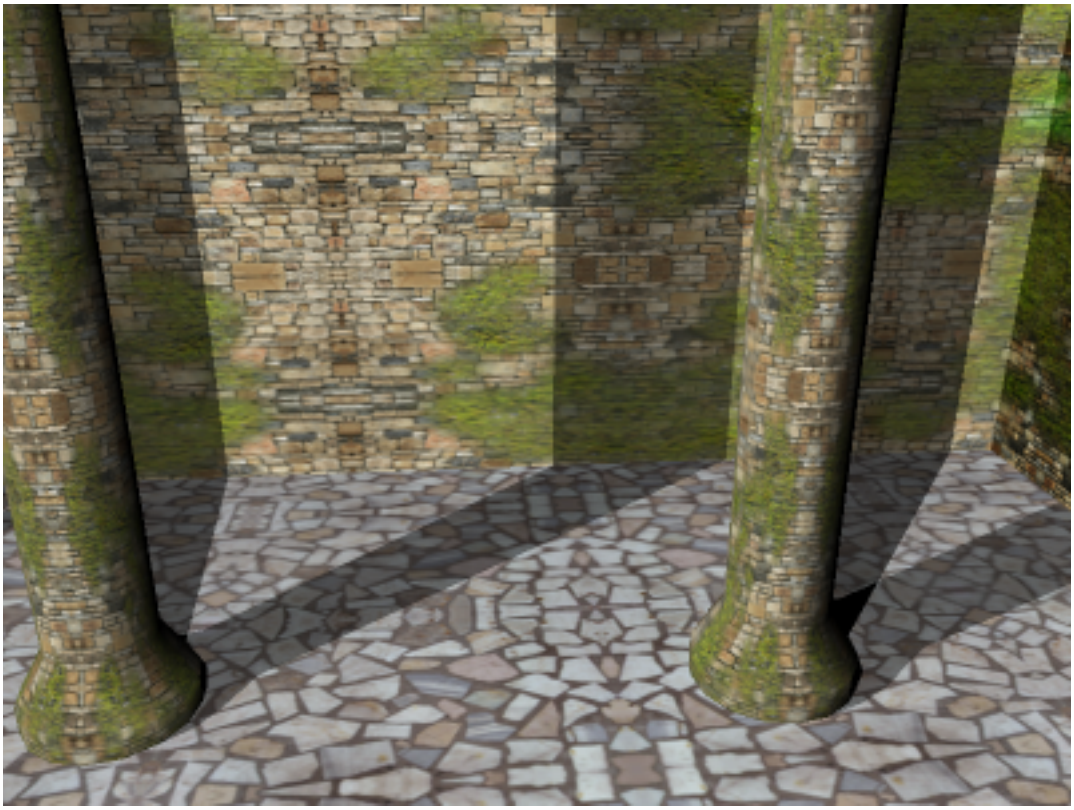
[27] *Stereographics* [online]. 2003. <http://local.wasp.uwa.edu.au/~pbourke/papers/HET409_2003/stereo.html> [cit. 2009-05-05].

[28] *No-glasses 3D LCD used for ads outside NY store* [online]. Poslední změna 12.5.2009. <http://www.tcmagazine.com/comments.php?shownews=26376&catid=2> [cit. 2009-05-12].

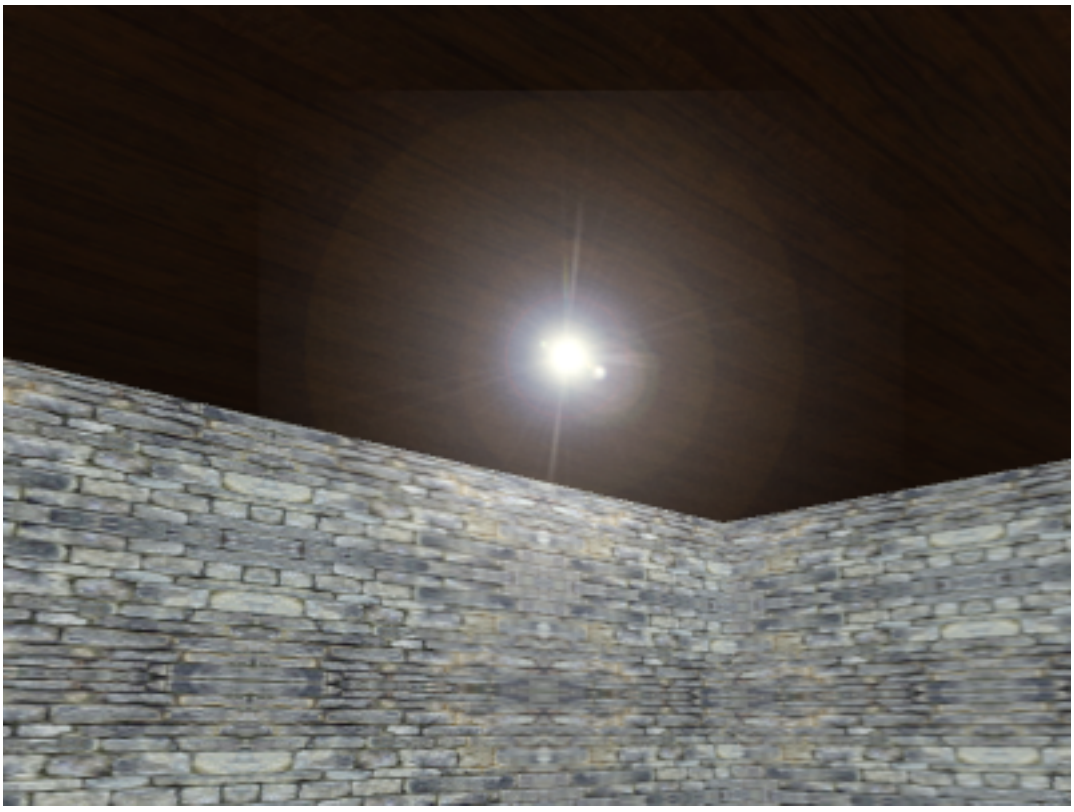
[29] *MOC - Minimal Ogre Collision - Lightweight collision detection.* 2009. http://www.artifexterra3d.com/moc-minimal-ogre-collision.php> [cit. 2009-05-02]

Dodatek A

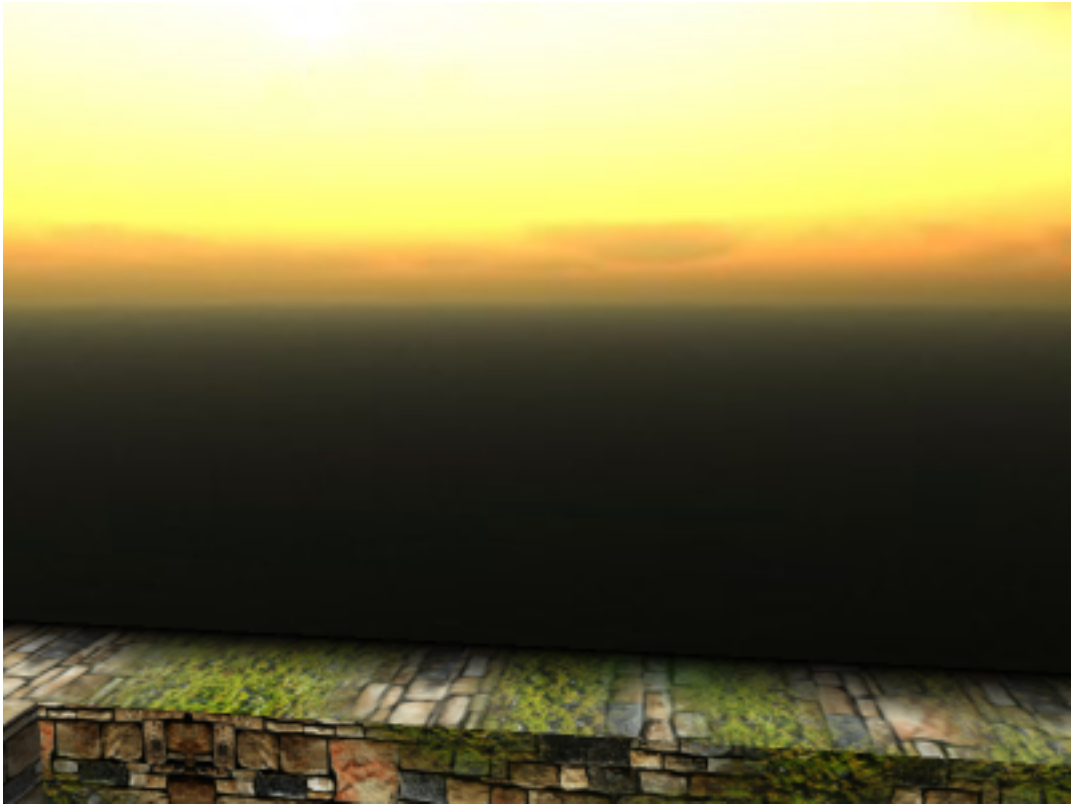
Obrázky



Obrázek A.1: Stíny pro více světel



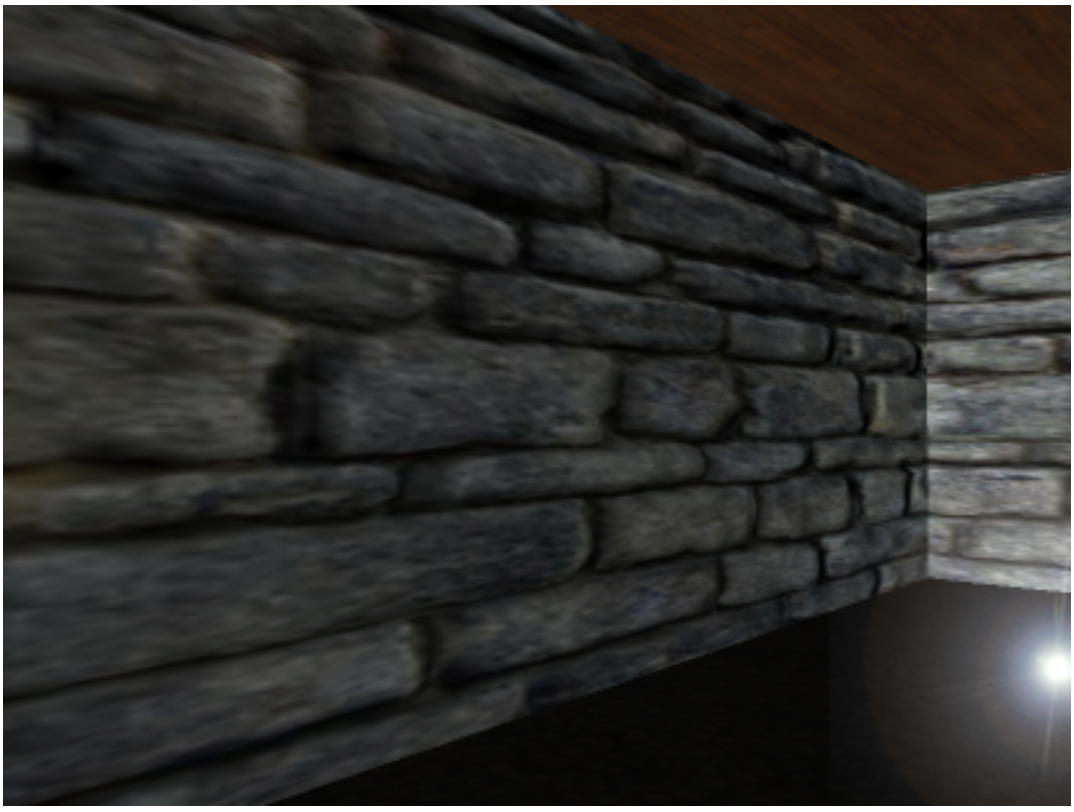
Obrázek A.2: Billboard záře na pozici světla



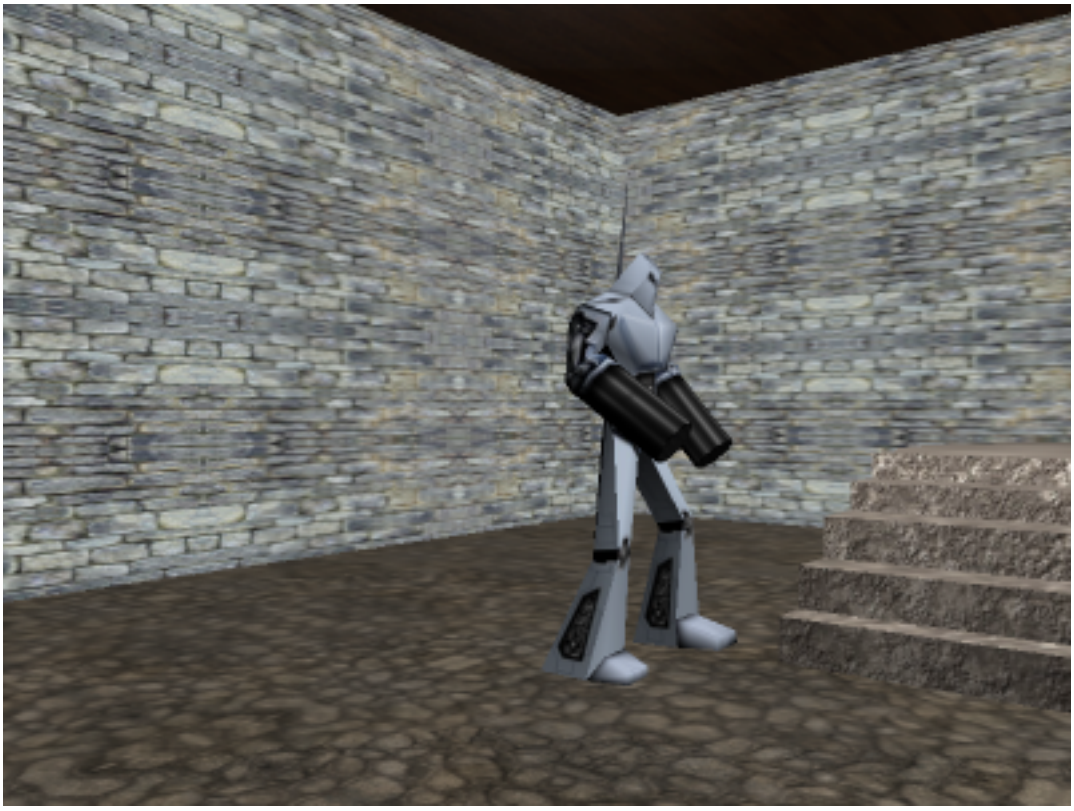
Obrázek A.3: SkyBox za použití CubeMap

Obrázek A.4: Výsledek multitexturingu

Obrázek A.5: Socha Atény s použitým Normal Mappingem



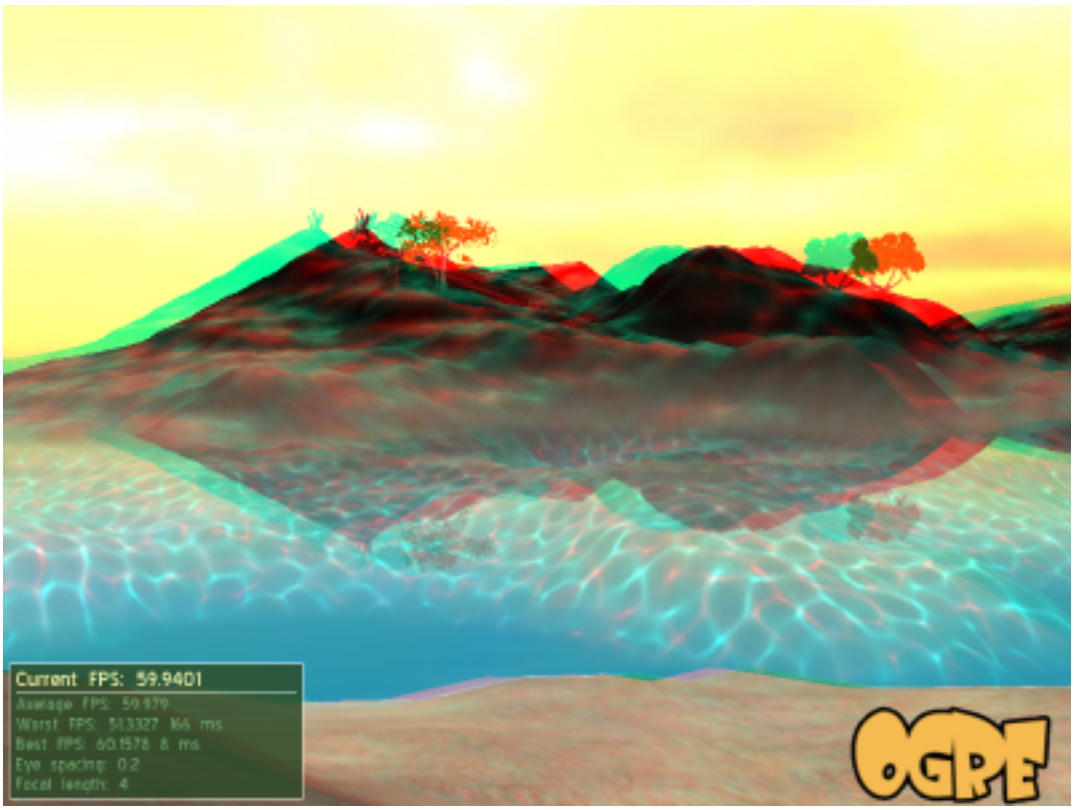
Obrázek A.6: Parallax Mapping na stěnách místnosti



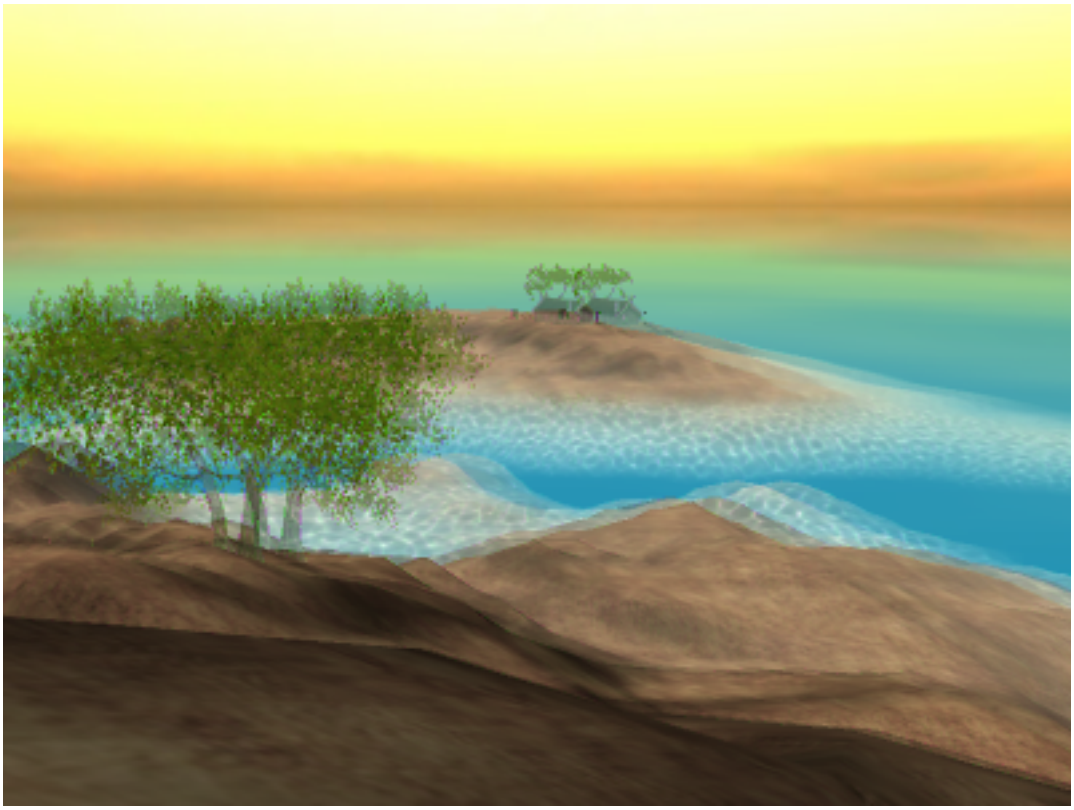
Obrázek A.7: Animovaný robot

Obrázek A.8: Netriviální scéna

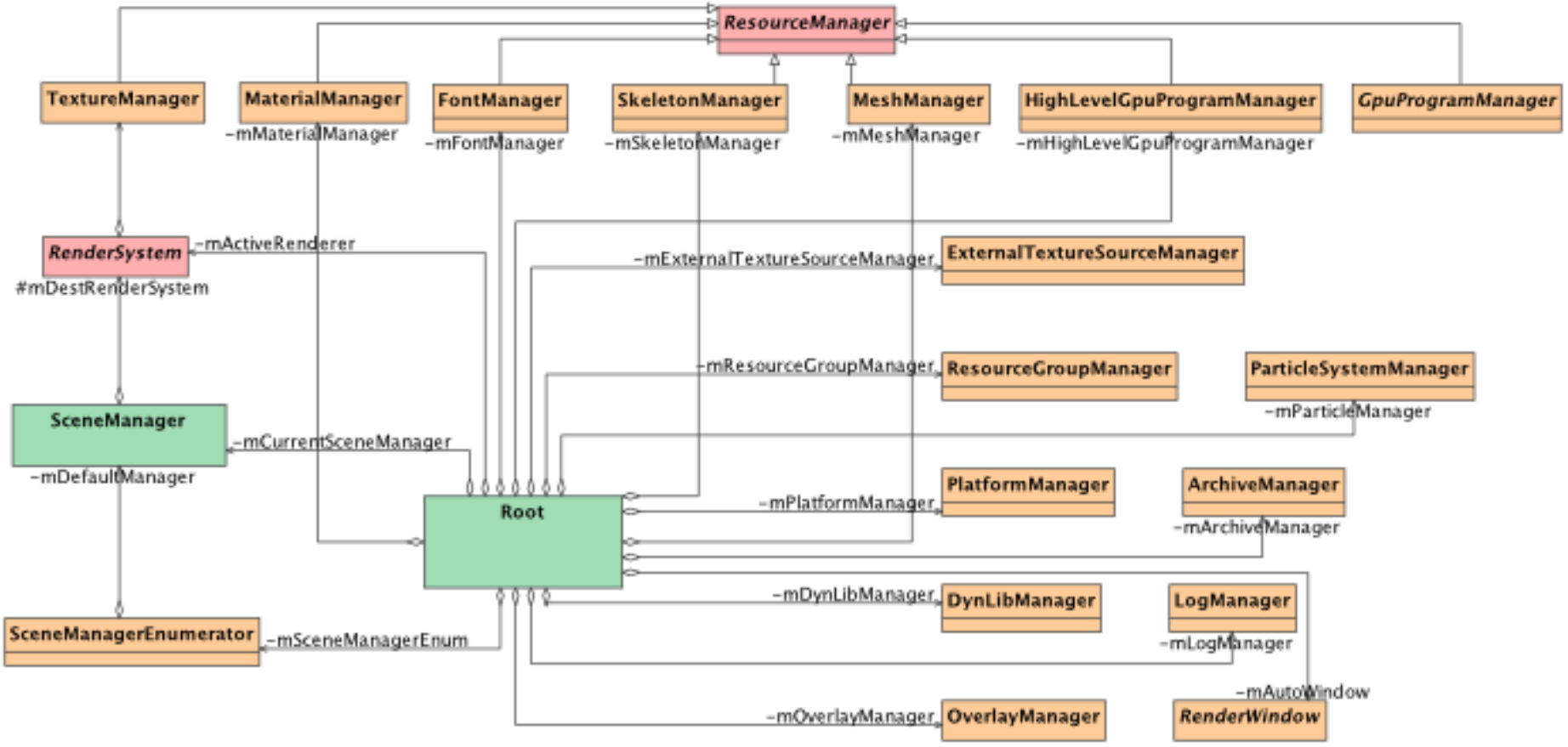
Obrázek A.9: Viditelný odraz na vodní hladině



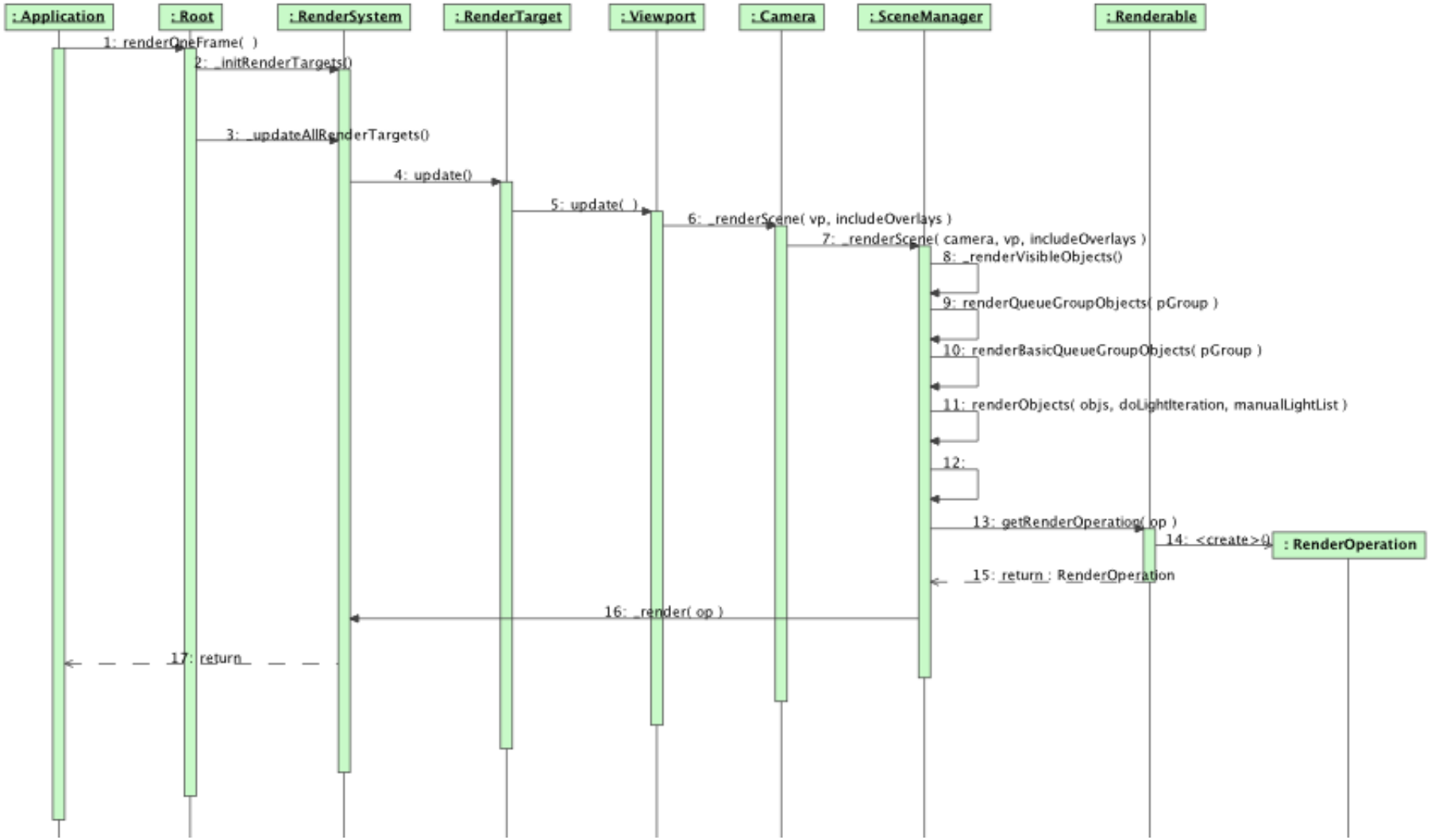
Obrázek A.10: Ostrov za použití anaglyfu



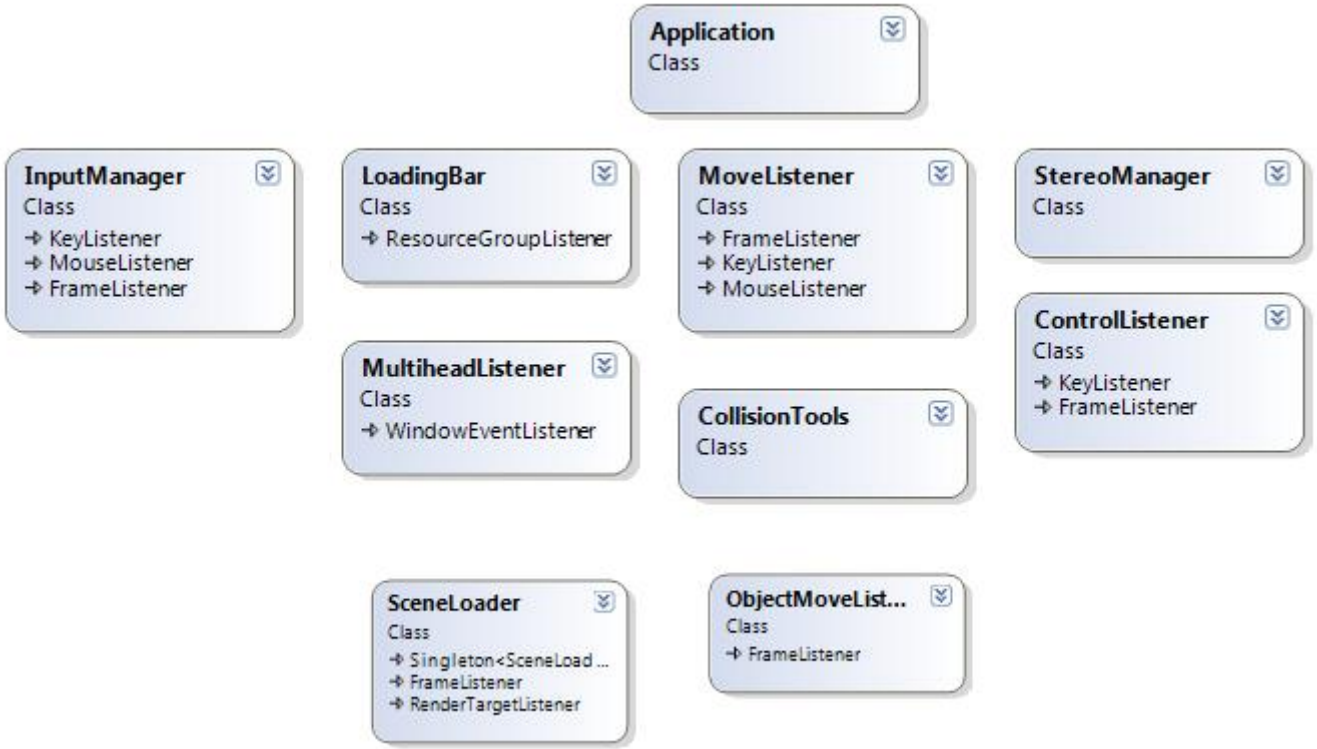
Obrázek A.11: Ostrov za použití Sharp 3D



Obrázek A.12: Diagram návrhu enginu (převzato z [4])



Obrázek A.13: Renderovací cyklus (převzato z [4])



Obrázek A.14: Diagram tříd programu

Dodatek B

Zdrojové kódy

B.1 Materiál pro multitexturing

```
// definice materialu, techniky a průchodu
...
// první texturovací jednotka
texture_unit
{
    // zdrojem je první textura
    texture brick.jpg
    tex_address_mode mirror
    scale 0.1 0.1
}

// druhá texturovací jednotka
texture_unit
{
    // zdrojem je textura s alfa hodnotou
    texture moss_alpha.png
    tex_address_mode mirror
    scale 0.5 0.5
    // jako operace bude použita
    // operace alfa blending
    colour_op alpha_blend
}

// třetí texturovací jednotka
texture_unit
{
    // druhá textura
    texture moss.jpg
    tex_address_mode mirror
    scale 0.1 0.1
    // operace provede změnu pixelu na barvu
    // pixelu této textury v závislosti na alfa hodnotě
    colour_op_ex blend_current_alpha src_texture src_current
}
```

B.2 Materiály a shadery pro stereoskopické techniky

B.2.1 Pixel shader pro anaglyf

```
// Pixel shader (fragment program) pro anaglyf
// je standardní součástí stereo vision manageru
float4 Anaglyph_fp (
    // vstupní texturovací koordináty
    in float2 texCoord: TEXCOORD0,
    // statické (pro všechny pixely konstantní) textury
    uniform sampler2D LeftTex : register(s0),
    uniform sampler2D RightTex : register(s1)
) : COLOR {
    // načtení pixelů z obou textur podle pozice
    float4 left = tex2D(LeftTex, texCoord);
    float4 right = tex2D(RightTex, texCoord);

    // sestavení výsledné barvy pixelu
    // LEFT_MASK a RIGHT_MASK jsou parametry zadané při
    // kompilování shaderu a umožnují použít jeden program
    // pro více typů anaglyfu
    return left * float4(LEFT_MASK) + right * float4(RIGHT_MASK);
}
```

B.2.2 Pixel shader pro Sharp 3D

```
// Pixel shader (fragment program) pro Sharp 3D
// vytvořen speciálně pro tuto práci
// jedná se v podstatě o kombinaci shaderu pro anaglyf (viz. výše)
// a shaderu pro interlaced (prokládaný) mód odkud je převzato
// určování sudosti a lichosti pixelu.
float4 Sharp3D_fp (
    // input
    in float2 texCoord: TEXCOORD0,
    // parameters
    uniform float4 screenSize,
    uniform sampler2D LeftTex : register(s0),
    uniform sampler2D RightTex : register(s1)
) : COLOR {
    // vrátí  pokud je pozice pixelu lichá, jinak vrací 1
    float2 p = step(0.5f, frac(texCoord * screenSize.xy * 0.5f));
    // načtení pixelů z obou textur podle pozice
    float4 left = tex2D(LeftTex, texCoord);
    float4 right = tex2D(RightTex, texCoord);

    // pokud je pixel na sudé pozici
    if (p.x)
    {
        // pravý, levý, pravý obraz
        return float4(right.r, left.g, right.b, 1);
    }
    else
    {
        // levý, pravý, levý obraz
        return float4(left.r, right.g, left.b, 1);
    }
```

```
}
```

B.2.3 Materiál a definice pixel shaderu pro Sharp 3D

```
// Definice fragment programu pro techniku
// Sharp 3D
fragment_program Stereo/Sharp3D_fp cg
{
    // program bude načítat ze souboru
    // Stereoscopy.cg
    source Stereoscopy.cg
    // profily překladu
    profiles ps_2_0 arbfp1
    // název funkce, která bude volána
    // popis funkce výše
    entry_point Sharp3D_fp
    // vstupní parametry fragment programu
    // zde se nastavuje parametr pro velikost
    // viewportu
    default_params
    {
        param_named_auto screenSize viewport_size
    }
}

// Materiál použitý pro Sharp 3D
// vyházi z materiálu kompozitoru
material Stereo/Sharp3D : Stereo/BaseCompositorMaterial
{
    technique
    {
        pass
        {
            // v průchodu bude využit fragment program
            // Stereo/Sharp3D_fp uvedený výše
            fragment_program_ref Stereo/Sharp3D_fp
            {
            }
        }
    }
}
```

Dodatek C

Uživatelská dokumentace

C.1 Spuštění programu

Program lze spustit bez parametrů i s parametry. V případě, že parametry nejsou uvedeny, je zobrazen standardní dialog Ogre pro nastavení parametrů programu (rozlišení obrazovky, zda bude program spuštěn přes celou obrazovku apod.).

Parametry lze zadat např. v následujícím tvaru:

```
program.exe ogre-config="cesta k souboru" scene-name=NazevSceny
```

Jak je vidět z příkladu, tak lze hodnoty parametrů zadávat bez uvozovek nebo v uvozovkách v případě, že se v hodnotě nachází nějaká mezera (parametry jsou rozdělovány podle mezery). Všechny použitelné parametry i s popisy jsou uvedeny v tabulce C.1.

C.1.1 Problémy se spuštěním

Při spuštění programu může dojít k chybě, kdy Windows ohlásí, že mu chybí knihovna „d3dx9_41.dll". To je způsobeno tím, že upravené zdrojové kódy Ogre byly přeloženy pod nejnovější verzí DirectX SDK (březen 2009) a knihovna se nemusí vyskytovat v systému. Pokud tento problém nastane je daná knihovna na datovém médiu přiložena.

C.2 Nastavení

V případě spuštění programu bez parametrů je zobrazen konfigurační dialog (viz obrázek. C.1), kde lze nastavit programu několik parametrů. Hlavním nastavením je

Parametr	Vysvětlivka
ogre-config	Alternativní konfigurační soubor Ogre
scene-name	Načtení scény podle názvu zadaného v XML souboru. Scéna musí být uvedena v souboru scén
scene-file	Přímé zadání souboru se scénou
stereo-config	Konfigurační soubor pro stereo.

Tabulka C.1: Přehled spouštěcích parametrů

volba renderovacího zařízení. V programu jsou přístupné jen DirectX 9 a OpenGL[1]. Podle volby se dále zobrazí seznam parametrů. Většinou se jedná o nastavení vázané ke zvolenému renderovacímu zařízení, ale je zde přidáno i několik dalších přímo pro program. Mezi ně patří následující.

Scene

Název renderované scény získaný ze souboru XML. Seznam scén je získán ze souboru scén, který se nalézá ve stejném adresáři jako spouštěcí soubor pod jménem „scenes.cfg".

Stereo mode

Zvolený typ stereoskopického zobrazování. Seznam je načítán ze správce stereoskopického zobrazování (seznam podporovaných technik).

Rendering Device

Tento parametr v normální verzi Ogre není. Přibyl po aplikaci „Multiple render windows" patche. Dovoluje nastavit primární zobrazovací zařízení (monitor).

C.2.1 Nastavení stereoskopického zobrazování

Většinu parametrů pro stereoskopické zobrazování lze nastavit buď v konfiguračním dialogu nebo přímo v programu. Změnu vzdálenosti očí nebo ohniskové vzdálenosti lze měnit jen podle nějakého kroku, který nemusí poskytovat přesné nastavení požadovaných parametrů. Tyto parametry lze nastavit nezávisle na programu v konfiguračním souboru „stereo.cfg", kde lze zadat přesné hodnoty.

C.3 Ovládání

Ovládání programu je popsáno v tabulce C.2 a pro otáčení kamery slouží myš.

[1]Program byl převážně testován pod DirectX a v případě spuštění pod OpenGL může dojít k nesprávnému zobrazení renderované scény.

Obrázek C.1: Dialog nastavení Ogre

Klávesa	Akce
W, A, S, D	Pohyb po virtuálním světě
O, P	Přepínání typu stínu
F	Přehození pravé/levé oko
I	Zobrazení/skrytí informačního GUI
U	Uložení konfiguračního souboru pro stereo
Num +, Num -	Úprava vzdálenosti mezi očima
Num *, Num /	Úprava ohniskové vzdálenosti
Num ,	Nastavení ohniskové vzdálenosti na nekonečno
Num 0	Nastavení vzdálenosti mezi očima na nulu
M	Vytvoření screenshotu

Tabulka C.2: Ovládání programu

Dodatek D

Vytvoření vlastní scény

Ukázková aplikace umožňuje vytvoření a načtení vlastní scény pomocí XML. Předvytvořené scény lze nalézt v adresáři „media/scenes" a podle nich se inspirovat v návrhu dalších.

Pro snazší vytváření byl vytvořen soubor se schématem, podle kterého lze snadno napsat správně soubor scény. Proto se k psaní velice hodí nějaký XML editor s podporou schémat. Mě se velice osvědčil integrovaný editor ve Visual Studio 2008 (měl by být dostupný i v Express Edition).

Soubor je navržen tak, že elementy uvnitř nějakého jiného elementu budou po načtení do programu taktéž závislé na daném rodičovském elementu. Vytváří se tak vlastně stromová struktura zobrazované scény. Většina atributů elementů je odvozena podle atributů jednotlivých objektů v enginu a jejich nastavení má stejný vliv jako kdyby se scéna psala přímo v programu. Jen některé elementy mají specifické atributy, které dovolují další efekty nad samotný engine. Např. elementu „Light" lze zadat, že se má na pozici světla zobrazovat billboard či zapnout náhodný pohyb po scéně.

Více zde strukturu vysvětlovat nebudu, jelikož díky schématu, názvům atributů elementu a předvytvořeným scénám lze vytvořit další scény bez větších problémů.

Přidání scény do programu

V souboru „scenes.cfg" je uveden seznam scén, které může program načíst. Proto stačí přidat další řádku s adresou na nově vytvořenou scénu. Po spuštění programu se objeví možnost zvolit si scénu podle názvu zadaného v XML souboru.