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

POKROČILÝ RENDERING POMOCÍ KNIHOVNY OPENSCENEGRAPH

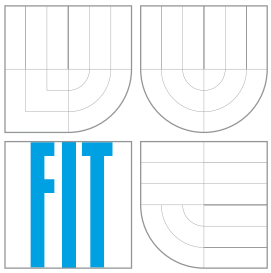
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ ONDRUŠKA

BRNO 2009

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

POKROČILÝ RENDERING POMOCÍ KNIHOVNY OPENSCENEGRAPH

ADVANCED RENDERING IN OPENSCENEGRAPH LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ ONDRUŠKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. MIROSLAV ŠVUB
SUPERVISOR

BRNO 2009

Abstrakt

Tato bakalářská práce má za úkol předvést knihovnu OpenSceneGraph a její použití v kombinaci se shadery psanými v OpenGL Shading Language. Práce se nejdříve věnuje právě základům jazyka GLSL. Následující kapitola představí základy práce s knihovnou OpenSceneGraph. Další kapitoly se pak věnují postupně jednotlivým shaderům. Prvně je to vertex displacement mapping, po něm následuje cartooning a posledním shaderem je vodní hladina.

Abstract

This bachelor's thesis presents OpenSceneGraph library and its use with shaders written in OpenGL Shading Language. First of all it presents the basics of GLSL. Next chapter introduces the basics of OpenSceneGraph library. Following chapters are about different shaders. First is vertex displacement mapping, followed by cartooning and the last one is water surface.

Klíčová slova

OpenGL Shading Language, GLSL, OpenSceneGraph, OSG, graf scény, posun vertexu, nerealistické zobrazení scény, detekce hran, laplaceův operátor, sobelův operátor, vodní hladina, hloubková mapa, normálová mapa

Keywords

OpenGL Shading Language, GLSL, OpenSceneGraph, OSG, shader, displacement, cartooning, cartoon, toon, cel-shading, image processing, edge detection, laplacian, sobel, robert-cross, water surface, fresnel, depth map, normal map

Citace

Jiří Ondruška: Pokročilý rendering pomocí knihovny OpenSceneGraph, bakalářská práce, Brno, FIT VUT v Brně, 2009

Pokročilý rendering pomocí knihovny OpenSceneGraph

Prohlášení

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

.....
Jiří Ondruška
19. května 2009

Poděkování

Rád bych vyjádřil své díky Ing. Miroslavu Švubovi, vedoucímu mé bakalářské práce, za poskytnutou pomoc a veškerý čas strávený konzultacemi.

Obsah

Kapitola 1

Úvod

Počítačová grafika prošla za poslední léta obrovským vývojem a to nejen díky cenově dostupným a přitom výkonným grafickým adaptérům, schopných akcelerovat výpočty realizované pomocí krátkých programů (shaderů).

Předmětem této práce je provést čtenáře základy práce s knihovnou OpenSceneGraph a použitím těchto shader programů v rámci této knihovny k vytvoření pokročilých efektů a sestavení webového návodu (tutoriálu) popisujícího, jak tyto efekty realizovat. Rozčlenění práce a její celkový styl tomu bude též odpovídat.

Nejdříve se budu zabývat základy OpenGL Shading Language, jazyka sloužícího pro psaní shader progamů. Dále uvedu pár základních informací o knihovně OpenSceneGraph, jak vypadá jednoduchý program vytvořený s její pomocí. Také popíši, jak vypadá práce se shadery v prostředí OpenSceneGraphu.

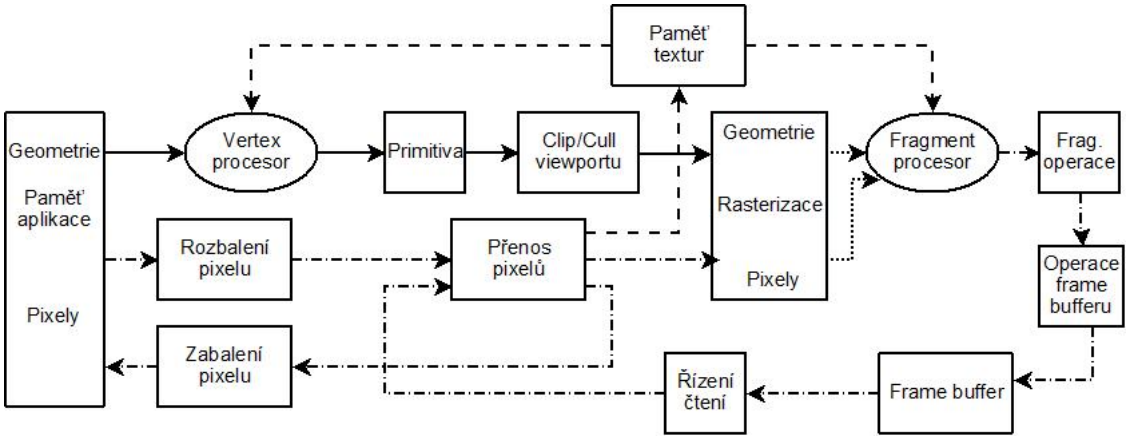
Po úvodních kapitolkách budou následovat jednotlivé efekty. Nejsnažší z nich displacement, poté cartooning s detekcí hran a na závěr vodní hladina. U každého efektu uvedu jeho principy, jak jej naimplementovat pomocí OpenGL Shading Language a jak připravit jednoduchou scénu v OpenSceneGraphu pro předvedení těchto shaderu.

Kapitola 2

GLSL - OpenGL Shading Language

2.1 Úvod

V dnešní době grafické karty čím dál tím víc nahrazují statické postupy zpracování programovatelností. Nejvíce je to znát při zpracování vertexů a fragmentů. Zpracováním vertexu rozumíme operace, které se provádí pro každý vertex, např. transformace či osvětlení. Fragmenty jsou pak datové struktury pro jednotlivé pixely, získané rasterizací grafických primitiv. Zpracování fragmentů probíhá opět pro každý fragment, jedná se o operace jako např. čtení z textur. Zavedením OpenGL Shading Language se tedy ze statické funkcionality stává programovatelné prostředí, které zvládne to samé jako původní statická funkcionalita, ale i mnohem víc. Celý průběh zpracování dat v OpenGL lze vidět na obrázku 2.1.



Obrázek 2.1: Průběh zpracování v OpenGL. (OpenGL pipeline)

GLSL jsou vlastně dva velice si blízké jazyky, které slouží k vytváření shaderů pro programovatelné procesory obsažené v OpenGL, s drobnými rozdíly specifickými právě pro vertex a fragment shader. Součástí standardu OpenGL je již od verze 2.0.

2.2 Základy jazyka GLSL

GLSL je vysoko-úrovňový procedurální jazyk, založený na programovacích jazycích z rodiny C/C++, ovšem je typově přísnější než tyto jazyky. Obsahuje ovšem i dost prvků z různých konkurenčních jazyků určených pro psaní shaderů.

Všechny proměnné a funkce musí být před použitím řádně deklarovány. V GLSL není žádný implicitní datový typ. Všechny proměnné a funkce musí mít určený datový typ a volitelně i kvalifikátor. Funkce jsou volány podle návratových hodnot.

V GLSL neexistují žádné implicitní konverze typů, s jedinou vyjímkou. Integer se v určitých případech může vyskytnou v místech, kde se očekává float, příčemž dojde ke konverzi.

Popsání všech vlastností není předmětem této práce, proto zde uvedu jen ty nejhlavější aspekty. Zbylé informace jsou dostupné ve specifikaci jazyka.

2.2.1 Datové typy

Klasické typy Mezi tyto typy patří `void` pro beznávratové funkce, `int` (celá čísla se znaménkem), `bool` (boolean) a `float` (čísla v plovoucí řádové čárce), známé právě z jazyka C.
Vektorové typy Jedná se o skupinu datových typů sloužících pro práci s vektory. Vektory nesou buď dvě (`vec2`), tří (`vec3`) nebo čtyři (`vec4`) prvky typu `float`. Dostupné jsou i ve variantách specifických pro zbylé základní typy, například `bvec2` je vektor nesoucí dva prvky typu `bool` či `ivec4` nese čtyři prvky typu `int`.
Maticové typy Jsou to nositeli prvků typu `float`, jde o tři základní typy matic, přesněji `mat2`(matice 2x2), `mat3`(matice 3x3) a `mat4` (matice 4x4). Dále jsou definovány typy pro další rozměry matic, jako například `mat2x3` či `mat4x2`. Z názvu těchto typů jasně vyplívá velikost odpovídající matice. Takto jsou definovány všechny rozměry matic od $2x2$ pro $4x4$.
Texturové typy - samplery Tyto typy slouží k zpřístupnění textur. Obdobně jako u vektorů a matic exitují pro jednotlivé typy textur odpovídající typy samplerů. Tedy `sampler1D`, `sampler2D`, `sampler3D` či `samplerCube` a trochu specifičtější `sampler1DShadow` a `sampler2DShadow`.

2.2.2 Paměťové kvalifikátory

const Konstanta. Kvalifikátor lze použít pro globální proměnné stejně jako pro lokální, popřípadě i v parametru funkce, říkajíc tak, že je parametr jen pro čtení.
attribute Určuje, že se jedná o spojení mezi vertex shaderem a OpenGL daty. Lze užít jen na globální úrovni.
uniform Hodnota s tímto kvalifikátorem zůstává stejná po celou dobu zpracování primitiva. Uniform vytváří spojení mezi shaderem, OpenGL a aplikací. Lze použít jen pro globalní proměnné.
varying Vytváří spojení mezi vertex a fragment shaderem, umožňujíc tak předávat interpolované hodnoty. Opět lze použít jen pro globální proměnné.

2.2.3 Vestavěné proměnné

GLSL obsahuje řadu vestavěných proměnných a uniformů. Popsat všechny by bylo nad rámec této práce, proto zde uvedu jen ty nejhlavnější. Některé z těchto proměnných jsou dostupné jen z určitého shaderu.

Vertex shader

vec4 gl_Position Jedná se o výstup z vertex shaderu určující pozici daného vertexu. Zapsání této hodnoty je povinné.
vec3 gl_Normal Tento atribut nese informaci o normále daného vertexu. Vstupní hodnota, jen pro čtení.

vec4 gl_Color Atribut barvy daného vetexu. Jedná se o RGBA barvu. Jen pro čtení.
vec4 gl_MultiTexCoord0 Tento atribut je nositelem texturových koordinátů. Číslo(0-7) na konci určuje texturu, ke které dané koordináty patří. Opět jen pro čtení.
vec4 gl_TexCoord[] Proměnná s kvalifikátorem `varying` sloužící k předání souřadnic textury do fragment shaderu. Možnost zápisu i čtení. Velikost toho pole je dána konstantou `gl_MaxTextureCoords`.

Fragment shader

vec4 gl_FragColor Výstupní hodnota z fragment shaderu. Určuje barvu fragmentu.
vec4 gl_FragCoord Souřadnice daného fragmentu.

Vestavěné uniformy

mat4 gl_ModelViewProjectionMatrix Matice nesou transformaci pro pohled a projekci. Jde o kombinaci matic `gl_ModelViewMatrix` a `gl_ProjectionMatrix`. Dostupné jsou i inverzní matice a další.
gl_LightSource[] Jedná se o pole struktur typu `gl_LightSourceParameters`. Obsahuje vlastnosti zdroje světla, jako například pozici.
gl_FrontMaterial Zpřístupňuje vlastnosti materiálu. Stejně tak existuje i `gl_BackMaterial`.

2.2.4 Vestavěné funkce

Opět si uvedeme jen pár hlavních funkcí, které budeme nezbytně potřebovat v dalších kapitáloch.

Trigonometrické funkce

- genType sin(genType)
- genType cos(genType)
- genType tan(genType)

Parametry těchto funkcí a jejich návratové hodnoty jsou typu `genType`, jedná se o generický datový typ, lze jej nahradit jedním z následujích typů: `float`, `vec2`, `vec3`, `vec4`.

Exponenciální funkce

- genType pow(genType, genType)
- genType sqrt(genType)

Náhledy textur

Tyto funkce vrací hodnotu hledaného pixelu textury.

- vec4 texture2D(sampler2D, vec2 [,float bias])
- vec4 texture3D(sampler3D, vec3 [,float bias])
- vec4 textureCube(samplerCube, vec3 [,float bias])

- vec4 shadow2D(sampler2DShadow, vec3 [,float bias])

Tyto funkce existuji i ve verzích pro projekci, ke jménu funkce se dodá `Proj` i ve verzích pro 1D textury.

Ostatní funkce

- genType floor(genType)Vrací nejbližší menší celé číslo.
- genType ceil(genType)Vrací nejbližší větší celé číslo.
- genType clamp(genType x, genType minVal, genType maxVal)Upraví hodnotu x na interval $<minVal, maxVal>$.
- genType normalize(genType)Upraví vektor na délku 1, při zachování jeho směru.

2.3 Základní program v GLSL

2.3.1 Vertex shader

Základní náplní každého vertex shaderu je umístění vertexu z pohledu kamery. K tomu je zapotřebí znát tři věci. První je pozice vertexu ve světě (scéně), druhou je projekční (projection) matice a poslední je pohledová (view) matice. Všechny tři hodnoty jsou ve vertex shaderu přímo dostupné, dokonce je dostupná i kombinace matic, takže je již nemusíme násobit.

Základní vertex shader by tedy vypadal následovně.

```
void main(){
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Místo těchto výpočtů lze použít funkci `ftransform()`, avšak v poslední verzi jazyka (1.40) byla tato funkce zrušena.

Vhodné je též předat barvu vertexu do fragment shaderu, pokud tedy neplánujeme použít texturu objektu. Upravený program by mohl vypadat například takto.

```
varying vec4 color;
void main(){
  color = gl_Color;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

2.3.2 Fragment shader

Obdobně jako základní činností vertex shaderu bylo nastavení pozice vertexu, fragment shader má za povinnost nastavit barvu daného fragmentu. Za předpokladu, že jsme si předali z vertex shaderu barvu, by základní program fragment shaderu vypadal takto.

```
varying vec4 color;
void main(){
  gl_FragColor = color;
}
```

Kapitola 3

OpenSceneGraph

3.1 Úvod

OpenSceneGraph (dále jen OSG) je vice platformní OpenSource sada nástrojů pro vývoj grafických aplikací jako jsou hry, simulátory či vizualizace. OSG je založeno na konceptu SceneGraphu, poskytujíc objektově zaměřené prostředí vystavěné nad OpenGL. Vývoj aplikace lze tedy urychlit díky absenci práce s nízkoúrovňovými voláními a jejich optimalizací.

3.2 Historie

Počátek OSG se datuje do roku 1998, kdy Don Burns započal práci na svém simulátoru rogala za pomoci Performer scene graphu pro Linux.

Následující rok se k projektu připojil Robert Osfield, který část kódu týkající se SceneGraphu přepracoval pro platformu Windows. Poté byly zdrojové texty zpřistupněny jako open source a vznikl web openscenegraph.org[9]. Ještě tentýž rok R.Osfield přepracoval scene graph do standardu C++ s využitím návrhových vzorů.

Nakonec pak v roce 2001 projekt přešel na plně profesionální úroveň, díky rostoucímu zájmu o něj.

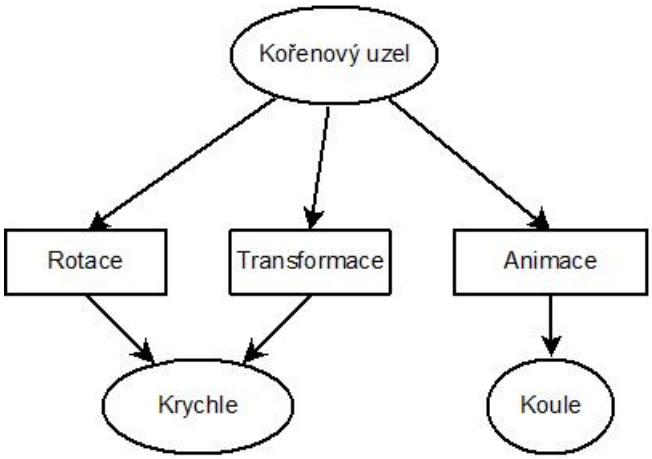
3.3 Základ OpenSceneGraphu

3.3.1 Scene Graph

Grafem rozumíme datový typ složený z uzlů (angl. nodes) a hran, popisujících vazbu jednotlivých uzlů. Základem mnoha grafických aplikací je právě graf scény (Scene Graph), který ulehčuje správu objektů a jejich transformací a tím i správu celého programu.

Formálně lze graf scény definovat jako strom nebo orientovaný neperiodický graf s libovolným počtem potomků. Jednoduchý graf scény lze vidět na obr. 3.1.

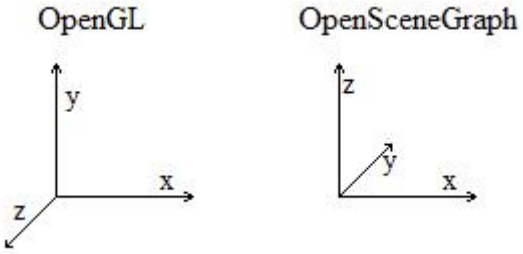
Grafem scény pak rozumíme datovou strukturu zaměřenou na logické (např. prostorové) rozčlenění grafické scény. Scéna je vykreslována od kořenového uzlu dolů až po listy. Takto lze reprezentovat celý „svět“ ve scéně. Jednotlivé uzly mohou být skupiny objektů, transformace nebo i animace objektů. Listy jsou pak samotné objekty, jejich geometrie či materiály.



Obrázek 3.1: Graf scény

3.3.2 Souřadný systém OSG

Přestože je OSG vystavěn nad OpenGL v některých ohledech jsou naprosto odlišné. Hlavním rozdílem oproti OpenGL, na který je nutné si dát pozor, je souřadný systém. Díky svému využití v simulacích využívá OSG stejný souřadný systém jako ve fyzice. Nejlépe lze rozdíl pochopit z obrázku 3.2.



Obrázek 3.2: Souřadný systém

3.3.3 Přehled hlavních knihoven OSG

Knihovna osg obsahuje všechny třídy vztahující se k uzlům grafu, ale také třídy pro matematické operace.
Knihovna osgDB obsahuje třídy a metody pro vytvoření a renderování 3D databází a zároveň poskytuje plug-iny pro čtení a zápis podporovaných souborů.
Knihovna osgUtil obsahuje třídy a metody pro manipulaci s grafem a jeho obsahem. Také poskytuje metody pro sběr statistik a optimalizaci grafu.
Knihovna osgViewer poskytuje flexibilní a vysoce upravitelnou třídu Viewer pro zobrazení scény.

3.3.4 Seznam užitečných tříd

osg::ref_ptr<T> Jedná se o univerzální „chytrý“ ukazatel na jakýkoli objekt OSG. „Chytrý“ ukazatel proto, že udržuje počet referencí na objekt, při nulovém počtu automaticky ob-

jekt dealokuje. Lze odkazovat i na objekty vlastní třídy, pak je ale nutné danou třídu odvodit ze třídy `osg::Referenced`. Pro použití je nutné připojit dva hlavičkové soubory: `<osg/ref_ptr>` a `<osg/Referenced>`.

osgViewer::Viewer Jak jsem již zmínil, tato třída nám poslouží pro průchod grafem, tedy pro zobrazení scény. Pro zobrazení scény musí viewer vědet, co má zobrazit. Toho docílíme voláním jeho metody `setSceneData(osg::Node *node)`, které parametrem předáme zobrazovanou scénu. Další užitečná metoda je `addEventHandler()`, kterou jsme schopni přidat např. zpracování událostí klávesnice. Nejdůležitější metodou je ovšem `run()`, která spouští renderovací smyčku. Tuto třídu lze použít po připojení `<osgViewer/Viewer>`.

osg::Geometry Tato třída reprezentuje samotnou geometrii objektu (pole vertexů). Zároveň nese informace o barvě, koordinátech textury či normále daných vertexů. Lze ji najít v hlavičkovém souboru `<osg/Geometry>`.

osg::Geode Tato třída slouží jako uzel nesoucí geometrii nebo jakýkoli objekt odvozený ze třídy `osg::Drawable`. Třída se nachází v `<osg/Geode>`.

osg::Node Základní třída pro všechny uzly grafu. Poskytuje rozhraní pro nejběžnější operace s uzly. Lze najít v `<osg/Node>`.

osg::Group Obecná třída spravující seznam uzlů (potomků). K nalezení v `<osg/Group>`.

osg::StateSet Třída sloužící k uchování sady atributů reprezentujících stavy OpenGL. Každý uzel v grafu obsahuje referenci na tuto třídu. Obsahuje poměrně velké množství různých metod, mezi jinými např. `addUniform(osg::Uniform *uniform)`. Třída je součástí jádra osg(`<osg>`).

osg::Texture Jedná se o třídu, ze které jsou odvozeny další třídy pro práci s texturami jako např. `osg::Texture1D`, `osg::Texture2D` nebo `osg::TextureCubeMap`. Jednotlivé třídy lze najít v odpovídajících hlavičkových souborech, např. `<osg/Texture2D>`.

osg::Light Třída zapouzdřuje funkcionalitu OpenGL `glLight()`. Slouží k nastavení vlastností světla. Objekt se předá třídě `osg::LightSource`, která pak tvoří uzel pro světlo scény. Třídy lze najít v hl. souborech `<osg/Light>` a `<osg/LightSource>`.

osg::Uniform Slouží k předání dat do GLSL programů, vytvářejíc spojení mezi aplikací a shaderem, jak bylo zmíněno dříve (2.2.2).

Třídy vektorů OSG poskytuje sadu tříd odpovídajících datovým typům známých z GLSL, `osg::Vec2`, `osg::Vec3` a `osg::Vec4` v různých variacích, např. `osg::Vec2f`. Třídy se nachází přímo v `<osg>`.

Třídy matic Stejně jako třídy vektorů, OpenSceneGraph poskytuje třídy matic. Například `osg::Matrixf`. Tyto třídy lze opět najít přímo v `<osg>`.

Třídy callbacků Tyto třídy nám slouží k pravidelné aktualizaci či modifikaci uzlů. Fungují na principu zpětného volání, kdy se při průchodu grafem kontroluje, zda daný uzel nemá nastavený nějaký callback, pokud ano, tak je daný callback vyvolán. V OSG existuje velké množství callbacků, některé jsou univerzálnější jako například `osg::NodeCallback`, některé slouží ke specifickým účelům, za všechny například `osg::Uniform::Callback`.

3.4 Základní program

Zde bych rád popsal základní strukturu aplikace, kterou jsem využil ve všech aplikacích uvedených dále.

První věc, kterou je třeba v programu vytvořit je kořenový uzel grafu. Nejlépe tak lze učinit vytvořením instance třídy `osg::Group`.

```
osg::ref_ptr<osg::Group> rootNode = new osg::Group;
```

Následně budeme potřebovat nějaký model, který budeme chtít zobrazit. Máme několik možností jak vytvořit geometrii, buď můžeme nahrát objekt ze souboru nebo vytvořit geometrii sami pomocí `osg::Geometry` a `osg::Geode`. Výsledek bude stejný. Získáme uzel grafu, který připojím ke kořenu. Jako ukázka poslouží kód pro načtení modelu ze souboru.

```
osg::Node* loadedModel = osgDB::readNodeFile("cow.osg");
```

Funkce `osg::readNodeFile("cow.osg")` se skrývá v hl. souboru `<osgDB/ReadFile>`. Takto načtený uzel nesoucí model, pak stačí připojit ke kořenu voláním jeho metody.

```
rootNode->addChild(loadedModel)
```

Následně je potřeba vytvořit instanci `osgViewer::Viewer`. Před voláním jeho metody `run()`, je potřeba nastavit pár věcí. Voláním metody `setCameraManimulator()`, které předáme jako parametr objekt typu `osgGA::TrackballManipulator`, lze velice snadno nastavit základní ovládání kamery pomocí myši. Zdrojový kód by vypadal následovně.

```
osgViewer::Viewer viewer;
viewer.setCameraManipulator( new osgGA::TrackballManipulator() );
```

Volitelně lze změnit barvu „pozadí“ kamery voláním metody `setClearColor()`. Podstatné je však předat kořenový uzel vieweru. Provedeme opět voláním jedné z jeho metod.

```
viewer.setSceneData(rootNode.get());
```

Zde je zajímavé pozastavit se nad rozdílným způsobem volání metod kořenového uzlu. Volání metody přes `->` způsobí, že se volá metoda třídy `osg::Group`, kdežto při volání přes tečkovou notaci, voláme metodu třídy `osg::ref_ptr<T>`, ta vrací ukazatel, v našem případě ukazatel na `osg::Group`.

Poté co jsme vše nastavily, lze již zavolat metodu `run()` a ukončit naši aplikaci, oboje zaráz.

```
return viewer.run();
```

Tímto bychom měli opravdu velice základní aplikaci. Pokud bychom např. chtěli přidat do scény osvětlení lze to udělat několika řádky kódu.

3.5 Práce se shadery v OpenSceneGraph

Před tím než se dostaneme k samotným efektům v GLSL, by bylo vhodné zmínit jak v OpenSceneGraphu aplikovat shader na objekt či skupinu objektů.

Pro práci se shadery budeme v OSG potřebovat dvě třídy. První z těchto tříd bude `osg::Shader`, která bude sloužit jako nositel vertex a fragment programu. Tato třída umožnuje pomocí své metody přímo nahrát zdrojový kód shaderu ze souboru. V ukázkových aplikacích dostupných v OSG lze najít následující funkci pro načtení shaderu, kterou jsem hojně využil při své práci se shadery.

```
void LoadShaderSource(osg::Shader* shader, const std::string& fileName){
  std::string fqFileName = osgDB::findDataFile(fileName);
  if( fqFileName.length() != 0 )
    shader->loadShaderSourceFromFile( fqFileName.c_str() );
  else
    osg::notify(osg::WARN) << "File \"" << fileName
                           << "\" not found." << std::endl;
}
```

Druhou třídou, kterou budeme potřebovat, je `osg::Program`. Tato třída je vlastně zapouzdřením funkcionality OpenGL, přesněji `glProgram`. Třídě programu lze přiřadit jméno programu a shadery, jak si postupně ukážeme.

Pro použití shader programu si nejdříve definujeme proměnné, přesněji statické ukazatele. Lze tak učinit, pro usnadnění práce, na globální úrovní.

```
static osg::Program* ShaderProgram;
static osg::Shader*  VertexShaderObj;
static osg::Shader*  FragmentShaderObj;
```

Před použitím těchto ukazatelů je pak nutné je správně inicializovat. Po inicializaci lze volat již zmíněnou funkci `LoadShaderSource()`.

```
ShaderProgram = new osg::Program;
VertexShaderObj = new osg::Shader(osg::Shader::VERTEX);
FragmentShaderObj = new osg::Shader(osg::Shader::FRAGMENT);
LoadShaderSource(VertexShaderObj, "shader.vert");
LoadShaderSource(FragmentShaderObj, "shader.frag")
```

Následně je vhodné nastavit programu nějaký název, který by nám posloužil pro jeho identifikaci v případě, kdy bychom program umísťovali například do seznamu či obdobného datového typu. Pak lze již přidat samotné shadery voláním metody programu.

```
ShaderProgram->setName( "shaderUkazka" );
ShaderProgram->addShader( VertexShaderObj );
ShaderProgram->addShader( FragmentShaderObj );
```

Nyní máme připravený program shaderu pro použití. Abychom mohli aplikovat shader na některý objekt či skupinu objektů je nutné přiřadit jej danému objektu. Toho lze docílit získáním a úpravou stavu objektu. Přesněji využitím třídy `osg::StateSet`. Objekt této třídy lze z objektu (uzlu) získat voláním jeho metody.

```
osg::StateSet *stateset = loadedModel->getOrCreateStateSet();
```

Proměnná `loadedModel` je typu `osg::Node*` z předchozí části. Teď lze již snadno přiřadit shader program danému objektu.

```
stateset->setAttributeAndModes(ShaderProgram, osg::StateAttribute::ON)
```

Nyní jsme aplikovali úspěšně shader na objekt eventuálně více objektů, pokud tyto objekty sdílí daný `stateset`.

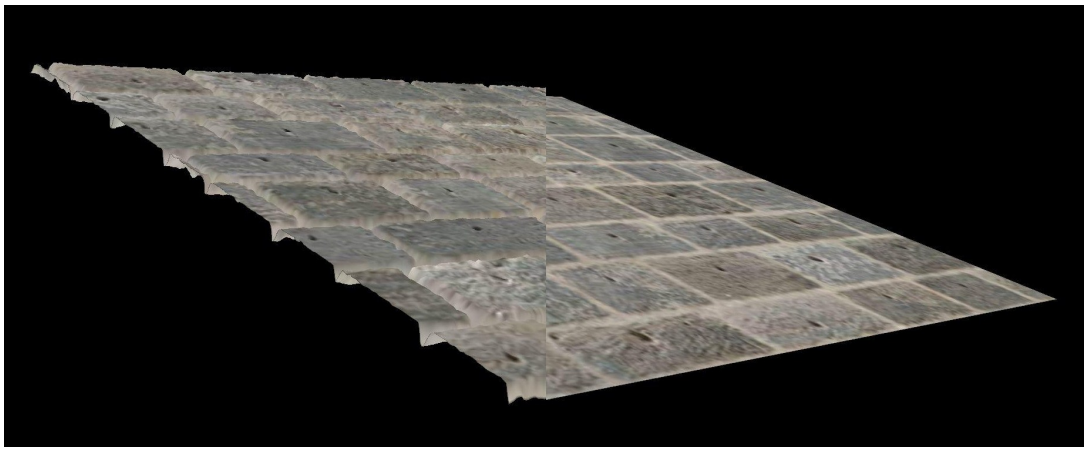
Kapitola 4

Displacement mapping

V předchozích kapitolách jsme se zabývali jazykem GLSL pro psaní shaderů a knihovnou OpenSceneGraph. Nyní se již můžeme dostat k samotným shader programům.

Displacement mapping (displacement = náhrada, přemístění, posuv) nebo též vertex displacement mapping je technika umožňující deformaci polygonové mřížky. Jedná se o posuv pozice vertexu podle dané displacement mapy pro dosažení lepšího povrchového detailu a hloubky, dovolujíc použití efektů jako self-occlusion či self-shadowing.

Dnešní grafické procesory umožňují snadné použití této techniky v reálném čase. Využití principů displacementu lze tedy najít např. u algoritmů pro generování terénu nebo i u aplikací zobrazujících mapy ve 3D.

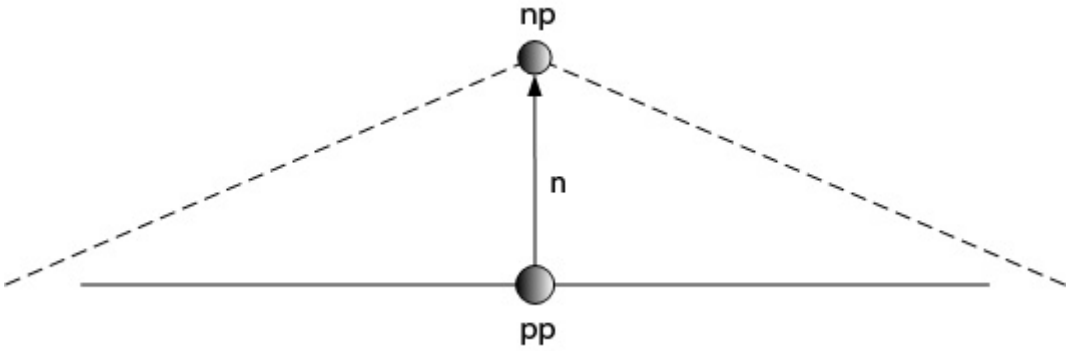


Obrázek 4.1: Ukázka vertex displacement mappingu

4.1 Princip displacementu

Jak jsem již zmínil, základem displacement mappingu je vektor $\overrightarrow{dv}$, získaný z textury, nesoucí informaci o posunu. Tento vektor může nést RGBA barvu (normálová mapa viz obr. 4.3) nebo skalární hodnotu (výšková mapa v odstínech šedi viz obr. 4.3).

Daný vektor, přesněji suma jeho komponent, se pak použije pro změnu pozice vertexu. Nejběžnější metodou je posuv po normále vertexu. Tato metoda má však jednu nevýhodu. Posunuje jen pozici vertexu a zachovává původní normály. O řešení tohoto problému se zmíním později. Z obrázku 4.2 lze odvodit následující rovnici.



Obrázek 4.2: Posun vertexu po normále

$$\overrightarrow{np} = \overrightarrow{pp} + (\overrightarrow{n} * df)$$

kde $\overrightarrow{pp}$ je původní pozice vertexu, $\overrightarrow{np}$ je nová pozice po posuvu, $\overrightarrow{n}$ je normála vertexu a df je normalizovaná hodnota posuvu.

Rovnici pro výpočet nové pozice lze rozšířit o takzvaný váhový faktor sf, kterým je možné ovlinit výslednou „sílu“ displacementu.

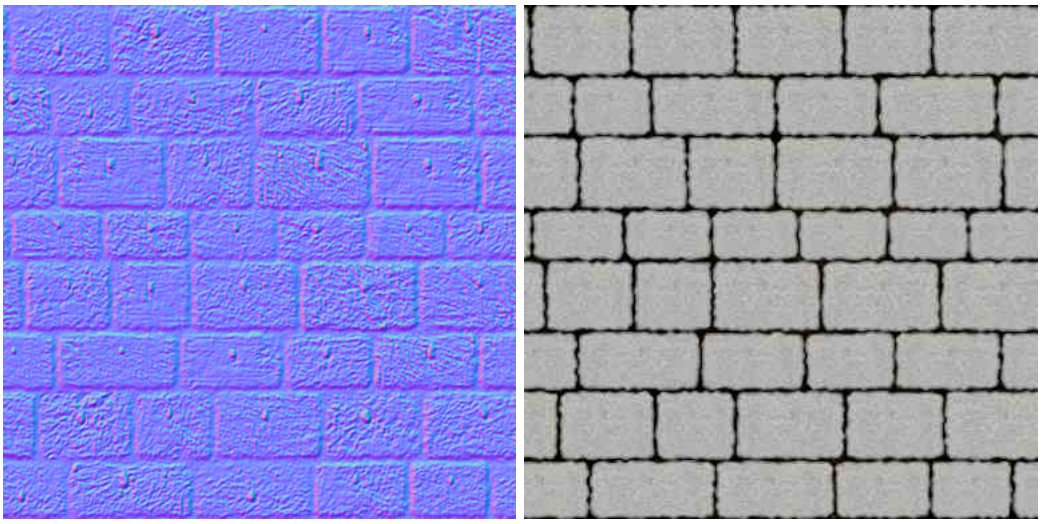
$$\overrightarrow{np} = \overrightarrow{pp} + (\overrightarrow{n} * df * sf)$$

V případě použití vektoru nesoucího RGBA hodnotu lze df spočítat takto:

$$df = 0.3 * dv.r + 0.59 * dv.g + 0.11 * dv.b$$

Při použití skalárního vektoru $\overrightarrow{dv}$ lze jako df použít kteroukoli ze složek $\overrightarrow{dv}$.

$$df = dv.r$$



Obrázek 4.3: Normálová (vlevo) a výšková mapa (vpravo)

BumpDisplacement Mapping Tato technika kombinuje vertex displacement s bump mappingem. Vertex shader zůstává až na drobné změny stejný jako u vertex displacementu, avšak ve fragment shaderu se využije normálové mapy místo normál pro výpočet světla. Tato technika dosahuje relativně dobrých výsledku s drobnými odchylkami, kdy jsou některé oblasti osvětleny, i když by neměli být.

4.2 Implementace

Displacement mapping je, co se shaderu týče, převážně naimplementován ve vertex shaderu. Implementace víceméně přesně odpovídá popsané teorii. V oddíle věnovanému OSG aplikaci pro displacement, bude popsáno načtení textur, jejich aplikace na objekt a předání do shaderů, dále se zmíním o předání hodnot do shaderů pomocí uniformů.

4.2.1 Vertex shader

Jak již název Vertex Displacement Mapping napovídá, hlavní část implementace tohoto efektu bude provedena právě zde, ve vertex shaderu.

První věc kterou budeme ve vertex shaderu potřebovat je samotná mapa pro posuv. Nastavíme si uniform, který nám umožní předat si tuto texturu (mapu) z OSG.

```
uniform sampler2D displacementMap;
```

V těle programu, tedy ve funkci `void main(void)`, pak jako první nastavíme souřadnice pro texturu objektu. Tyto souřadnice se nám budou hodit ve fragment shaderu.

```
gl_TexCoord[0].xy = gl_MultiTexCoord0.xy;
```

Nyní k samotnému displacementu. Načteme z mapy vektor $\overrightarrow{dv}$ posunu.

```
vec4 dv = texture2D( displacementMap, gl_MultiTexCoords0.xy );
```

Pokud hodláme použít pro posuv hodnotu z normálové mapy, pak provedeme přepočet na odstín šedi.

```
float df = 0.30*dv.r + 0.59*dv.g + 0.11*dv.b;
```

V případě, že použijeme výškovou mapu, lze tento řádek kódu vynechat a použít namísto `df` přímo `dv.r`. Použijemeli však výpočet pro `df` nad výškovou mapou, nic se nezmění, jen budeme zbytečně provádět matematické operace. Výšková mapa již je v odstínech šedi.

Než přejdeme k výpočtu nové pozici vertexu je třeba nastavit váhový faktor. Lze tak učinit buď přimo v shaderu nebo si jej předat z programu pomocí uniformu, stačí jen na začátek shaderu doplnit řádek kódu.

```
uniform float scaleFactor;
```

Nyní lze spočítat pozici vektoru přesně podle zmíněné rovnice. Jen je nutno si dát pozor na `gl_Normal`, jde o `vec3`, ale pozice vertexu (`gl_Vertex`) je `vec4`.

```
vec4 newVertexPos = vec4(gl_Normal * df * scaleFactor, 0.0) + gl_Vertex;
```

Takto získanou novou pozici vertexu stačí již jen vynásobit maticí, abychom získali pozici z pohledu kamery a tím jsme hotovi s vertex shaderem.

```
gl_Position = gl_ModelViewProjecionMatrix * newVertexPos;
```

Vyhlazení Pokud bychom chtěli displacement mapu trochu zjemnit (vyhladit), lze využít například funkci, kterou zveřejnila společnost nVidia na svých webových stránkách pro vývojáře[8]. Její GLSL přepis pak vypadá následovně.

```
vec4 texture2D_bilinear( sampler2D texture, vec2 uv)
{
  vec2 frac = fract(uv.xy * textureSize);
  vec4 tex00 = texture2D(texture, uv);
  vec4 tex10 = texture2D(texture, uv + vec2(texelSize, 0.0));
  vec4 texA = mix(tex00, tex10, frac.x);
  vec4 tex01 = texture2D(texture, uv + vec2(0.0, texelSize));
  vec4 tex11 = texture2D(texture, uv + vec2(texelSize, texelSize));
  vec4 texB = mix(tex01, tex11, frac.x);
  return mix(texA, texB, frac.y);
}
```

Pro její použití budeme potřebovat znát dvě hodnoty, které lze získat pomocí uniformů z OSG.

První je `textureSize`, která nám říká rozměr textury. Za zmínku stojí, že jen jeden rozměr textury, předpokládá se tedy čtvercová textura.

Druhá hodnota `texelSize` pak říká, jak velký je jeden pixel textury. Lze ji buď spočítat v OSG a předat nebo ji spočítat přimo v shaderu. Předpokládáme-li, že má textura rozměr 1.0, pak velikost texelu je $\frac{1.0}{textureSize}$.

Získání těchto hodnot pomocí uniformů by vypadalo následovně.

```
uniform float textureSize;
uniform float texelSize;
```

Pokud tuto funkci chceme využít, stačí nahradit řádek

```
vec4 dv = texture2D( displacementMap, gl_MultiTexCoords0.xy );
```

za řádek s voláním funkce.

```
vec4 dv = texture2D_bilinear( displacementMap, gl_MultiTexCoords0.xy );
```

Tuto funkci lze použit pro vyhlazení i ve fragment shaderu. Osobně jsem však nezaznamenal žádný znatelnější dopad použití této funkce, jak ve vertex tak i ve fragment shaderu.

4.2.2 Fragment shader

Fragment shader je pak velice jednoduchý. Stačí jen namapovat na objekt texturu, kterou jsme si předali z OSG.

```
uniform sampler2D colorMap;

void main(void){
   gl_FragColor = texture2D(colorMap, gl_TexCoord[0].xy);
}
```

4.2.3 OpenSceneGraph

U implementace scény v OSG budeme vycházet ze základního programu, který jsem zmínil v kapitole o OSG.

Jako první věc, o kterou je nutné základní aplikaci rozšířit je načtení potřebných textur. Textury načteme ze souboru pomocí funkce `osgDB::readImageFile()`

```
osg::Image *displMapImg = osgDB::readImageFile("displacementMap.jpg");
if(!displMapImg)
  return false;
osg::Image *colorMapImg = osgDB::readImageFile("colorMap.jpg");
if(!colorMapImg)
  return false;
```

Takto načtené obrázky ještě nejsou texturami, proto musíme přiřadit tyto objekty do tříd `osg::Texture2D` a nastavit jejich vlastnosti. U displacement mapy se pak hlavně jedná o nastavení filtrování na hodnotu NEAREST.

```
osg::Texture2D *displMap = new Texture2D();
displMap->setImage(displMapImg);
displMap->setTextureSize(displMapImg->s(), displMapImg->t());
displMap->setFilter(osg::Texture2D::MIN_FILTER, osg::Texture2D::NEAREST);
displMap->setFilter(osg::Texture2D::MAG_FILTER, osg::Texture2D::NEAREST);

osg::Texture2D *colorMap = new Texture2D();
colorMap->setImage(colorMapImg);
colorMap->setTextureSize(colorMapImg->s(), colorMapImg->t());
```

Jak si lze všimnout u textury (colorMap) objektu není nastaveno filtrování. Je vhodnější ho nastavit, ale není to nezbytně nutné.

Dále budeme potřebovat nějakou plochu, na které si předvedeme displacement. Postačí rovná plocha složená se sítě polygonů. Za tímto účelem jsem si napsal funkci, jejíž zdrojový kód lze najít v příloze A. Funkce podle zadaných parametrů vygeneruje síť trojúhelníků. Jejich rozestavění je načrtnuto na obr. 4.4. Pro teď nám stačí tuto funkci zavolat. Jako parametry předáme požadovanou pozici plochy, její velikost a počet vertexů na hranu čtverce.

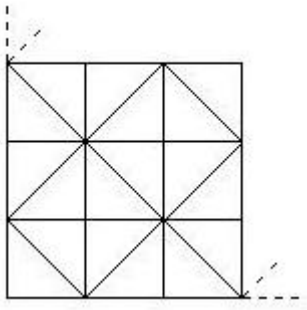
```
int textureSize = displacementMap->getTextureWidth();
osg::Vec3 position = osg::Vec3(-textureSize/2, -textureSize/2, 0.0f);
osg::Geometry *quadGeom = createQuad(position, textureSize, textureSize );
```

Takto získanou geometrii plochy přiřadíme do uzlu a ten pak připojíme ke kořenovému uzlu.

```
osg::Geode *quad = new osg::Geode;
quad->addDrawable(quadGeom);
rootNode->addChild(quad);
```

Nyní budeme chtít aplikovat na danou plochu displacement shader. Toto provedeme totožně s tím, co jsem již ukázal dříve (3.5). Jen s jedním drobným rozdílem. Pro získání objektu `osg::StateSet` budeme volat metodu třídy `osg::Geode`, která je ovšem totožná.

```
osg::StateSet *ss = quad->getOrCreateStateSet();
ss->setAttributeAndModes( ShaderProgram, osg::StateAttribute::ON);
```



Obrázek 4.4: Rozložení trojúhelníků plochy

Dále budeme potřebovat předat textury a další hodnoty do shaderů. Využijeme metod třídy `osg::StateSet`. Pro předání textur je nutné ji nejdříve přiřadit objektu a pak předat shaderu číslo, na které pozici je daná textura umístěna.

```
ss->setTextureAttributeAndModes(0, displMap, osg::StateAttribute::ON);
ss->addUniform( new Uniform("displacementMap", 0) );
ss->setTextureAttributeAndModes(1, colorMap, osg::StateAttribute::ON);
ss->addUniform( new Uniform("colorMap", 1) );
```

Ještě zbývá předat váhový faktor, pokud jej shader očekává a popřípadě i velikost textury a texelu(velikost jednoho pixelu textury).

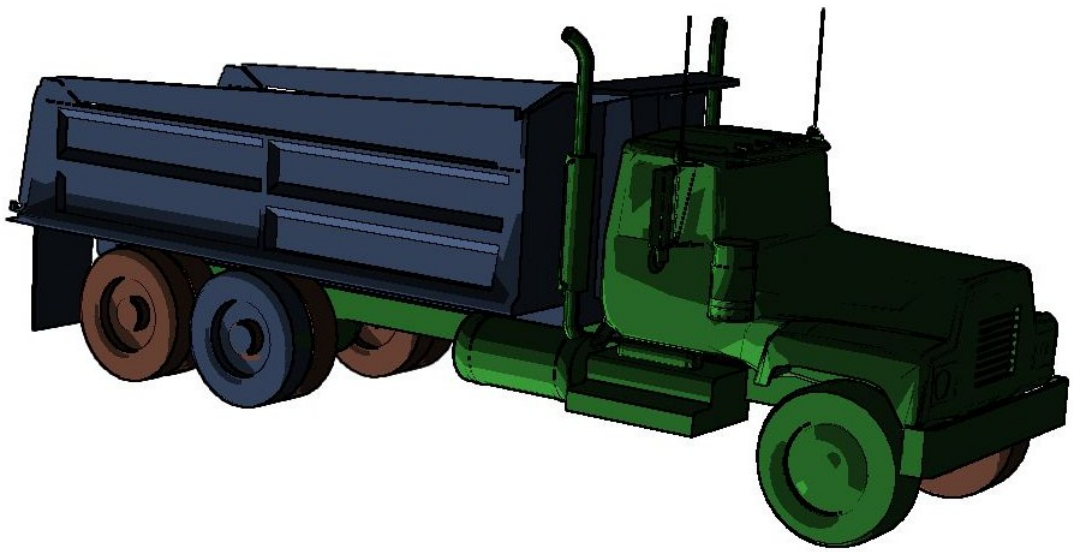
```
ss->addUniform( new Uniform("scaleFactor", 10.0f) );
ss->addUniform( new Uniform("textureSize", float(textureSize) ) );
ss->addUniform( new Uniform("texelSize", 1.0f/float(textureSize) ) );
```

Kapitola 5

Cartooning

Cartooning nebo též cel-shading (či Toon) je jeden z mála nefotorealistických efektů. Tento efekt se snaží napodobit styl klasických komiksů nebo animovaných filmů.

V dnešní době, kdy se aplikace jako např. hry snaží o co největší míru fotorealismu, se použití techniky cartooningu vidí docela vzácně. Přestože pomocí tohoto, v základní variantě relativně snadného, efektu můžeme docílit velmi zajímavých a poutavých výsledků.

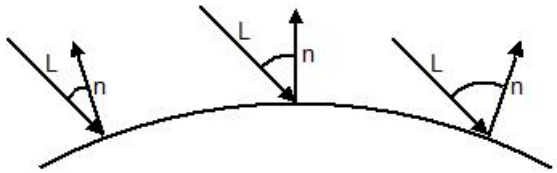


Obrázek 5.1: Ukázka Cartoon shaderu

5.1 Princip Cartooningu

Zpracování cartooningu začíná klasickým 3D modelem. Cartooning se od klascikých renderovacích technik liší právě použitím nerealistického osvětlení. Při cartooningu jsou hodnoty světla počítány klasicky pro každý pixel, poté jsou přepočítány na malý počet diskrétních odstínů, které tvoří typický plochý vzhled tak, že stínování připomíná bloky barev.

Nejjednodušší varianta tohoto efektu funguje na porovnání normály vertexu a směrnice světla. Následně podle úhlu, který tyto vektory svírají, se určí barva daného pixelu.



Obrázek 5.2: Princip světla v cartooningu

Čistě cartooning nemá žádnou možnost, jak zvýraznit obrysy objektu. Tohoto lze obecně docílit třemi metodami.

První metoda je relativně snadná a přináší docela kvalitní výsledky. Jde v podstatě o renderovací techniku. Objekty se nejdříve vykreslí jako wireframe (síť polygonů) se širšími linkami, přesněji se vykreslí jen vertexy odvrácené od kamery. Poté se vykreslí objekty normálně s využitím cartooningu. Tím, že jsme vykreslili objekty přes wireframe, zůstanou vidět z wireframe jen obrysy objektu.

Další možností, jak docílit obrysů kolem objektů, je vykreslit odvrácené vertexy, posunuté podle jejich normály, v barvě obrysu. Lze toho docílit relativně snadno pomocí vertex shaderu.

Nejzajímavější a v podstatě i nejnáročnější metodou je detekce hran. Jedná se o efekt aplikovaný až po vyrenderování scény. Tudíž nemá možnost manipulace s objekty scény, tedy nemá ani přístup k jejich vlastnostem. Více o detekci hran v následujícím oddílu.

5.2 Detekce hran

Jde o metody zpracování obrazu za účelem detekování ostrých změn v obraze. Důvodů pro použití těchto metod je spousta, od detekce změn jasu až prostou detekci hran/obrysů objektů v obraze.

V ideálním případě dostaneme po aplikování některého z operátorů sadu křivek znázorňujících okraje objektů, ale i změny v orientaci dané plochy. Bohužel ne vždy je možné získat takto dokonalé křivky z reálných obrazů (např. fotografií), což nás při použití v rámci cartooningu nemusí příliš trápit.

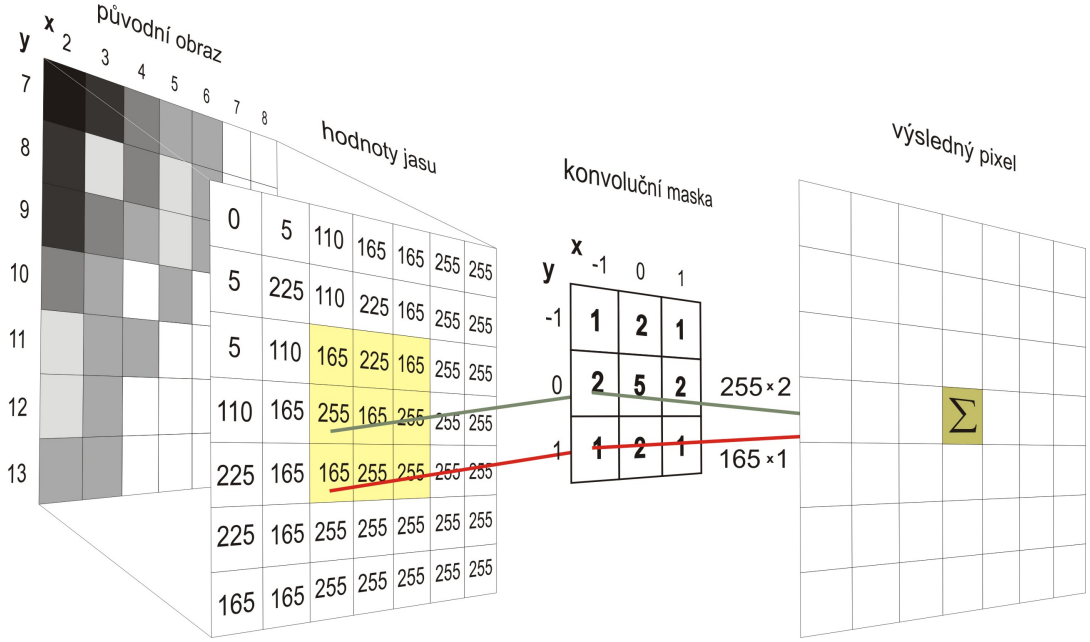
Předtím než uvedu samotné principy detekce hran, bych rád zmínil, jaké změny ve scéně pro nás přináší její použití. Abychom totiž mohli provést detekci hran, budeme muset scénu vyrenderovat do textury, která se předá detekci. A to nejen jednou. Budeme potřebovat vyrenderovat do textury scénu s aplikovaným cartooningem, pak tzv. depth buffer, tedy hloubku scény a nejlépe i mapu normál. Detekce se aplikuje na hloubku a normálovou mapu, její výsledek se poté aplikuje na texturu s objekty. Tím dostame kompletní scénu.

5.2.1 Princip detekce hran

Základem všech běžných algoritmů pro detekci hran je nějaká matice, popřípadě i více matic, která se použije jako operátor konvoluce, přesněji diskrétní konvoluce, jejíž formální zápis vypadá následovně:

$$f(x,y) * h(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x+i, y+j) \cdot h(i,j)$$

Jak vypadá tato konvoluce v praxi znázorňuje obrázek 5.3.



Obrázek 5.3: Znázornění diskrétní konvoluce [11]

Každá z matic má své vlastnosti, díky kterým se výsledek může velice lišit podle použité matice. Matice mohou a většinou i jsou citlivé na hrany/přechody pod určitým úhlem. Tento úhel se dá u většiny nejpoužívanějších matic upravit rotací matice. Součet jednotlivých prvků matice pro detekci hran je roven nule, pokud by nebyl roven nule, dostaneme matici pro blur (rozmazání). V případě operátorů s více maticemi, se při použití jen jedné z jeho matic získá přechod (gradient) místo hrany.

Matice se postupně aplikuje na všechny pixely obrazu. Obecná rovnice pro výslednou hodnotu bodu by pak vypadala následovně:

$$\begin{aligned} b_{x,y} = {} & (a_{x-1,y-1} * m_{x-1,y-1}) + (a_{x-1,y} * m_{x-1,y}) + (a_{x-1,y+1} * m_{x-1,y+1}) \\ & + (a_{x,y-1} * m_{x,y-1}) + (a_{x,y} * m_{x,y}) + (a_{x,y+1} * m_{x,y+1}) \\ & + (a_{x+1,y-1} * m_{x+1,y-1}) + (a_{x+1,y} * m_{x+1,y}) + (a_{x+1,y+1} * m_{x+1,y+1}) \end{aligned}$$

kde a je zdrojový obraz, m je maska (jádro konvoluce) a b je výsledný obraz.

5.2.2 Přehled nejpoužívanějších matic a jejich vlastností

Robert Cross 2x2 Tento operátor se skládá z páru matic. Druhá matice je vlastně první, ale otočená o 90°. Matice jsou sestaveny tak, aby reagovaly maximálně na hrany pod úhlem 45°, pro každou úhlopříčku jedna.

$$G_x = \begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix} * A \qquad G_y = \begin{vmatrix} 0 & 1 \\ -1 & 0 \end{vmatrix} * A$$

$$|G| = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y|$$

Laplaceův operátor 3x3 Tato matice a matice 5x5 detekují hrany jak vertikálně tak i horizontalně. Ovšem jsou velmi citlivé na šum.

$$G = \begin{vmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{vmatrix} * A$$

Laplaceův operátor 5x5

$$G = \begin{vmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 24 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{vmatrix} * A$$

Sobelův operátor 3x3 Tyto matice jsou navrhnuty tak, aby měly co největší odezvu pro vodorovné a svislé hrany. Po jedné matici pro daný směr. Pro změnu odezvy na úhlopříčky, lze matice rotovat o 45°. Lze využít i všech čtyřech matic zaráz. Samostatně použity, pak tyto matice vytváří gradienty.

$$G_x = \begin{vmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{vmatrix} * A \qquad G_y = \begin{vmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{vmatrix} * A$$

$$|G| = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y|$$

Prewittové operátor 3x3 Tyto matice (i matice 5x5) lze libovolně rotovat a tím získat sadu 8 matic, s relativně vysokou přesností, avšak i náročností. Tyto operátory uvádím spíše pro úplnost.

$$G_x = \begin{vmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{vmatrix} * A \qquad G_y = \begin{vmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{vmatrix} * A$$

$$|G| = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y|$$

Prewittové operátor 5x5

$$G_x = \begin{vmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{vmatrix} * A \qquad G_y = \begin{vmatrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{vmatrix} * A$$

$$|G| = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y|$$

5.3 Implementace

Díky metodě jakou budeme detekovat hrany se implementace poněkud zkomplikuje. Nebude nám stačit jeden shader program. Použijeme přesněji tří, kvůli oddělení funkčnosti, aby bylo patrné, co přesně který shader dělá.

Prvně si uvedeme samotný Cartooning, jde o jednoduchý program, zhruba rovnoměrně využívající vertex i fragment shader. Jako druhý si ukážeme opravdu triviální shader, ale pro správnou funkčnost detekce hran nezbytný, který má za účel zobrazit normály objektů. Jako poslední si pak ukážeme detekci hran, jejíž činnost se celá odehrává ve fragment shaderu.

5.3.1 Cartooning - Vertex shader

Náplní vertex shaderu bude získání směrnice světla a normály vertexu. Popřípadě i k získání barvy objektu, pokud nechce využít 1D textur pro objekty.

Prvně si tedy určíme co chceme předávat do fragment shaderu. Tedy normálu, směrnici světla a barvu objektu.

```
varying vec3 normal, lightDirection;
vyrying vec4 color;
```

Ve funkci `main()`, pak tyto hodnoty nastavíme.

```
normal = gl_NormalMatrix * gl_Normal;
lightDirection =  normalize(vec3(gl_LightSource[0].position));
color = gl_Color;
```

Místo `gl_Color`, lze též použít `gl_FrontColor` nebo, jak jsem již zmínil, bychom mohli barvu vynechat úplně.

Pro úplnost je nutné do shaderu dopsat poslední řádek, umísťující vertex.

```
gl_Position = ftransform();
```

5.3.2 Cartooning - Fragment shader

Ve fragment shaderu budeme vlastně počítat intenzitu světla, kterou využijeme pro škálování barvy, či posunu na 1D textuře. Nejdříve si převezmeme hodnoty z vertex shaderu.

V těle shaderu si pak normalizujeme normálu, směrnici světla a spočítáme úhel, který svírají, tedy intenzitu výsledné barvy.

```
vec3 norm = normalize(normal);
vec3 light = normalize(lightDirection);
float intensity = dot(light, norm);
```

Nyní díky intenzitě určíme odstín barvy. Lze toho snadno dosáhnout jednoduchou konstrukcí podmínek(`if-else`), kterou určíme koeficient. Počet odstínů si můžeme zvolit libovolně.

```
float tmp;
if (intensity > 0.95)
  tmp = 1.0;
else if (intensity > 0.75)
```

```
  tmp = 0.75;
else if (intensity > 0.5)
  tmp = 0.5;
else if (intensity > 0.25)
  tmp = 0.25;
else
  tmp = 0.1;
```

Výslednou barvu pak získáme snadno. Násobením barvy a koeficientu.

```
gl_FragColor = color * vec4(tmp, tmp, tmp, 1.0);
```

Pokud bychom chtěli použít 1D texturu, musíme si ji předat z OSG.

```
uniform sampler1D texture;
```

Tělo shaderu se nám při použití textury zkrátí. Vypustíme nepotřebnou konstrukci z podmínek. Výsledný odstín pak získáme z textury tím, že jako souřadnici použijeme přímo intenzitu, pokud je kladná.

```
gl_FragColor = texture1D(texture, max(intensity, 0.0) );
```

5.3.3 Normal shader

Jak jsem již naznačil v úvodu, tento shader je triviální. Ve vertex shaderu se spočítá normála a předá se fragment shaderu, kde se po normalizaci použije jako barva.

Vertex shader

```
varying vec3 normal;
void main(){
   normal = gl_NormalMatrix * gl_Normal;
   gl_Position = ftransform();
}
```

Fragment shader

```
varying vec3 normal;
void main(void){
   gl_FragColor = vec4(normalize(normal), 0.0);
}
```

5.3.4 Detekce hran - Vertex shader

U detekce hran vertex shader poslouží jen pro přenos souřadnic textury.

```
void main(){
  gl_TexCoord[0] = gl_MultiTexCoord0;
  gl_Position = ftransform();
}
```

5.3.5 Detekce hran - Fragment shader

Nejdříve budeme potřebovat několik hodnot z OSG. Hloubkovou mapu, normálovou mapu a mapu barev (textura s vyrenderovanou scénou). Také budeme potřebovat rozměry textury, abychom mohli spočítat offsety pro posun o jeden pixel na textuře.

```
uniform sampler2D depth, normalMap, color;
uniform float xx, yy;
```

V těle shaderu pak nejdříve spočítáme offsety, které si uložíme do pole o devíti prvcích, pokud hodláme použít matici 3x3.

```
float x = (1.0/xx);
float y = (1.0/yy);
vec2 tc_offset[9];
tc_offset[0] = vec2(-1.0*x, -1.0*y);
...
tc_offset[8] = vec2( 1.0*x,  1.0*y);
```

Takto získané offsety použijeme pro získání vzorků z hloubkové a normálové mapy. Vzorky si uložíme do samostatných polí.

```
vec4 sample[9], sample2[9];
sample[0] = texture2D(normalMap, gl_TexCoord[0].xy + tc_offset[0]);
...
sample[8] = texture2D(normalMap, gl_TexCoord[0].xy + tc_offset[8]);
sample2[0] = texture2D(depth, gl_TexCoord[0].xy + tc_offset[0]);
...
```

Pole vzorků lze naplnit pomocí cyklu `for`, avšak z vlastní zkušenosti nedoporučuji. Volání funkce `texture2D` je relativně výpočetně náročné a v kombinaci s cyklem značně prodlužuje dobu strávenou v shaderu, tím i výsledný počet snímku za sekundu.

Nyní máme vše co potřebujeme, abychom mohli určit, zda je daný bod součástí hrany. Využijeme jeden z uvedených operátorů. Popřípadě i více, jelikož každý z operátorů dosahuje trochu rozdílných výsledků. Experimentálně lze zjistit, který operátor přináší požadované výsledky pro jednotlivé mapy. Osobně mi jako nejvhodnější připadá použít Laplaceův operátor na normálovou mapu a Sobelův operátor na hloubkovou mapu.

Samotný výpočet je pak součet jednotlivých vzorků s určitou váhou podle zvoleného operátoru. Pro Laplaceův operátor by kód vypadal následovně.

```
vec4 laplacian = -  sample[0]-  sample[1]-  sample[2]
                 -  sample[3]+8*sample[4]-  sample[5]
                 -  sample[6]-  sample[7]-  sample[8];
```

U práce s normálovou mapou je podstatné zmínit, že výsledkem není barva v odstínech šedi, museli bychom převést do šedi všechny vzorky z mapy, což je až zbytečně výpočetně náročné. Získat odstín šedi lze buď výpočtem nebo použitím barevné složky s největší hodnotou, přičemž složku alpha nastavíme na nulu, kvůli pozdějšímu použití této barvy.

```
float intensity = 0.30*laplacian.r + 0.59*laplacian.g + 0.11*laplacian.b;
float intensity = max(laplacian.r, max(laplacian.g, laplacian.b));
laplacian = vec4(intensity, intensity, intensity, 0.0);
```

Sobelův operátor vypadá obdobně. Jen přibude řádek s výpočtem či aproximací hodnoty.

```
vec4 gx =   sample2[0]             -  sample2[2]
          +2*sample2[3]            -2*sample2[5]
          +  sample2[6]            -  sample2[8];
vec4 gy =   sample2[0]+2*sample2[1]+  sample2[2]
          -  sample2[6]-2*sample2[7]-  sample2[8];
vec4 sobel = sqrt( gx*gx + gy*gy );
```

Nyní stačí oba výsledky sečíst a omezit jejich rozsah na interval $<0, 1>$. Případně můžeme pomocí podmínky oříznout příliš jemné hrany.

```
vec4 tmp = clamp(sobel+laplacian, 0.0, 1.0);
if(tmp.x < 0.15)
  tmp = vec4(0.0, 0.0, 0.0, 0.0);
```

Výslednou hodnotu nyní stačí odečíst od odpovídající barvy z mapy barev. Vhodné je též opravit rozsah zpět na interval $<0, 1>$.

```
gl_FragColor = clamp(texture2D(color, gl_TexCoord[0].st) - tmp, 0.0, 1.0);
```

5.3.6 OpenSceneGraph

Díky použití několika shader programů se nám implementace scény poněkud zkomplikuje. Bude nutné scénu vyrenderovat do textury, a to nejen jednou, ale dvakrát a navíc vytvořit i hloubkovou mapu. Navíc výslednou scénu (po aplikaci detekce hran) budeme muset nějak zobrazit, nejlépe na ploše překrývajíci celý obraz, ale k tomu až později.

Stejně jako v předchozí kapitole o displacementu, i zde budeme vycházet ze základní aplikace pro OSG uvedenou dříve (3.4).

Prvním rozšířením základní aplikace bude zjištění rozlišení obrazovky. Toho lze docílit voláním funkce OSG ze jmenného prostoru `osg::GraphicsContext`.

```
unsigned int width, height;
osg::GraphicsContext::WindowingSystemInterface* wsi =
    osg::GraphicsContext::getWindowingSystemInterface();
wsi->getScreenResolution(osg::GraphicsContext::ScreenIdentifier(0),
                         width, height);
```

Načteme si ukázkový model. Například model nákladního automobilu („dumptruck.osg“) dostupného v OSG, načtení modelu jsme si ukázali v kapitole 3.4. Lze jej editovat v libovolném textovém editoru, kvůli přidání barvy k základní geometrii, což se nám může hodit, pokud nebudeme používat 1D texturu pro objekt.

Na takto načtený model potřebujeme aplikovat dva shader programy, cartooning jako takový a normálový shader. Lze to udělat několika způsoby, ukažme si jeden z těch nejjednodušších. Vytvoříme si dvě instance třídy `osg::MatrixTransform`, oběma přiřadíme načtený model.

```
osg::MatrixTransform* xform = new osg::MatrixTransform;
xform->addChild(model);
```

Nyní na tento objekt zapouzdřený v transformacích aplikujeme shadery, stejným způsobem jako v předchozích kapitolách (3.5). Na první transformaci použijeme cartooning shader, na druhou pak aplikujeme normálový shader.

```
osg::StateSet *ss = xform->getOrCreateStateSet();
CartoonProgram = new osg::Program;
 ...
ss->setAttributeAndModes(CartoonProgram, osg::StateAttribute::ON);
ss = xform2->getOrCreateStateSet();
NormalProgram = new osg::Program;
 ...
ss->setAttributeAndModes(NormalProgram, oag::StateAttribute::ON);
```

Cartooning shader pro své výpočty potřebuje mít ve scéně světlo. Zdroj světla lze přidat pomocí tříd `osg::Light` (vlastnosti světla) a `osg::LightSource` (uzel grafu pro světlo). Nejdříve tedy nastavíme požadované vlastnosti světla, v tomto případě nám bude stačit pozice světla a směr světla.

```
osg::Light* light = new osg::Light;
light->setLightNum(0);
light->setPosition(osg::Vec4(1.5f,-1.5f,1.5f,0.0f));
light->setDirection(osg::Vec3(0.0f,0.0f,0.0f));
osg::LightSource* lightSrc = new osg::LightSource;
lightSrc->setLight(light);
lightSrc->setLocalStateSetModes(osg::StateAttribute::ON);
```

Před tím než přiřadíme uzel ke kořenu, tak mu nastavíme stateset shodný s první transformací našeho modelu.

```
lightSrc->setStateSetModes(xform->getStateSet(), osg::StateAttribute::ON);
rootNode->addChild(lightSrc);
```

Takto připravené objekty budeme chtít vyrenderovat do textur. Připravíme si dvě textury ve formátu RGBA a jednu pro hloubkovou mapu. Texturám nastavíme filtrování na lineární, popřípadě nejbližší (nearest). Rozměry textur nastavíme podle zjištěného rozlišení.

```
osg::Texture2D* colorMap = new osg::Texture2D;
colorMap->setInternalFormat(GL_RGBA);
colorMap->setTextureSize(width, height);
colorMap->setFilter(osg::Texture2D::MIN_FILTER, osg::Texture2D::LINEAR);
colorMap->setFilter(osg::Texture2D::MAG_FILTER, osg::Texture2D::LINEAR);
osg::Texture2D* normalMap = new osg::Texture2D;
 ...
depthMap->setInternalFormat(GL_DEPTH_COMPONENT);
 ...
```

Textury jsou připraveny, takže můžeme vyrenderovat scénu. K tomu bude zapotřebí kamera. Kamera má v OSG množství nastavení. Většinu z nich bude nutné nastavit. Začneme vytvořením kamery, nastavením pozadí a masky. Zde zvolená barva pozadí určí barvu ve výsledné scéně.

```
osg::Camera* camera = new osg::Camera;
camera->setClearColor(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f));
camera->setClearMask(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFER_BIT);
```

Nastavíme rozměr zobrazovací plochy kamery shodně s texturou, do které budeme renderovat. Dále nastavíme kameru tak, aby se renderovala před hlavní kamerou, navíc kameře určíme, že má použít FBO (frame buffer object).

```
camera->setViewport(0, 0, width, height);
camera->setRenderOrder(osg::Camera::PRE_RENDER);
camera->setRenderTargetImplementation(osg::Camera::FRAME_BUFFER_OBJECT);
```

Nyní potřebujeme kameru umístit a zajistit, že se bude její pozice aktualizovat zaráz s hlavní kamerou. Toho dosáhneme tím, že nastavíme projekční a pohledovou matici na jednotkovou a nastavíme dědění transformace z grafu.

```
camera->setProjectionMatrix(osg::Matrixd::identity());
camera->setViewMatrix(osg::Matrixd::identity());
camera->setReferenceFrame(osg::Transform::RELATIVE_RF);
```

Takto nastavenou kameru můžeme již přidat ke kořenovému uzlu. Nejdříve ale musíme nastavit poslední podstatný detail. Musíme kameře říct, co má zobrazovat a kam má uložit výstup. Prvně tedy, co má kamera zobrazit.

```
camera->addChild(xform);
```

Následujícím řádkem řekneme kameře, že má svůj „color buffer“ ukládat do námi připravené textury `colorMap`.

```
camera->attach(osg::Camera::COLOR_BUFFER, colorMap);
```

Nyní přidáme kameru ke kořeni.

```
rootNode->addChild(camera);
```

Tímto jsme získali kameru sychronní s hlavní, která před hlavní kamerou vyrenderuje námi požadovanou scénu do textury. Nyní budeme potřebovat druhou kameru, která udělá totéž s druhou transformací objektu (s aplikovaným normálovým shaderem). Z druhé kamery můžeme též získat i hloubkovou mapu.

Vytvoříme si tedy novou kameru a nastavíme vše stejně jako u předešlé s vyjímkou barvy pozadí, kterou nastavíme na černou pro lepší kontrast s normálovým shaderem.

```
camera2->setClearColor(osg::Vec4(0.0f, 0.0f, 0.0f, 1.0f));
```

Dalším rozdílem bude řádek s připojením textury pro renderování. Renderovat do textury budeme jak „color buffer“, tak i hloubku. Poté stačí jen přiřadit správnou transformaci modelu/scény.

```
camera2->attach(osg::Camera::COLOR_BUFFER, normalMap);
camera2->attach(osg::Camera::DEPTH_BUFFER, depthMap);
camera2->addChild(xform2);
```

Nyní k zobrazení scény. Využijeme jednoduchý obdélník, zobrazený přes celou obrazovku, na který aplikujeme shader pro detekci hran. Vytvoření jednoduchého obdélníku je snadné, kód lze najít v nespočtu tutoriálů dostupných k OSG. Vytvoření toho obdélníku zapouzdříme ve funkci `createSimpleQuad()`, jejíž kód je v příloze B. Dále bude potřeba vypnout osvětlení pro tento obdélník.

```
osg::Geode* quad = createSimpleQuad(width, height);
ss = quad->getOrCreateStateSet();
ss->setMode(GL_LIGHTING, osg::StateAttribute::OFF);
```

Pro statické vykreslení obdélníku přes celý obraz budeme potřebovat ortogonální kameru. Její nastavení vypadá následovně.

```
camera_ortho->setReferenceFrame(Transform::ABSOLUTE_RF);
camera_ortho->setViewMatrix(Matrix::identity());
camera_ortho->setProjectionMatrixAsOrtho2D(0,width,0,height);
camera_ortho->setRenderOrder(Camera::NESTED_RENDER);
```

Nyní již stačí na daný obdélník aplikovat shader pro detekci hran a předat všechny potřebné uniformy, které shader očekává.

Kapitola 6

Vodní hladina

Před tím než se dostaneme k popisu samotného efektu vodní hladiny, by bylo vhodné zamyslet se jak vypadá vodní hladina v reálu, jaké jsou její vlastnosti.

Při pohledu na vodní hladinu je za prvé vidět odraz okolní krajiny. Při bližším pohledu pak i případné dno. První z optických jevů se nazývá reflekce. Druhý jev se pak nazývá refrakce. Za zmínku stojí i vlnění vodní hladiny (distortion) a třpyt na hladině jako důsledek odrazu světla.

Obrázek 6.1: Ukázka vodní hladiny

6.1 Princip vodní hladiny

U toho efektu ani zdaleka nebudeme dodržovat všechny fyzikální zákony. Půjde nám více o to, jak výsledný efekt bude vypadat, než aby byl korektní podle fyzikálních zákonů. Popišme si tedy základní optické efekty, které budeme chtít napodobit.

Refrakce je jev, který nastává při průchodu vlnění rozhraním mezi dvěmi prostředími, ve kterých má vlnění různou fázovou rychlost. Je popsána Snellovým zákonem.

$$n_1 \sin \alpha_1 = n_2 \sin \alpha_2$$

Reflekce může nastat, pokud se vlnění dostane k rozhraní dvou různých prostředí. Řídí se zákonem odrazu. „Úhel odrazu je roven úhlu dopadu, přičemž odražené paprsky zůstávají v rovině dopadu.“

$$\alpha = \alpha'$$

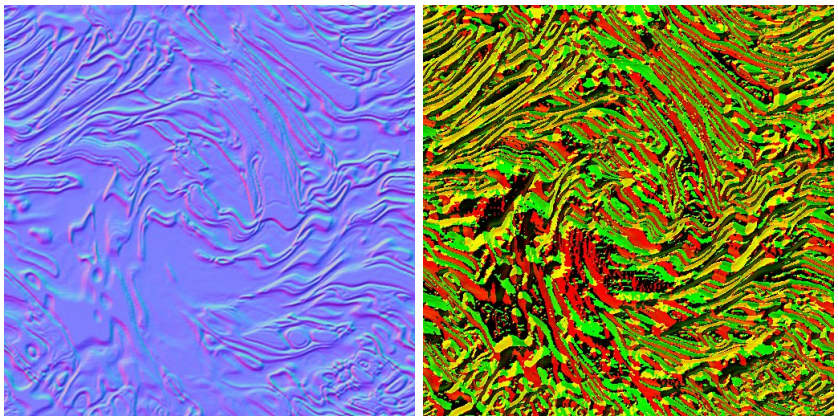
Chování vodní hladiny, tedy spojení obou těchto jevů je vyjádřeno Fresnelovými rovnicemi. Čistě pro ukázku si uveďme tyto rovnice, abychom viděli, proč se nebudeme držet přesně fyzikálních zákonů.

$$R_s = \left[\frac{\sin(\alpha_t - \alpha_i)}{\sin(\alpha_t + \alpha_i)}\right]^2 = \left[\frac{n_1 \cos \alpha_i - n_2 \cos \alpha_t}{n_1 \cos \alpha_i + n_2 \cos \alpha_t}\right]^2$$

$$R_p = \left[\frac{\tan(\alpha_t - \alpha_i)}{\tan(\alpha_t + \alpha_i)}\right]^2 = \left[\frac{n_1 \cos \alpha_t - n_2 \cos \alpha_i}{n_1 \cos \alpha_t + n_2 \cos \alpha_i}\right]^2$$

$$T_s = 1 - R_s \qquad T_p = 1 - R_p$$

Základním kamenem celého efektu vodní hladiny je normálová mapa. Texturu vody nepotřebujeme, stačí nám jen normálová mapa. Výsledný vzhled vody bude velice záviset právě na kvalitě normálové mapy a z ní odvozené dudv mapy, takovouto dvojici lze vidět a obrázku 6.2. Normálová mapa je důležitá právě proto, že z ní budeme vypočítávat fresnel, místo použití složitých Fresnelových rovnic.



Obrázek 6.2: Normálová (vlevo) a z ní odvozená dudv (vpravo) mapa

Proměnlivosti vodní hladiny docílíme pak tím, že s plynoucím časem budeme posouvat po ploše normálovou mapu a dudv mapu. Tím vytvoříme dojem pohybu vodní plochy. Směr „toku“ bude záviset na směru pohybu obou map a na rychlosti s jakou jimi budeme pohybovat.

6.2 Implementace

Efekt vodní hladiny bude z větší části naimplementován ve fragment shaderu, jelikož nejhlavnější činností je právě práce s texturami. Implementace ukázkové scény v OpenSceneGraphu bude díky předešlým kapitolám relativně snadná, jelikož jsme si větší část potřebných konstrukcí již ukázali.

6.2.1 Vertex shader

Začneme tím, co bude vertex shader očekávat od OSG. Shaderu budeme muset předat pozici kamery, pozici světla a dvě s časem se měnící hodnoty. Dále si definujeme, jaké hodnoty budeme chtít předat z vertex shaderu do fragment shaderu. Ve fragment shaderu budeme potřebovat směrnici světla, směrnici pohledu, pozici vertexu pro výpočet projekce a nakonec i dvoje koordináty pro textury.

```
varying vec4 lightVector, viewVector, vertexPosition, texMove1, texMove2;
uniform vec4 viewPos, lightPos;
uniform vec3 time, time2;
```

Vertex shader nebude potom o ničem jiném než o naplnění hodnot pro fragment shader. Začneme směrnicí světla, kterou spočítáme z pozice vertexu a pozice světla. Obdobně spočítáme i směrnici kamery.

```
lightVector = gl_Vertex - lightPos;
viewVector = viewPos - gl_Vertex;
```

Jako další potřebujeme získat souřadnice pro textury a pozici vertexu. Pro výpočet souřadnic použijeme ony s časem měnící se hodnoty `time`, `time2`. Nakonec pak nastavíme pozici vertexu.

```
texMove1 = gl_MultiTexCoord0 + vec4(time, 0.0);
texMove2 = gl_MultiTexCoord0 + vec4(time2, 0.0);
vertexPosition = gl_ModelViewProjectionMatrix * gl_Vertex;
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

6.2.2 Fragment shader

Náplň fragment shaderu bude poněkud zajímavější než tomu bylo u vertex shaderu. Zde budeme vytvářet samotný efekt vodní hladiny.

Nejdříve si opět definujeme uniformy, které budeme očekávat od aplikace. V tomto případě to bude pět textur. Normálová mapa, dudv mapa, reflekce, refrakce, hloubková mapa. Navíc si budeme očekávat i vektor s barvou vody.

```
uniform smapler2D water_normalmap, water_dudvmap;
uniform sampler2D water_reflection, water_refraction, water_depthmap;
uniform vec4 waterColor;
```

Definujeme si hodnoty předávané z vertex shaderu (6.2.1). Nyní můžeme přejít k samotnému shaderu. První věc, kterou je potřeba udělat je normalizování směrnic světla a pohledu.

```
vec4 light = normalize(lightVector);
vec4 viewt = normalize(viewVector);
```

Pusťme se nyní do výpočtu zakřivení. Prvně použijeme jedny texturové koordináty pro získání náhledu na dudv mapu. Hodnotu tohoto bodu dudv mapy pak použijeme pro získání zakřivení v kombinaci s druhými texturovými koordináty. Výsledné zakřivení pak získáme normalizací a následným vynásobením přiměřeně malým číslem, kupříkladu pěti tisícinami.

```
vec4 partialDistortion = texture2D(water_dudvmap, vec2(texMove1*timeScale));
vec4 distortion = texture2D(water_dudvmap,
                    vec2(texMove2 + partialDistortion*texCoordScale));
vec4 finalDistortion = normalize(distortion) * 0.005;
```

Dále budeme potřebovat hodnotu z normálové mapy. Souřadnice opět budeme modifikovat hodnotou z dudv mapy. Získanou hodnotu z normálové mapy, budeme chtít mírně upravit, takže před její normalizací je vhodné od normálové mapy odečíst nějakou hodnot, popřípadě i následně vynásobit. Tím dosáhneme zvýraznění normály. Optimální čísla je nutné zjistit experimentálně, jsou závislé jednak na použité normálové mapě, ale i na subjektivním pocitu pozorovatele.

```
vec4 tmp = texture2D(water_normalmap,
              vec2(texMove2 + partialDistortion*texCoordScale) );
tmp = (tmp - vec4(0.25,0.25,0.25,1.0)) * vec4(2.0,2.0,2.0,1.0);
vec4 normalMap = normalize(tmp);
```

Před tím, než se podíváme na zpracování reflekce a refrakce, si spočítáme projekční koordináty, podle kterých budeme reflekci a refrakci zobrazovat. Výpočet projekce je snad dostatečně znám, jen dodám, že k projekčním koordinátům přičteme zakřivení. Nakonec zajistíme, že souřadnice nebudou nikdy nabývat mezních hodnot, abychom zamezili nepříjemným chybám na okraji map.

```
vec4 projCoord = vertexPosition - vertexPosition.w;
projCoord += vec4(1.0);
projCoord *= vec4(0.5);
projCoord = projCoord + finalDistortion;
projCoord = clam(projCoord, 0.001, 0.999);
```

Dalším krokem po výpočtu projekčních souřadnic je získaní hloubky a její inverzní hodnoty. Hodnoty hloubkové mapy mívájí též tendeci nabývat hodnot blízkých horní mezi intervalu $<0, 1>$. Lepšího rozprostření lze dosáhnout umocněním hodnoty, opět záleží na scéně a subjektivním hodnocení pozorovatele.

```
vec4 waterDepth = texture2D(water_depthmap, vec2(projCoord));
waterDepth = vec4(pow(waterDepth.x, 4.0));
vec4 invertDepth = 1.0 - waterDepth;
```

Hodnotu z normálové mapy získanou o něco dříve, použijeme spolu s vektorem pohledu k výpočtu fresnelu a jeho inverzní hodnoty.

```
vec4 fresnel = vec4( dot(normalMap, viewt) );
vec4 invertFresnel = vec4(1.0) - frensel;
```

Nyní máme již vše potřebné ke zpracování reflekce a refrakce. Reflekci získáme vynásobením odpovídající hodnoty z textury hodnotou fresnelu. Refrakce je pak kombinací refrakční textury, inverzní hodnoty fresnelu a inverzní hloubky.

```
vec4 reflection = texture2D(water_reflection, vec2(projCoord));
reflection *= fresnel;
vec4 refraction = texture2D(water_refraction, vec2(projCoord));
refraction *= invertFresnel;
refraction *= invertDepth;
```

Hladina vody se pak získá sečtením reflekce a refkrace. Nejdříve však je vhodné přidat zabarvení vody. Nelze však jen přičíst barvu vody, barvu budeme muset mírně upravit, to lze provést vynásobením hloubkou a fresnelem.

```
tmp = waterColor * waterDepth * invertFresnel;
refraction += tmp;
vec4 water = refraction + reflection;
```

Poslední věc, kterou zbývá určit je odlesk světla od hladiny. Využijeme k tomu funkci GLSL `reflect()`, které předáme směrnici světla a hodnotu z normálové mapy, tím získáme vektor odrazu světla. Tento vektor pak porovnáme s úhlem pohledu. Čím blíže jsou tyto vektory, tím větší odlesk získáme. Výsledek pak sečteme s vodní hladinou.

```
tmp = normalize(reflect(light, normalMap));
float temp = max(0.0, dot(viewt, tmp) );
temp = pow(temp, exponent);
vec4 highlight = vec4(temp);
gl_FragColor = water + highlight;
```

6.2.3 OpenSceneGraph

Popsat celou implementaci ukázkového programu není nutné, větší část programu jsme si již ukázali v předcházejících kapitolách o displacementu a cartooningu.

Opět vyjdeme ze základní aplikace uvedené v 3.4. Zjistíme si rozlišení obrazovky (5.3.6). Připravíme si potřebné textury. Budeme potřebovat tři textury načtené ze souboru. Normálovou mapu vody, takzvanou dudv mapu vody a texturu pro dno. Jak načíst textury ze souboru jsme si již ukázali v 4.2.3. U těchto textur budeme potřebovat nastavit opakování. Provedeme tak voláním metody.

```
normalMap->setWrap(osg::Texture2D::WRAP_S, osg::Texture2D::REPEAT);
normalMap->setWrap(osg::Texture2D::WRAP_T, osg::Texture2D::REPEAT);
```

Dále si připravíme textury, do kterých budeme renderovat scénu. Budeme potřebovat celkem tři, pro reflekci, refrakci a hloubku. Přípravu textur pro tento účel jsme si již ukázali (5.3.6).

Pro demostraci můžeme využít model ze základního OSG programu. Dále budeme potřebovat nějaké dno, abychom viděli, že funguje refrakce. Vytvoříme si jednoduchý čtverec voláním funkce `createSurface()`, její kód lze najít v příloze C a přidáme mu texturu.

```
osg::Geometry* groundDraw =
    createWaterSurface(20.0f, -20.0f, 20.0f, -20.0f, -5.0f);
osg::StateSet* stateset = groundDraw->getOrCreateStateSet();
stateset->setTextureAttributeAndModes(0, groundTex, StateAttribute::ON);
Geode* ground = new osg::Geode;
ground->addDrawable(groundDraw);
```

Dále budeme potřebovat světlo. Jeho vytvoření jsme si opět již ukázali v implemetaci cartooningu (5.3.6). Světlo, model i dno můžeme připojit ke kořenovému uzlu, jelikož je rozhodně budeme chtít všechny zobrazit.

Určíme si hladinu vody tím, že si ji uložíme do konstanty typu float. Budeme ji několikrát potřebovat.

Nyní přejdeme k vytvoření ořezové plochy a k vyrenderování reflekce. Ořezovou plochu neboli clip plane vytvoříme pomocí odpovídající třídy `osg::ClipPlane`, tu pak přidáme do odpovídajícího uzlu (`osg::ClipNode`). Jelikož této ploše předáme k zobrazení otočenou scénu, tak chceme, aby bylo viditelné vše pod touto plochou.

```
osg::ClipPlane* clipplane = new osg::ClipPlane;
  clipplane->setClipPlane(0.0,0.0,-1.0,-z_water);
  clipplane->setClipPlaneNum(0);
osg::ClipNode* clipNode = new osg::ClipNode;
  clipNode->addClipPlane(clipplane);
```

Otočenou scénu/model získáme pomocí transformace, které k zobrazení předáme scénu, v našem případě načtený model a světlo.

```
osg::StateSet* dstate = clipNode->getOrCreateStateSet();
osg::MatrixTransform* reverseMatrix = new osg::MatrixTransform;
  reverseMatrix->setStateSet(dstate);
  reverseMatrix->preMult( osg::Matrix::translate(0.0f,0.0f,z_water)*
                          osg::Matrix::scale(1.0f,1.0f,-1.0f)*
                          osg::Matrix::translate(0.0f,0.0f,-z_water));
  reverseMatrix->addChild(loadedModel);
  reverseMatrix->addChild(light);
  clipNode->addChild(reverseMatrix);
```

Takto připravenou „clip plane“ předáme k zobrazení kameře, která výsledek vyrenderuje do textury. Nastavení takovéto kamery bylo opět již zmíněno dříve v implementaci cartooningu (5.3.6).

Reflekci bychom měli, přejděme k refrakci. Postup je obdobný jako u reflekce, jen nemusíme otáčet scénu. Vytvoříme si tedy novou clip plane se stejným nastavením. Jen jí předáme scénu bez úprav.

```
clipNode2->addChild(loadedModel);
clipNode2->addChild(light);
clipNode2->addChild(groundPlane);
```

Přichystáme si druhou kameru pro renderování do textury. Předáme ji k zobrazení druhou clip plane. Z této kamery budeme též chtít získat hloubkovou mapu. Oboje jsme si ukázali již dříve (5.3.6).

V tomto okamžiku máme všechny potřebné textury. Můžeme tedy vytvořit plochu vody a aplikovat na ni shader (3.5).

```
osg::Geometry* waterGeom = createSurface(-10.0, 10.0, -10.0, 10.0, -z_water);
osg::Geode* waterGeode = new osg::Geode;
waterGeode->addDrawable(waterGeom);
osg::StateSet *ss = waterGeode->getOrCreateStateSet();
WaterProgram = new Program;
...
```

Nyní předáme všechny potřebné uniformy, které shader očekává. Menší problém ovšem nastane u uniformů `time`, `time2` a `viewPos`. Tyto musí být aktualizovány za chodu aplikace. Toho lze docílit použitím callbacků, zmíněných v kapitole 3.3.4.

```
viewpos->setUpdateCallback(new updateViewPos( &viewer ));
osg::Vec3 diff = osg::Vec3(1.0f/2512.0f, 1.0f/2512.0f, 1.0f/2512.0f);
time1->setUpdateCallback( new updateUniformTime( diff ));
time2->setUpdateCallback( new updateUniformTime( -diff ));
```

Samotný callback pro aktualizaci pozice kamery je snadný. Získáme ho vytvořením vlastní třídy odvozené z `osg::Uniform::Callback`.

```
class updateViewPos : public osg::Uniform::Callback{
protected:
  osgViewer::View* view;
public:
  updateViewPos( osgViewer::Viewer* v) : view(v) {}
  virtual void operator()(osg::Uniform* uniform, osg::NodeVisitor* nv){
    osg::Vec3 eye, center, up;
    view->getCamera()->getViewMatrixAsLookAt( eye, center, up, 1.0 );
     uniform->set(osg::Vec4(eye, 0.0));
  }
};
```

Obdobným způsobem získáme i callback pro aktualizaci času. Jako parametr konstruktoru si předáme vektor, který se má při každém zavolání přičíst k počítadlu uchovaném v callbacku.

Kapitola 7

Závěr

Práce s knihovou OpenSceneGraph je oproti čistému OpenGL mnohem snažší, ale jen do jisté míry. Řešení problému vzniklého v OpenGL, lze po většinou velice snadno dohledat na webu, kdežto problém v OSG musí člověk ve většině případů vyřešit sám.

Další výhodou či nevýhodou, záleží na úhlu pohledu, je ono ohromné množství různých tříd, které jsou součástí OSG. Jejich množství a místy až příliš stručná dokumentace ovšem nutí k neustálému experimentování a opakujícímu se procházení zdrojových textů ukázkových programů, jejichž komentáře mnohdy řeknou více než samotný popis tříd v dokumentaci.

Na druhou stranu není moc věcí, které by bylo třeba vysloveně vymýšlet. Pokud něco potřebujete, je to s největší pravděpodobností již součástí knihovny, avšak občas to najít, může trvat skoro stejně dlouho, jako bychom to měli znovu naimplementovat.

Naproti tomu práce v OpenGL Shading Language je vysloveně přímočará. Na webu lze najít spoustu kvalitních i těch méně kvalitních návodů, jak vytvářet různé efekty. Dokumentace tohoto jazyka je velice detailní a lze v ní najít opravdu vše.

Osobně si myslím, že koncept OpenSceneGraphu je velice zajímavý a má budoucnost. Jistě se časem objeví další podobné systémy na podobných principech jako OSG. Možná s další verzí OpenGL.

Literatura

[1] Bill Green: Edge Detection Tutorial. http://www.pages.drexel.edu/ weg22/edge.html, 2002, [cit. 15.května 2009].

[2] Kessenich, J.: The OpenGL®Shading Language. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf, [cit. 15.května 2009].

[3] Kolektiv autorů: Hypermedia Image Processing Reference 2 [online]. http://homepages.inf.ed.ac.uk/rbf/HIPR2/, 2003, [cit. 15.května 2009].

[4] Rost, R. J.: *OpenGL®Shading Language, Second Edition*. Addison Wesley Professional, 2006, iSBN 0-321-33489-2.

[5] WWW stránky: Bonzai Software. http://www.bonzaisoftware.com/, [cit. 15.května 2009].

[6] WWW stránky: GLSL Programming. http://www.ozone3d.net/tutorials/index_glsl.php, [cit. 15.května 2009].

[7] WWW stránky: GLSL Tutoial. http://www.lighthouse3d.com/opengl/glsl/, [cit. 15.května 2009].

[8] WWW stránky: NVIDIA Developer Web Site. http://developer.nvidia.com/page/home.html, [cit. 15.května 2009].

[9] WWW stránky: OpenSceneGraph. http://www.openscenegraph.org/, [cit. 15.května 2009].

[10] WWW stránky: Wikipedia - Cel-shaded animation. http://en.wikipedia.org/wiki/Cel-shaded_animation, [cit. 15.května 2009].

[11] WWW stránky: Wikipedia - Konvoluce. http://cs.wikipedia.org/wiki/Konvoluce, [cit. 15.května 2009].

Příloha A

createQuad

```
osg::Geometry* createQuad(osg::Vec3 origin, float plane_size, int quads){
  float offset = plane_size / quads;
  float tc_offset = 1.0f / quads;
  osg::Geometry* quadGeometry = new osg::Geometry;
  osg::Vec3Array* quadVertices = new osg::Vec3Array;
  osg::Vec2Array* texcoords = new osg::Vec2Array;
  bool direct = true;
  int quad_num = 0;
  osg::DrawElementsUInt* triBase = NULL;
  triBase = new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
  for(int y = 0; y <= quads; y++){
    bool test = direct;
    for(int x = 0; x <= quads; x++){
      quadVertices->push_back(
          osg::Vec3(origin.x()+x*offset, origin.y()+y*offset, origin.z()));
      texcoords->push_back(osg::Vec2(x*tc_offset, y*tc_offset));
      if(x < quads && y < quads){
        int front_left = quad_num + y;
        int front_right = front_left + 1;
        int back_left = front_right + quads;
        int back_right = back_left + 1;
        triBase->push_back(back_left);
        triBase->push_back(front_left);
        triBase->push_back(direct ? front_right : back_right);
        triBase->push_back(direct ? back_left : front_left);
        triBase->push_back(front_right);
        triBase->push_back(back_right);
        direct = !direct; quad_num++;
    } }
    if(test == direct) direct = !direct;
  }
  quadGeometry->setVertexArray(quadVertices);
  quadGeometry->setTexCoordArray(0, texcoords);
  quadGeometry->addPrimitiveSet(triBase);
  return quadGeometry; }
```

Příloha B

createSimpleQuad

```
osg::Geometry* createSimpleQuad(float x, float y){
  osg::Geometry* quadGeometry = new osg::Geometry();
  osg::Vec3Array* quadVertices = new osg::Vec3Array;
  quadVertices->push_back( osg::Vec3( 0, 0, 0) );
  quadVertices->push_back( osg::Vec3( x, 0, 0) );
  quadVertices->push_back( osg::Vec3( x, y, 0) );
  quadVertices->push_back( osg::Vec3( 0, y, 0) );
  quadGeometry->setVertexArray( quadVertices );
  osg::Vec4Array  * c = new osg::Vec4Array( 1 );
  (*c)[0] = osg::Vec4( 1, 0,0, 1 );
  quadGeometry->setColorArray( c );
  quadGeometry->setColorBinding( osg::Geometry::BIND_OVERALL );
  osg::DrawElementsUInt* quadBase =
    new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
  quadBase->push_back(3);
  quadBase->push_back(2);
  quadBase->push_back(1);
  quadBase->push_back(0);
  quadGeometry->addPrimitiveSet(quadBase);
  osg::Vec2Array* texcoords = new osg::Vec2Array(4);
  (*texcoords)[0].set(0.0f,0.0f);
  (*texcoords)[1].set(1.0f,0.0f);
  (*texcoords)[2].set(1.0f,1.0f);
  (*texcoords)[3].set(0.0f,1.0f);
  quadGeometry->setTexCoordArray(0,texcoords);
  return quadGeometry;
}
```

Příloha C

createSurface

```
osg::Geometry* createWaterSurface
  (float xMin, float xMax, float yMin, float yMax, float z){
    osg::Geometry* geom = new osg::Geometry;
    osg::Vec3Array* coords = new osg::Vec3Array(4);
    (*coords)[0].set(xMin,yMax,z);
    (*coords)[1].set(xMin,yMin,z);
    (*coords)[2].set(xMax,yMin,z);
    (*coords)[3].set(xMax,yMax,z);
    geom->setVertexArray(coords);
    osg::Vec3Array* norms = new osg::Vec3Array(1);
    (*norms)[0].set(0.0f,0.0f,1.0f);
    geom->setNormalArray(norms);
    geom->setNormalBinding(osg::Geometry::BIND_OVERALL);
    osg::Vec2Array* tcoords = new osg::Vec2Array(4);
    (*tcoords)[0].set(0.0f,1.0f);
    (*tcoords)[1].set(0.0f,0.0f);
    (*tcoords)[2].set(1.0f,0.0f);
    (*tcoords)[3].set(1.0f,1.0f);
    geom->setTexCoordArray(0,tcoords);
    osg::Vec4Array* colours = new osg::Vec4Array(1);
    (*colours)[0].set(1.0f,1.0f,1.0,1.0f);
    geom->setColorArray(colours);
    geom->setColorBinding(osg::Geometry::BIND_OVERALL);
    geom->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::QUADS,0,4));
    return geom;
}
```

Příloha D

Obsah CD

Na přiloženém CD se nachází ukázkové programy popsané v této práci. Ve složce `src` jsou umístěny zdrojové texty těchto programů a všechny soubory použité danými programy, včetně souboru pro Microsoft Visual Studio (tzv. solution). O správném nastavení solution pro Microsoft Visual Studio se lze dočíst více v souboru `readme.txt`.

Zdrojové texty shader programů končí koncovkou `.vert` pro vertex shader a `.frag` pro fragment shader.

Složka `osg` obsahuje archivy s distribucí knihovny OpenSceneGraph.

Ve složce `bin` jsou pak umístěny spustitelné verze ukázkových programů pro operační systém Windows.

Složka `latex` obsahuje zdrojové texty této technické zprávy. Zpráva ve formátu PDF je umístěna přímo na cd.

Ve složce `tutorial` lze pak najít webovou stránku návodu (tutoriálu) k implementovaným shaderům. Úvodní stránka je tvořena souborem `index.html`.