VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

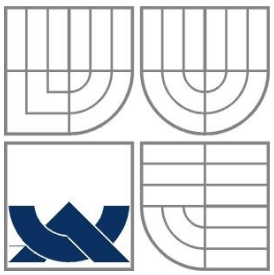
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# PROGRAMOVATELNÉ SHADERY V OPENSCENEGRAPH

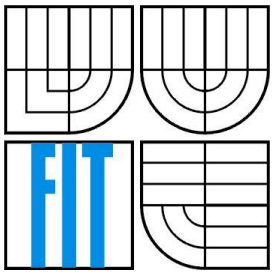
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE ZSOLT CZOMPÁL
AUTHOR

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

# PROGRAMOVATELNÉ SHADERY V OPENSCENEGRAPH

PROGRAMMABLE SHADERS IN OPENSCENEGRAPH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ZSOLT CZOMPÁL

VEDOUCÍ PRÁCE
SUPERVISOR

ING. MIROSLAV ŠVUB

BRNO 2009

# Abstrakt

Dnešní grafické procesory mají programovatelné jednotky, tzv. shadery. Programováním těchto shaderů dosáhneme lepších výsledků, něž kdyby náš grafický efekt nebo program běžel na procesoru. Existují různé programovací jazyky právě pro programování shaderů. V této práci byl zvolen jazyk GLSL (OpenGL Shading Language) s OpenSceneGraph.

Hlavním úkolem následujících kapitol bude demonstrace práce se shadermi v OpenSceneGraph a implementace vybraných grafických efektů (anisotropic lighting, morphing, vlnění trávy).



# Abstract

Graphic processors, that are used nowadays, have programmable units called shaders. By programming these units we can achieve better results in graphical applications, than by running the effect or the program on the processor. There are different programming languages for programming these units. In this thesis I have chosen the language GLSL with OpenSceneGraph API.

The main point of the following capitols is to demonstrate how do shaders work with OpenSceneGraph, and the implementation of common graphic effects.



# Klíčová slova

OpenSceneGraph, GLSL, vykreslovací řetězec, vertex shader, fragment shader, anizotropické osvětlení, vertex morphing, animovaná tráva

# Keywords

OpenSceneGraph, GLSL, rendering pipeline, vertex shader, anisotropic lighting, vertex morphing, animated grass


# Citace

Czompál, Zsolt: *Programovatelné shadery v OpenSceneGraph*. Bakalářská práce, Brno, FIT VUT v Brně, 2009.

# Programovatelné shadery v OpenSceneGraph

## Prohlášení

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

……
Zsolt CZOMPÁL
20. 5. 2009

## Poděkování

Chtěl bych se poděkovat vedoucímu práce, Ing. Miroslavovi Švubovi, za poskytnutí pomoci a podkladů a za podporu, kterou projevoval po celou dobu vytvoření této práce.

# Obsah

# 1 Úvod

Programy, ktoré používajú prvky počítačovej grafiky môžeme dnes nájsť v skoro každom počítači. Najväčšiu časť tvoria počítačové hry, dizajnérske, CAD a vizualizačné programy. Väčšina animovaných filmov sa dnes vyrábajú už renderovaním grafických scén a práca grafika a animátora sa zmenila z kreslenia na programovanie. Vo filmoch, hrách a dizajnérskych programoch nájdeme také špeciálne efekty, ktoré si pred pätnástimi rokmi programátori nemohli ešte ani predstaviť.

Vývoj grafických kariet sa začal ešte v osemdesiatych rokoch min. storočia. Vtedy *grafické adaptéry* (nemôžeme ešte hovoriť o samostatných grafických kartách) boli súčasťou základných dosiek. Pamäť adaptéru, tzv. *video pamäť* bola tiež integrovaná do základnej dosky a mala kapacitu 64 kB. Zobrazenie na výstupnom monitore bolo čiernobiele a adaptér pracoval len v tzv. *textovom režime*. Na konci osemdesiatych rokov bol grafický adaptér už samostatný komponent a komunikoval s procesorom cez systémovú zbernicu. Zbernica bola veľmi pomalá a kapacita video pamäte (približne 256 kB) ešte neumožnila *grafický režim* pre výstup. Aj po zavedení zbernice *ISA* bola komunikácia medzi adaptérom a procesorom ešte stále veľmi pomalá. Preto do grafického adaptéru zabudovali grafický čip s video pamäťou o kapacite 2MB. Rýchly vývoj hardwarových komponentov znamenal, že grafické adaptéry dostali pamäte s väčšou kapacitou, rýchlejšie grafické čipy a po objavení systémovej zbernice *PCI* už nebol problém ani s rýchlosťou komunikácie s procesorom. Napriek tomu, že grafické adaptéry boli už „výkonné", mali veľké obmedzenia týkajúce sa výkonu. Boli veľmi drahé, preto počítače s výkonnými grafickými čipmi sa našli len vo veľkých firmách alebo výskumných laboratóriách.

Prvá grafická karta, ktorá sa rozšírila aj do domácich počítačov bola karta *Voodoo* od spoločnosti *3Dfx*. Karty podporovali *grafický API Glide*, čo umožnilo programátorom aby písali programy priamo pre grafický čip *Voodoo* bez používania *OpenGL* alebo *DirectX*. Spoločnosť *3Dfx* odkúpila firma *NVIDIA*, ktorá vydala prvú grafickú kartu (rady *GeForce3*) s programovateľnými *shader* jednotkami.

V grafických *čipoch NV20* programovateľné jednotky nahradili tzv. *fixed pipeline*. Čip podporoval programovateľné *vertex* a *pixel shadery*, ale tieto jednotky umožnili ešte len implementáciu základných osvetľovacích modelov a *per-pixel* programov.

Od vydania karty *GeForce3* sa vývoj grafických čipov a programovateľných shaderov zrýchlil, výrobcovia vydávajú nové čipy každých 6 mesiacov. Vedľa vertex a fragment shaderov (v *OpenGL* pixel shader sa nazýva *fragment shader*) sa „objavil" aj *geometry shader*, čo umožňuje programátorom generovať ďalšie vertexy, body, čiary a trojuholníky ku vstupným polygónom grafického pipeline. Pre podporu programovania shaderov vznikli vysokoúrovňové programovacie jazyky, ako napríklad *HLSL* k *Direct3D, GLSL* k *OpenGL, Cg* od *NVIDIA*, ktoré nahradili nízkoúrovňový assemblerovský jazyk.

Moderné grafické čipy umožnili programátorom vytvoriť fotorealistické obrázky a filmy, počítačové hry v 3D, ktoré modelujú realitu skoro dokonale.

## 1.1 Motivácia

Knižnica *OpenSceneGraph* je čisto objektovo orientovaný grafický toolkit pre modelovanie 3D scén, ktorá nahradí trošku zastarané *OpenGL*. S objektovo orientovanými prvkami a zabudovanými funkciami vytvorenie 3D scén je rýchlejšie, ako v *OpenGL*.
Shadery zvolené na implementáciu predstavujú základnú funkčnosť programovateľných shader jednotiek v oblasti osvetľovania, práce s vertex a fragment shaderom a textúrami.

# 2 Základy programovateľných shaderov

Programovateľné jednotky v grafických čipoch umožňujú programátorom aby vytvorili špeciálne grafické efekty ku 3D scénam. Tieto efekty môžu byť veľmi rôzne, napr. osvetľovanie scény, špecifické textúrovanie (*bump mapping, normal mapping, parallax mapping*), animácia scény (*morphing*). Programovanie shaderov znamená zmenu vo fixnom vykresľovacom reťazci (*fixed pipeline*) grafických čipov. Je možnosť zmeniť správania vertex a fragment procesoru, a u novších kariet aj u geometry procesoru, ale s tým sa v tejto práci zaoberať nebudem. Zmena fixnej funkcionality grafického reťazca znamená, že scénu zo vstupu reťazca zmeníme podľa potreby, aby sme dostali na výstup požadovaný grafický efekt. Zjednodušene môžeme s vertex a fragment shaderom zmeniť pozíciu a farbu vertexu scény.

Pre pochopenie nasledujúcich kapitol je najprv nutné vysvetliť si pár pojmov z 3D počítačovej grafiky. Základné elementy každej scény sú body alebo vrcholy, z ktorých pozostávajú rôzne geometrické útvary. Tieto vrcholy sa nazývajú ***vertexy***. Tieto vertexy sú spojené ***hranami***, ktoré ohraničujú ***polygony***. V počítačovej grafike sa najčastejšie používajú troj-, alebo štvoruholníky. Povrch nejakého modelu pozostáva z trojuholníkov. Vhodným počtom vertexov (vyšší počet vertexov znamená rozpracovanejší povrch) je dosiahnuteľné, aby model vypadal reálne. Vertexy, hrany a polygóny sa nachádzajú v priestore ***3D world space***. Je to modelovací priestor reálneho sveta, kde je možné postaviť scénu. Scéna sa nachádza v ortogonálnom súradnicovom systéme, s osami *x, y* a *z*. Môžeme si to predstaviť tak, že sa na obrazovke smerom zľava doprava zmení *x*-ová, smerom zdola nahoru *y*-ová, a s hĺbkou obrazu zmení *z*-ová súradnica. Smery *y* a *z* sú v niektorých systémoch vymenené, napr. v *OpenSceneGraph* *z* je zdola nahor a *y* je hĺbka. Stred súradnicového systému je stredom *world space*. Ďalším priestorom, ktorý sa používa v grafike je tzv. ***eye space***. Od predchádzajúceho priestoru sa líši v tom, že scéna je transformovaná do súradnicovému systému, ktorý je závislý na pozícii virtuálnej ***Kamery***. Transformovaná scéna bude vyzerať ako keby sme sa na ňu pozerali z bodu kamery. Posledný priestor je ***2D image space*** čo je *eye space* mapovaný do dvoch dimenzii. Scéna v tomto priestoru je vlastne obrázok, ktorý vidíme z pozície kamery.

Renderovací reťezec (*rendering pipeline*) je model, ktorý popíše aké kroky potrebuje grafický systém na renderovanie nejakej scény na obrazovku monitora.


Reťazec môžeme zjednodušiť na štyri časti:

- **Aplikácia**: predstavuje zdroj grafických dát, výstupom sú body, čiary, polygóny.
- **Geometria**: v tejto časti pracuje systém s polygónmi a s vertexmi. Časť môžeme rozdeliť na päť krokov:
    - Transformácie pre modelovanie: v *3D world space* sa vykoná transformácia a rotácia scény.
    - Per-vertex osvetlenie: osvetlenie jednotlivých vertexov podľa zdroja alebo zdrojov svetla.
    - Vizuálne transformácie: z *3D world space* sa transformuje scéna do *eye space*, výsledkom je scéna viditeľná z pozície kamery.
    - Transformácie projekčné: z *eye space* sa transformuje do *2D image space*, mapovaním 3D scény do roviny viditeľnej z kamery.
    - Odstrihnutie: časť scény, ktorá nebude viditeľná sa odstriháva. Tento krok nie je nutný, ale urýchľuje renderovanie, lebo neviditeľné časti netreba rastrovať.
- **Rastrovanie**: *2D image space* je konvertovaný do rastrového formátu a aplikuje sa aktuálne rozlíšenie monitora.
- **Monitor**: výstupom na monitore sú farebné pixely.

Renderovací reťazec je mapovaný do grafického čipu, jeho vstupom sú vertexy. Tieto vertexy budú spracované transformáciami a per-vertex osvetlením. V tomto bode môžeme aplikovať vertex shader na vlastné transformácie a osvetlenie vertexov. Po rasterizácii môžeme aplikovať fragment shader a zmeniť farbu fragmentov. Výstupom budú pixel hodnoty zapísané do *frame bufferu* a zobrazené na monitore.

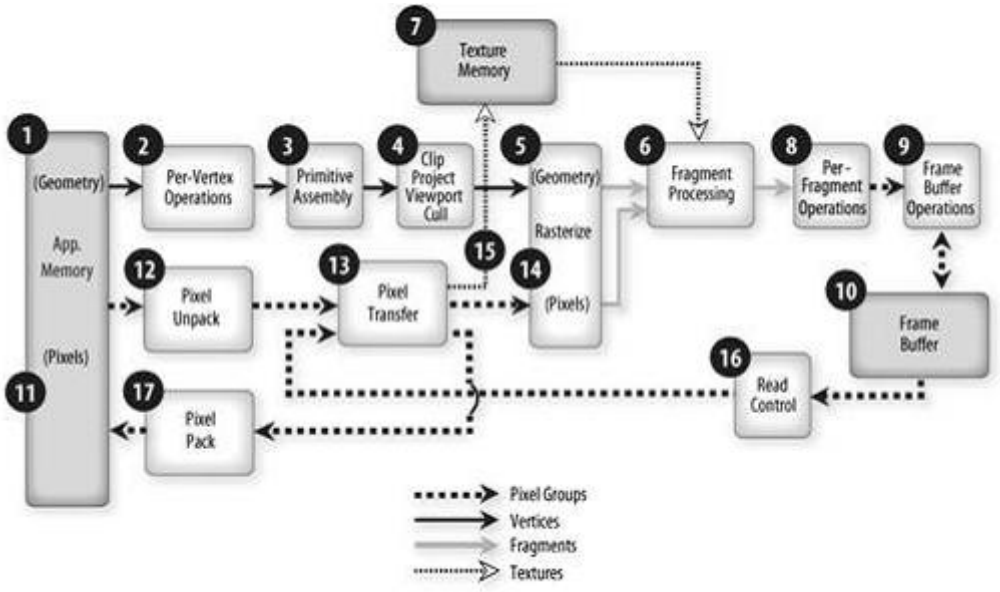
Vykresľovací reťazec je vhodný pre renderovanie, lebo umožní grafickému čipu *prúdové spracovanie* vstupných vertexov a fragmentov. Dnešné grafické čipy majú viac shader jednotiek – takto je možné zvýšiť výkon čipu (napr. AMD Radeon 4870 má 800 unifikovaných shader jednotiek).

Na programovanie grafiky existujú grafické API, najpoužívanejšie v aplikáciách sú *Direct3D* a *OpenGL*. *Direct3D* je súčasťou *DirectX*, čo je kolekcia rozhraní na programovanie multimediálnych aplikácii na platforme *Microsoft Windows*. Rôzne rozhrania preferujú iný vysokoúrovňový jazyk na programovanie shaderov, pre túto prácu bol zvolený jazyk *GLSL* (*OpenGL Shading Language*) ktorý podporuje rozhranie *OpenGL*.

*OpenGL* je viacplatformové grafické rozhranie na programovanie aplikácií. Na programovanie vykresľovacieho reťazca sa používa jazyk *GLSL*. Možnosť zmeniť fixnú funkcionalitu renderovacieho reťezca bola pridaná do rozhrania len s verziou *OpenGL 2.0* v roku 2004. S touto verziou mohli programátori implementovať vlastné algoritmy pre renderovanie, s použitím vysokoúrovňového jazyka *GLSL*.

## 2.1 Vykreslovací reťazec v OpenGL

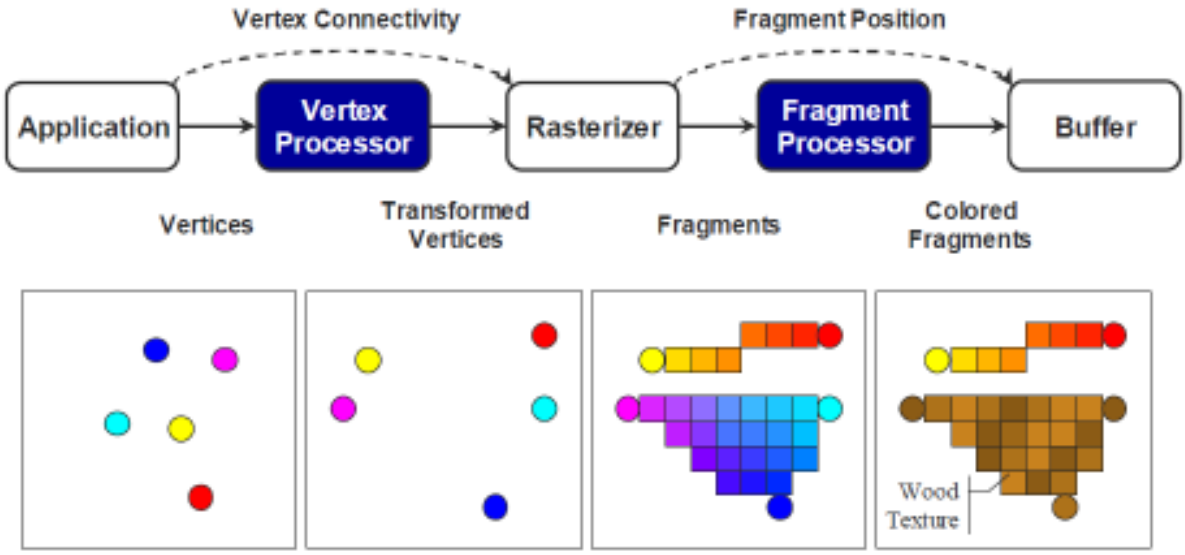
Fixné renderovací reťazec môžeme znázorniť s nasledujúcim obrázkom:




**1-Fixed function pipeline in OpenGL**

Kreslenie v *OpenGL* sa začne načítaním geometrie alebo geometrických primitív z pamäte systému alebo z *video pamäte*. Geometrické primitívy, ktoré *OpenGL* podporuje sú body, čiary, trojuholníky, polygóny a štvoruholníky. Načítané geometrické dáta sa dostanú do vertex procesoru, kde je vlastne začiatok vykresľovania.

Obrázok obsahuje časti reťazce, s ktorými sa v tejto práci zaoberať nebudem. Nasledujúci obrázok zobrazuje programovateľné časti reťazca v *OpenGL*:

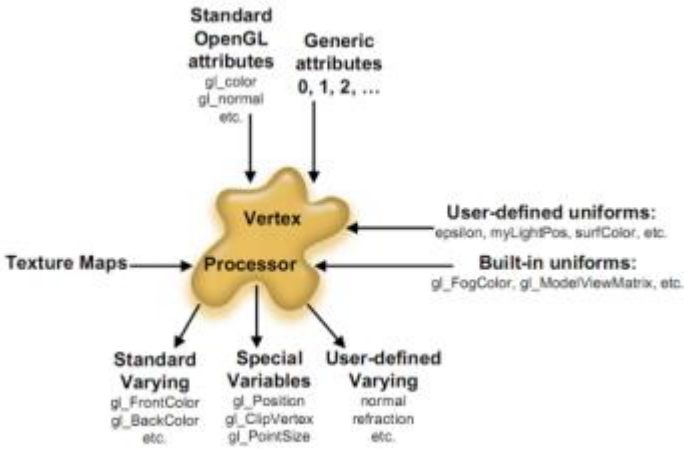



*Vertex* a *fragment procesor* je možné programovať *vertex* a *fragment shaderom.*

## 2.2 Vertex shader

Vertex procesor transformuje z *3D world space* do *2D image space* a osvetľuje vertexy. Keď používame vlastný vertex shader a zmeníme s tým fixnú funkcionalitu vykresľovacieho reťazca, musíme implementovať vyššie uvedené transformácie. Aspoň prvú časť osvetlenia scény je nutné spracovať, lebo systémové premenné, ktoré obsahujú informácie o svetelných zdrojoch nie sú dostupné z fragment shaderu.

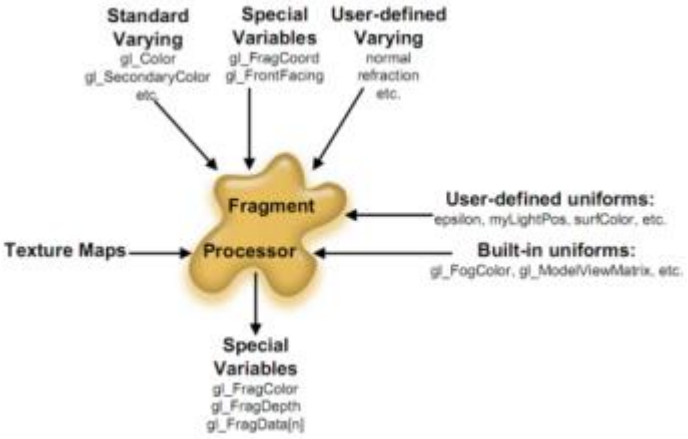
Vertex procesor pracuje prúdovo, a má informácie len o vertexe ktorý práve spracováva. Nemôže vytvoriť nové vertexy, len generuje zo vstupného vertexu výstupný. Pri generovaní môžeme zmeniť pôvodnú pozíciu vertexu podľa potreby. Takto môžeme dosiahnuť napríklad jednoduchý animačný efekt. Výslednú pozíciu musíme priradiť systémovej premennej *gl_Position*, čo je vlastne výstup vertex procesoru.




Osvetľovanie môžeme počítať vo vertex shaderu, ale dostaneme lepší výsledok, keď posunieme výpočet do fragment shaderu. Rozdiel medzi výpočtom per-vertex a per-pixel je celkom značný. Preto je nutné informácie dostať do fragment shaderu. Jazyk *GLSL* umožňuje jednosmernú komunikáciu medzi vertex a fragment shaderom. Z prvého môžeme pomocou špeciálnych premenných poslať dáta priamo do druhého. Na obrázku sú tieto výstupné premenné vyznačené ako „*varying*".

## 2.3 Fragment shader

Po rasterizacii fragment procesor vykoná textúrovanie a vypočíta výslednú hodnotu farby a iné atribúty jednotlivých fragmentov. Hodnoty z výstupu sa zapisujú do *frame bufferu*.




Vo fragment shaderu musíme tiež vypočítať hodnotu farby fragmentov aby renderovanie fungovalo správne. Na tento účel existujú systémové premenné *gl_FragColor*. Shader má prístup k textúram a z „*varying*" premenných k informácii od vertex shaderu. Kombináciou týchto vstupov je možné dosiahnuť veľmi komplexné osvetľovacie alebo textúrovacie modely, napr. *osvetľovací model Phong, bump mapping, normal mapping*, tieňovanie, spekulárne zvýrazňovanie alebo priesvitnosť.

## 2.4 Jazyk GLSL

V čase objavenia programovateľných shaderov ešte neexistovali vysokoúrovňové programovacie jazyky shader programov. Jedinou možnosťou bolo využitie asemblerovského programovacieho jazyka a programovať priamo grafický čip. Tento prístup k shaderom nebol najideálnejší. Riešením stavu by mohol byť vývin nového vysokoúrovňového programovacieho jazyka.

Jazyk *OpenGL Shading Language* bol súčasťou *OpenGL* vo verzii 2.0. *OpenGL* dostal s týmto jazykom veľmi efektívnu utilitu na programovanie shaderov. Základom jazyka je programovací *jazyk ANSI C*. Syntax príkazu a volanie funkcie sú odvodené práve od *C*, čo umožní veľmi rýchle učeniu jazyka. Programátori so skúsenosťou v jazyku *C* alebo *C++* sa budú vedieť *GLSL* veľmi rýchlo naučiť. Veľmi dôležitou časťou jazyka je súbor dátových typov, zabudovaných premenných a funkcií, ktoré podporujú 3D grafické operácie, napr. dátový typ matice a vektory, násobenie vektoru s maticou. [1]

Jazykom *GLSL* možné programovať vertex a fragment procesory, ktoré boli popísané v predchádzajúcej kapitole. Vertex procesor sa programuje s vertex, fragment procesor s fragment shader programom.

### 2.4.1 Základy jazyka GLSL

Pred programovaním shaderov je dobré sa zoznámiť so základnými vlastnosťami jazyka *GLSL*. Jazyk sa používa na programovanie grafického čipu, preto väčšina dátových typov, zabudovaných funkcií a premenných slúži na 3D grafické operácie a výpočty.

**Dátové typy** v *GLSL* môžeme rozdeliť do niekoľkých skupín. Základné skalárne typy sú prevzaté z *jazyka C*, a slúžia ako základ ku komplexnejším dátovým typom. Skalárne typy sú *float*, *int* a *bool*. Tieto typy sú podobné typom s rovnakým názvom z *jazyka C*.

Vektorové typy sú konštruované zo skalárnych typov. Existujú dvoj-, troj- a štvorhodnotové vektorové typy. Typ hodnôt je najčastejšie *float* (*vec4*), ale definícia jazyka *GLSL* umožní aj vytvorenie vektorov z typov *int* (*ivec4*) a *bool* (*bvec4*). Vektory sú veľmi užitočné, ukladajú hodnoty farieb, pozície a textúrové súradnice. *GLSL* nerozlišuje vektory farebné, pozičné a iné vektory. Z hľadiska jazyka je každý vektor obyčajným vektorom skalárnych typov. Jednotlivé hodnoty vektoru sú dostupné dvomi spôsobmi:

- Pracujeme s vektorom ako s jednoduchým poľom z *jazyku C*, hodnoty sú dostupné pomocou indexov: *vektor[0]*.
- Využívame špeciálnu funkciu vektorov, kde je vektor chápaný ako štruktúra a prístup k hodnotám je pomocou názvu komponentu: *vektor.x*. Nasledujúca tabuľka zobrazuje existujúce názvy komponentov rozdelené podľa účelu vektorov:

| | |
|---|---|
| x, y, z, w | Treat a vector as a position or direction |
| r, g, b, a | Treat a vector as a color |
| s, t, p, q | Treat a vector as a texture coordinate |

Pomocou tzv. *swizzling* je možné vybrať aj viac hodnôt z vektora: *vektor.xyz*. Názvy komponentov musia byť rovnakého typu, napr. kombinácia *vektor.xyba* je chybná. *Swizzling* je možné aplikovať na každý výraz s výsledkom vektorového typu.

Matice (*mat2*, *mat3*, *mat4*) v jazyku *GLSL* sú zostavované z reálnych čísiel, iné základné typy nie sú povolené. Môžu mať rôzne veľkosti: 2x2, 3x3 a 4x4, kde čísla reprezentujú počet stĺpcov a riadkov. Prístup k hodnotám matice je možné pomocou indexov. Matica je vlastne „vektor vektorov“, stĺpce sú dostupné prvým indexom, riadky druhým. Napr. *matica[0][1]* je hodnota z prvého stĺpca a z druhého riadku.

Dátový typ sampler slúži k práci s textúrami. Samotný shader napísaný v jazyku *GLSL* textúru načítať nemôže. Môže len prijať textúru od aplikácie vo forme *uniform premennej*. V *GLSL* existuje viac typov textúr: jedno-, dvoj- a trojrozmerná textúra, *cube-map textúra*, jedno- a dvojrozmerná *depth textúra*. Prístup k textúram je pomocou zabudovanej funkcie *texture2D(texture, coord)*, kde *texture* je dvojrozmerná textúra typu *sampler2D*; *coord* je vektor typu *vec2*, ktorý obsahuje textúrovacie súradnice.

Existujú ešte dátové typy štruktúra (*struct*) a pole (*array*), ktoré sú podobné ako isté typy v *jazyku C*. Typ void deklaruje funkciu, ktorá nemá návratovú hodnotu. Napr. funkcia *main* nevráti žiadnu hodnotu a musí byť deklarovaná ako *void*.

**Kvalifikátory** premenných určujú typ premennej (typ premennej neznamená dátový typ). Lokálne premenné môžu mať len kvalifikátor const. Globálne premenné môžeme deklarovať ako const, attribute, uniform a varying. Premenné typu const sú konštanty, ich hodnoty nie je možné zmeniť. Kvalifikátory attribute, uniform slúžia k prenosu dát do vertex a fragment shaderov z *OpenGL* a z aplikácie.

Premenné typu attribute prenášajú data z *OpenGL*. Sú aktualizované pri každom spustení vertex shaderu, t.j. per-vertex. Sú deklarovateľné len vo vertex shadere a sú iba na čítanie. Tento kvalifikátor môžu mať iba premenné typu *float*, alebo vektory a matice zostavené z reálnych čísel. Časť zabudovaných premenných v *GLSL* majú attribute kvalifikátor.

Kvalifikátor uniform dostanú premenné, ktoré prenášajú dáta z aplikácie do vertex alebo fragment shaderu. Ich hodnota je iba na čítanie. Hodnoty týchto premenných sú v jednom snímku konštantné. Dátový typ *uniform premennej* je ľubovoľný, textúry sa do shaderu prenášajú týmto typom premenných.

Premenné typu varying predstavujú most medzi vertex a fragment shaderom. Len s *varying premennými* je možné poslať dáta z vertex do fragment shaderu. Premenné je nutné deklarovať v obidvoch shaderoch, dátový typ a názov premennej musí súhlasiť. Do premennej môže zapisovať len vertex shader, vo fragment shaderu je premenná iba na čítanie.

**Zabudované premenné** v *GLSL* poskytujú transformačné matice, dôležité per-vertex premenné a informáciu o *OpenGL* stavoch. Delia sa na rôzne skupiny:

Špeciálne premenné vo vertex shaderu: *gl_Position*, *gl_PointSieze*, *gl_ClipVertex*; tieto premenné sú dostupné len vo vertex shaderu, pred zápisom do nich, ich hodnota je nedefinovaná. Vertex shader musí do premennej *gl_Position* zapísať výslednú vertex pozíciu, inak kompilátor bude hlásiť chybu. Ostané dve premenné špecifikujú funkciu vykresľovacieho reťazca medzi vertex a fragment shaderom. Zápis do nich nie je povinný.

Špeciálne premenné vo fragment shaderu: vo fragment shaderu sú tri premenné, do ktorých môžeme zapisovať výslednú hodnotu: *gl_FragColor*, *gl_FragData*, and *gl_FragDepth*. Premenná *gl_FragColor* predstavuje výstup z fragment shadera, zapisuje sa do nej výsledná hodnota farby fragmentu. Keď sa do premennej žiadna hodnota nezapíše, farba pixelu bude nedefinovaná. Ostatné premenné slúžia k ďalšej špecifikácie výstupu.

Zabudované vertex atribúty: tieto atribúty obsahujú per-vertex dáta, sú aktualizované pri každom spustení vertex shadera: *gl_Color*, *gl_SecondaryColor* – farby z *OpenGL*, *gl_Normal* - normála, *gl_Vertex* - pozícia vertexu, *gl_MultiTexCoord0* - textúrové súradnice, od 0 do 7, *gl_FogCoord* - hmla

Zabudované konštanty: minimálne hodnoty premenných, ktoré sú závislé na implementácie *OpenGL*.

Zabudované uniform premenné: tieto *uniform premenné* poskytujú prístup ku stavom *OpenGL*.

Zabudované varying premenné: slúžia k prenosu informácií medzi vertex a fragment shaderom. Napr. *gl_TexCoord[]* slúži k prenosu textúrovacých súradníc z vertex do fragment shaderu.

**Zabudované funkcie:** jazyk *GLSL* definuje rad zabudovaných funkcií pre skalárne a vektorové operácie. Medzi funkciami sú napr. matematické, geometrické, textúrovacie, vektorové, maticové a všeobecné funkcie.

Jazyk *GLSL* poskytuje veľa zabudovaných premenných a funkcií, tým uľahčí prácu programátora, lebo netreba definovať už existujúce veci.

# 3 Základy OpenSceneGraph

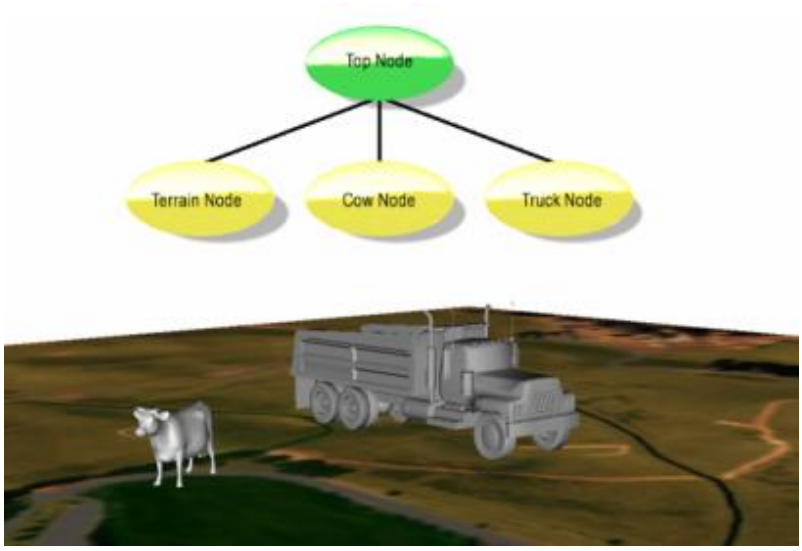
## 3.1 História OpenSceneGraphu

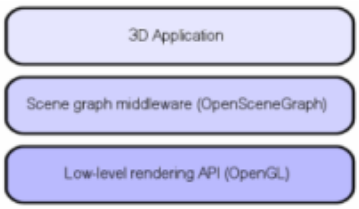
V roku 1997 pracovník spoločnosti Silicon Graphic, Inc (ďalej len SGI), **Don Burns** vyvinul vo svojom voľnom čase vetroňový simulátor, ktorý implementoval so scene graph *Performer*om od SGI. Aby jeho simulátor bol dostupný na Linux platforme (na Linux ešte scene graphy neboli), začal vývojom scene graphu *SG*. **Robert Osfield** sa zapojil do vývoja a radil kolegovi, aby *SG* bol *open source* projektom. Po **Don**ovom odchode z projektu, **Robert** nazval projekt ***OpenSceneGraph*** (ďalej len OSG). V roku 2001 sa na *SIGGRAPH*e objavili prví zákazníci, ktorí potrebovali open source scene graph knižnicu pre vývoj vlastných aplikácií. Vývojový tým sa rozrástol a časom implementovali všetky dnes dostupné knižnice. V súčasnosti mnoho aplikácie využívajú OSG na renderovanie 2D a 3D scén v oblasti grafických informačných systémov, hier, animácie alebo modelovania. [2]

## 3.2 Úvod k scene graphu

*Scene graph* je hierarchická stromová dátová štruktúra, ktorá organizuje jej obsah pre výkonnejšie renderovanie.

Na vrchole stromu *scene graph* je koreňový uzol (*root node*). Pod ním sú skupinové uzly (*group nodes*), ktoré organizujú geometriu a vykresľovacích stavov kontrolujúcich vzhľad scény. Koreňové a skupinové uzly môžu mať nula alebo viac synovských uzlov. Listy stromu sú geometrické modely, ktoré sa objavujú v scénach.






*Scene graphy* ponúkajú rôzne typy uzlov s odlišnou funkcionalitou, napr. *LOD* (*level of detail*) uzly, transformačné uzly, ktoré zmenia polohu geometrie v scéne, *switch uzly*, ktoré blokujú alebo odblokujú synovské uzly. Objektovo orientované *scene graphy* poskytujú túto variabilitu s vlastnosťou *dedičstva*. Uzly majú spoločný podklad zdedený od otcovských uzlov, a definujú vlastnú špeciálnu funkcionalitu. Vysoký počet rôznych uzlov a ich priestorová organizácia poskytuje také spôsoby ukladania dát, ktoré nízkoúrovňové renderovacie rozhrania nemajú. *OpenGL* a *Drect3D* sa zamerajú primárne na možnosti, ktoré sa dajú nájsť v grafickom hardware. Hoci hardware umožňuje uloženie geometrie a stavu, nízkoúrovňové rozhrania majú minimálne možnosti na priestorové uloženie dát, nevhodné pre väčšine 3D aplikácie. *Scene graphy* sa umiestnia medzi nízkoúrovňovým rozhraním a aplikáciami.

Scene graphy poskytujú napr. nasledujúce extra vlastnosti:

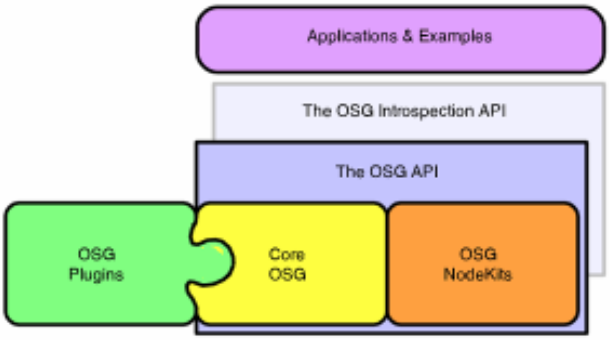
- **Priestorová organizácia:** *scene graph* stromová štruktúra.
- **Odstrihnutie**: odstraňovaním neviditeľných časti scény sa zrýchľuje vykresľovanie.
- **LOD** (Level Of Detail): úroveň podrobnosti. Výpočet vzdialenosti medzi objektom a kamerou umožní efektívne renderovanie. Časť scény sa načíta z disku do pamäti, keď vzdialenosť od kamery je menšia než nejaká predurčená hodnota. Keď je vzdialenosť väčšia, objekt sa z pamäte odstráni.
- **Minimalizácia zmeny stavu**: pre maximálny výkon je dôležité sa vyhnúť zbytočnej zmene stavu. *Scene graphy* triedia geometrie podľa stavu pre minimalizáciu zmeny.
- **Vstup/výstup súboru**: *scene graphy* sú efektívne ohľadne vstupu a výstupu 3D dát z disku. Vnútorné dátové štruktúry umožnia aplikáciu, aby po načítaní manipuloval efektívne dynamické 3D dáta. *Scene graphy* môžu byť preto dobrým konvertorom medzi formátmi súboru.
- Knižnice *scene graphu* poskytujú ešte mnoho **funkcií**, ktoré nízkoúrovňové rozhrania nemajú, napr. podpora pre renderovanie textu, grafické efekty (*particle effect*, tieňovanie), optimalizácie pri renderovaní.

## 3.3 Prehľad OpenSceneGraphu

OSG je súbor *open source* knižníc pre manažment scén a optimalizáciu procesu renderovania v aplikáciách. Je implementovaný v *ANSI C++* a využíva štandardné nízkoúrovňové grafické *API OpenGL*. Preto OSG je viacplatformový a beží na operačných systémoch *MS Windows*, Mac *OS X* a na *UNIX* a *Linux*. OSG je *open source* a je k dispozícii pod modifikovanou licenciou *Lesser General Public License*, alebo pod licenciou *Library GPL* (*LGPL*).

OSG pozostáva zo súborov knižníc, ktoré môžeme rozdeliť do piatich skupín:

- **Jadro OSG**: poskytuje základný scene *graph* model a vykresľovacie metódy, okrem toho ešte ďalšie funkcie, ktoré 3D aplikácie väčšinou potrebujú.
- **NodeKity**: rozširuje jadro OSG s vyššími typmi uzla a špeciálnymi efektmi.
- **Zásuvné moduly**: knižnice, ktoré načítajú a zapisujú na disk 2D obrázky a 3D modely.
- **Knižnice podporujúci integráciu** OSG napr. do skriptovacích jazykov.
- Súbor **aplikácii** a **príkladov** na demonštráciu využitie knižnice OSG.




**Jadro OSG** obsahuje štyri knižnice: *osg, osgDB, osgUtil, osgViewer*.

Knižnica *osg* je srdcom OpenSceneGraphu. Obsahuje *scene graph* uzlové triedy na modelovanie scény. Scéna vlastne pozostáva z týchto uzlov. Obsahuje ešte triedy pre vektory, matice a geometriu modelov, implementuje vykresľovacie stavy a ich manažment.

Triedy *scene graph* uzlov sú odvodené z triedy *osg::Node*. Používajú sa na vybudovanie *scene graphu*. Okrem základu zdedeného od otcovskej triedy, každý uzol (*group* a *listové uzly*) má vlastnú funkciu. Uzol root je vlastne *Node*, ktorý nemá rodiča. Trieda *Node* obsahuje metódy ktoré pomáhajú vykresľovaniu, *callback* metódy a manažment stavu. Ďalšou dôležitou triedou je *Group*, ktorá je základom všetkých uzlov, ktoré môžu mať potomka. Je veľmi dôležité ohľadne priestorovej organizácie *scene graphu*. Listovými uzlami grafu sú uzly typu *Geode (Geometry Node)*. Nemajú potomka, ale môžu obsahovať objekty *osg::Drawable* obsahujúci geometrický model na vykreslenie. Použitím uzla *LOD* je dosiahnuté, aby v scéne boli objekty s variabilnou úrovňou podrobnosti. Uzol *MatrixTransform* transformuje geometriu svojich potomkov podľa matice, ktorú obsahuje. Hodnoty matice rozhodujú o tom, či potomky budú otočení, posunutí, skosení alebo budú zmenené ich pomery. Pomocou uzla *Switch* môžeme zapnúť alebo vypnúť potomka uzla. Okrem vyššie uvedených existujú aj iné typy uzla.

Listové uzly *scene grafu* obsahujú geometrické dáta na vykresľovanie. K uzlu *Geode* sú pripojené objekty triedy *Drawable* ktoré sú virtuálne, geometrické dáta obsahujú odvodené triedy *Geometry* a *ShapeDrawable* (obsahuje preddefinované modely ako guľa, kocka, kužeľ). Trieda *PrimitiveSet* spolu s *Geometry* predstavujú funkciu *vertex array* z *OpenGL*. *Geometry* obsahuje pole vertexov, súradnice textúry, pole farieb a normál. *PrimitiveSet* poskytuje podporu pre vykresľovacie príkazy z *OpenGL*, špecifikuje aké primitívy budú vykresľované. Dôležité sú ešte vektorové triedy (*Vec2, Vec3, Vec4*) a polia vektorov (napr. *Vec2array, Vec3array*).

OSG umožní ukladať požadované *OpenGL* stavy do *scene grafu*. Uzly sú triedené podľa ich stavov, aby počet zmien stavov bol minimálny a renderovanie bolo rýchlejšie. Ku každému uzlu môžeme priradiť nejaký stav, ale pri vykresľovaní sú stavy hierarchicky odvodené z rodičovských

uzlov. Pomocou tzv. *Inheritance flags* môžeme ovplyvniť odvodenie stavov. Trieda *StateSet* skladuje súbor stavov jednotlivých uzlov, ich atribúty (*blending* funkcie, farba hmly, informácie o materiálu) a módy (môžu zapnúť alebo vypnúť fixné funkcionality *OpenGL*, ako je osvetľovanie, hmly, *blending*). *StateSet* pripojuje k uzlom textúry a shader programy (vertex a fragment shader).

Knižnica *osg* obsahuje ešte pár užitočných tried a pomocných programov. Časť toho tvoria špeciálny pamäťový manažment OSG, tzv. *reference-counted memory scheme*, čo má za úlohu vyhnúť sa *memory leak*. Bázová trieda všetkých *scene graph* uzlov a mnoho ďalších objektov je trieda *Referenced*. Obsahuje čítač odkazov ukazovaných na objekte pre sledovanie stavu pamäti. K objektom, ktoré sú odvodené od triedy *Referenced* a počet na nich ukazujúcich odkazov je nulový, bude privolaný ich deštruktor. V praxi táto vlastnosť OSG využívame pomocou triedy *ref_ptr<>*.

**Nodekity** OSG rozširujú uzly *Node* a *Drawable*, a objekty pre spracovanie stavov. Nodekity poskytujú podporu pre čítanie a zápis z/do *„.osg“* súborov. Súborový formát *„.osg“* je vlastným formátom OSG, pomocou tohto formátu je možné uložiť vytvorenú scénu.

V OSG verzia 2.0 existovalo šesť nodekitov:

- **osgFX:** ďalšie scene graph uzly pre renderovanie špeciálnych efektov (*anisotropic lighting, bump mapping, cartoon shading*).
- **osgSim:** podporuje *OpenFlight* databázu, ako elevačný model terénu a *DOF* transformačné uzly.
- **osgManipulator:** triedy pre manipuláciu s vybranými objektmi *scene graphu*.
- **osgParticle:** poskytuje renderovacie efekty, ako výbuch, oheň a dym.
- **osgText:** užitočné pri pridávaní textu k *scene graphu*.
- **osgTerrain:** podpora pre renderovanie terénu.
- **osgShadow:** obsahuje implementácie tieňovej techniky.

**Zásuvné moduly OSG** obsahujú podporu pre súborový vstup a výstup. Môžeme načítať napr. 2D obrázky, 3D modely, filmy aj nové typy písma. Existujú moduly podporujúce aj archívy a načítanie súborov z Internetu.

**Knižnice podporujúce integráciu** umožnia používať OSG v programovacích prostredí, ktoré podporujú spojenie s *C++*. Trieda *osgIntrospection* poskytuje podporu pre ďalšie prostredie. Napr. *Java, Tcl, Lua,* alebo *Python*.

OSG obsahuje ešte **príklady** a ďalšie **aplikácie** pre demonštráciu schopnosti rozhraní a podporu, ladenie a vývoj aplikácií.

# 4 Teória vybraných grafických efektov

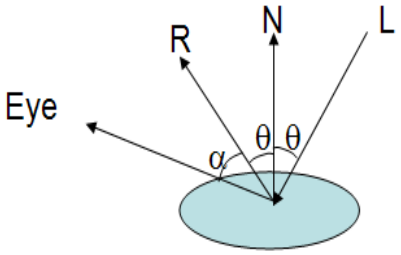
Pre demonštráciu práce s shadermi som vybral tri grafické efekty, ktoré som implementoval. Pri výberu bolo dôležité, aby moje shader programy predstavili funkčnosť vertex, aj fragment shaderu. Preto som zvolil osvetľovací model *anisotropic lighting* v pixel shaderu, *vertex morphing* vo vertex shaderu a *animovanú trávu*, čo potrebuje obidve shadery.

## 4.1 Anisotropic lighting

Jednoduché osvetľovacie modely považujú povrch objektov ako úplne plochý. Existujú ale objekty, napr. CD, ozdobné gule na vianočných stromčekoch, povrch šiat alebo vlasy, ktoré sa chovajú odlišne, keď skúmame ich povrch z rôznych uhlov. To znamená, že svetlo sa odráže iným spôsobom ako z plochých povrchov. Môžeme povedať, že osvetlenie nie je *izotropické*, ale *anizotropické*.

Osvetlenie je výpočet intenzity odrazeného svetla z telies scény. Základom anizotropického osvetlenia je *Phongov model*, ktorý popísuje intenzitu odrazeného svetla ako súčet troch zložiek: *diffuse, specular* a *ambient*.

**Difúzna zložka** je intenzitou svetla odrážajúceho sa z drsných povrchov. Táto zložka sa vypočíta pomocou *Lambertového odlesku*, svetlo sa odráža rovnakou intenzitou do každého smeru. Jas modelu sa otáčaním nezmení. Intenzita svetla je skalárny súčin normalizovanej normály a vektoru svetla. Výsledná hodnota je násobená hodnotou farby povrchu a difúznou intenzitou prichádzajúceho svetla.




Vzor výpočty:

$$I_{diffuse} = k_d(L \cdot N)I_d$$

**Zložka specular** (alebo **odlesk svetla)** je závislá od vektora smeru kamery, vektora odrážaného svetla a od konštanty lesku, čo určuje mieru odlesku. Výsledok je násobený sekulárnou intenzitou prichádzajúceho svetla a odraziacou konštantou povrchu:

$$R = 2N(L \cdot N) - L$$

$$I_{specular} = k_s(R \cdot V)^{\alpha} I_s$$

**Zložka ambient** je konštantná hodnota globálnej intenzity svetla. Jeho hodnota sa rovná súčinu ambientskej intenzity svetla a ambientskej konštanty povrchu:

$$I_{ambient} = k_a I_a$$

Výsledný vzorec pre výpočet *Phongov osvetľovaní* je nasledujúci:

$$I_{Phong} = k_a I_a + \sum_{lights} (k_d(L \cdot N) I_d + k_s(R \cdot V)^{\alpha} I_s)$$

Pre anizotropické osvetlenie sa používa zmenená verzia vyššie popísaného modelu, tzv. *vláknový osvetľovací model* [3]. Podstatou modelu je spôsob odlesku svetla z povrchu akým sú vlasy. Vektory *N* a *R* je nutné zmeniť. Normála *N'* bude normalizovaný vektor vektora, ktorý vznikne projekciou svetelného vektora do normálovej roviny vlákna v bode osvetlenia. Potom vektor *R'* môžeme vypočítať:

$$R' = 2N'(L \cdot N') - L$$

Pomocou matematických výpočtov a vlastnosti tzv. *tangent space* (vektory tangenta *T*, normála *N* a binormála *B* sú ortogonálne) je možné odvodiť nasledujúce vzorce:

$$L \cdot N' = (1 - (L \cdot T)^2)^{\frac{1}{2}}$$

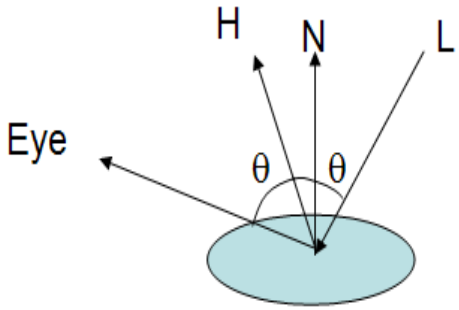
$$V \cdot R' = (1 - (L \cdot T)^2)^{\frac{1}{2}} (1 - (V \cdot T)^2)^{\frac{1}{2}} - (L \cdot T)(V \cdot T)$$

Keď tieto vzorce dosadíme do zmeneného *Phongovho modelu*, zistíme, že výpočet je funkciou dvoch skalárnych súčinov: *L·T* a *V·T*, kde je *T* tangenta. Najväčší problém bude s výpočtom tangenty. V kapitole **Návrh** bude rozobrané, ako túto hodnotu vypočítať.

Vedľa vyššie popísaného modelu anizotropického osvetlenia existuje ešte mnoho spôsobov ako dosiahnuť žiadaný efekt. Jeden z tých spôsobov využíva *osvetľovací model Blinn-Phong*. Model, a tak aj výpočet je jednoduchší, než *Phongov model*. Výsledný efekt bude ale viditeľne horší ako u predchádzajúceho osvetľovania. Základom osvetľovania je, že pri výpočte spekulárnej zložky sa používa tzv. *polovičný vektor* namiesto vektora odlesku. Výpočet *polovičného vektora* je veľmi jednoduchý,

$$H = V - L$$

jazyk *GLSL* má aj zabudovanú systémovú premennú, programátor nemusí nič počítať.




Odvodený vzorec [4] bude nasledujúci:

$$I_{Phong} = k_a I_a + \sum_{lights} (k_d (L \cdot N) I_d + k_s (N \cdot H)^{\alpha'} I_s)$$

## 4.2 Vertex morphing

Na začiatku deväťdesiatych rokov sa objavili filmy s efektmi generovanými počítačom [5]. Konkrétne sa jedná o film *Terminator 2: Judgement Day*. Nové efekty znamenali začiatok fotorealistickej počítačovej grafiky vo filmoch. Najatraktívnejšie efekty boli transformácie robotu *T-1000*, boli vytvorené technikou *morphing*. Proces sa odohráva v 3D world space, jeden model sa transformuje do druhého.

Jedným spôsobom, ako dosiahnuť vyššie uvedený efekt je *vertex tweening*. Kde každý vertex okrem zdrojového, má ešte absolútnu alebo relatívnu cieľovú pozíciu. Pomocou *zmiešajúceho faktora*, ktorý je totožný pre každý vertex, môžeme interpolovať medzi zdrojovou a cieľovou pozíciou. Výpočet aktuálnej pozície s relatívnou cieľovou pozíciou je nasledujúci:

$$Position_{output} = Position_{source} + Position_{destination} \cdot Factor$$

Výpočet s absolútnou cieľovou pozíciou:

$$Position_{output} = Position_{source} + (Position_{destination} - Position_{source}) \cdot Factor$$

Pozície sú 3D vektory (súradnice x, y, z) a zmiešajúci faktor je reálne číslo. Vzhľadom na to, že výpočet s relatívnou cieľovou pozíciou je rýchlejší, ďalej budeme používať len túto metódu. *Vertex tweening* je veľmi obmedzená technika. Prvý a druhý model sú úplne totožné so zmenenou geometriou. Nezmenia sa textúry, počet vertexov, materiály a stavy modelu. Preto sa používa táto technika len pri malých animáciách.

Pred vlastnou interpoláciou je nutné vypočítať cieľovú pozíciu ku každému vertexu. To je možné premietaním prvého modelu do druhého a obrátene. Definujeme nejaký bod v priestore, z ktorého vypustíme lúče cez vertexy. Hľadáme priesečníky lúča s plochou druhého modelu, vektor najbližšieho priesečníka bude dobrou voľbou vektora cieľovej pozície.

$$Direction = Center - Position$$

Aby efekt nezaťažoval procesor a dostali sme najlepší výkon, *morphing* môžeme implementovať vo vertex shaderu. Dostaneme lepší výsledok, keď transformujeme a renderujeme obidva modely naraz. Aktuálny interpolačný faktor bude konštantou v shadere. Pre jeden model je nutné invertovať faktor, aby obidva boli vždy v rovnakom stave. Teraz je možné interpolovať zo zdrojovej do cieľovej pozície. Ostatné per-vertex funkcie, ako osvetľovanie a výpočet textúrovacích súradníc môžu byť vykonané normálne.

Animácia určite nebude vypadať tak, ako sme si to predstavili. Sú ale určité optimalizácie, ktorými *vertex morphing* môže vypadať lepšie:

- Krokové objekty je možné pridať medzi zdrojovým a cieľovým modelom. Morphing bude vypadať hladšie, závisí len na tom, že koľko medzikrokov bude pridaných.
- Vedľa interpolácie pozícií je možné pracovať aj s inými atribútmi vertexov, napr. normálami, tangent vektormi. Pri osvetľovaní scény sú tieto vedľajšie interpolácie veľmi dôležité.
- Výpočet vektoru cieľovej pozície je celkom náročný. Najlepšou voľbou je vypočítať pozície vopred a ukladať do súboru. Pred renderovaním môže tento krok ušetriť čas.

## 4.3 Animovaná tráva

V minulosti v počítačových hrách bolo pozadie vždy pasívne. Technické základy neboli také, aby mohli programátori efektívne animovať vodu, oblohu, trávu alebo stromy tak, aby animáciami nezaťažovali procesor. Pomocou programovateľných shaderov je ale možné vytvoriť animáciu pozadia, ktorá "beží" na grafickom čipe, a získať tak reálnejšiu scénu a zároveň nezaťažovať procesor.

Animácia prírody je veľmi ťažká, pretože listy stromu a steblá trávy sa pohybujú nevypočítateľne a preto pre simuláciu sú potrebné dlhé matematické vzorce. Pri animácii trávy je dobré uvedomiť si, že vietor hýbe väčšou oblasťou trávy do podobného smeru. Naproti tomu rýchlejšie turbulencie spôsobujú menšie, ale náhodnejšie pohyby.

Pohybuje sa len horná časť stebla trávy, dolná časť je „pripevnená" k zemi. Preto, keď simulujeme steblo trávy zjednodušene - ako vektor so začiatočným bodom na zemi, bude sa pohybovať len koncový bod. Animácia trávy má isté obmedzenia, lebo dĺžka stebla je konštantná a preto bude koncový bod vektoru posunutý tak, aby sa dĺžka vektoru nezmenila. Pohyb stebla trávy môžeme popísať ako periodický, lebo po posunutí jeho konca sa steblo vždy vráti do bezveterného stavu. Tieto dve myšlienky vedú k tomu, že pohyb hornej časti stebla trávy je možné popísať periodickou funkciou sínus.

Aby bolo možné zmeniť frekvenciu a mieru animácie, na jednotlivé steblá sa aplikuje *sínusová vlna*:

$$y(t) = A \cdot \sin(\omega t + \theta)$$

kde *A* je *amplitúda*, ω je *uhlová rýchlosť*, φ je *počiatočná fáza*. Amplitúda umožní zmeniť veľkosť vyhýbania, zmenou uhlovej rýchlosti je možné zmeniť frekvenciu pohybu.

S jednou sínusovou vlnou sa steblo trávy pohybovalo veľmi vypočítateľne a nereálne. Preto je lepší nápad kombinovať viac sínusových vĺn, aby animácia vypadala reálnejšie. Každá vlna môže mať inú amplitúdu a kruhovú rýchlosť a tým sa ešte viac zlepší reálny pohyb trávy [7].

Generovanie sínusových vĺn vo vertex shaderu je dosiahnuteľné pomocou *Taylorovej rady*. *Taylorovým rozvojom* získame *Taylorov polynóm*, ktorý už je vypočítateľný v shadere.

Vzorec na výpočet *Taylorovho polynómu* je:

$$T_n = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k$$

kde *f'* je prvá derivácia funkcie *f*.

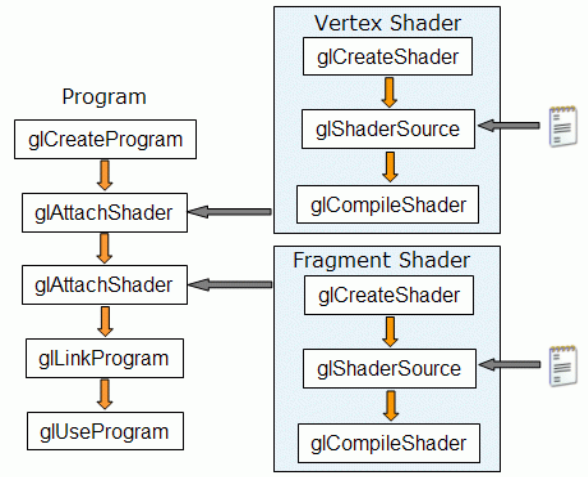
# 5 Návrh

## 5.1 Shadery v OpenSceneGraph

Po objavení sa prvých grafických kariet s programovateľnými shadermi do *OpenGL* s verziou 2.0 dostal programovací jazyk *GLSL* pre podporu nových čipov. Jazyk bol priamo integrovaný do aplikačného rozhrania. *GLSL* bol vyvinutý spoločne s *OpenGL*, takže aplikačné rozhranie podporuje každú funkciu a všetky vlastnosti jazyka. Aby bolo možné používať shadery v OpenSceneGraph, toto grafické rozhranie prevzalo a implementovalo vyššie popísanú podporu pre jazyk *GLSL*.

Shadery v OSG fungujú podobným spôsobom, ako v *OpenGL*, rozdielom je, že shader programy sú linkované k uzlom scény. Sú pripojiteľné ku každému uzlu. Pripojovaním programov k uzlom typu *Group* je možné aplikovať grafický efekt aj na viac listových alebo skupinových uzlov.

Hlavné kroky linkovania shaderov v *OpenGL* sú zobrazené na nasledujúcom obrázku:




Pre používanie shaderov sú potrebné dva objekty, *glShader* ktorý načíta, kompiluje a obsahuje shader, a *glProgram* ktorý pripojuje *glShader* k aplikácii.

Najprv sa vytvorí shader objekt pomocou funkcie *glCreateShader(shaderType)*, parametrom funkcie je typ shaderu (vertex alebo fragment). K jednému *program objektu* je možné pripojiť viac shaderov istého typu, ale zdrojové kódy musia obsahovať len jednu funkciu *main*. Takto je možné dosiahnuť modulárnosť pri programovaní shaderov. Potom sa do objektu pridá reťazec zdrojového kódu shadera s funkciou *glShaderSource()*. Zdrojový kód je možné načítať zo súboru. Aby *shader objekt* bol použiteľný, je nutné kompilovať kód. Túto funkciu plní *glCompileShader()*. Kompilácia je možná aj po pripojení objektov k *program objektu*.

Objekt *glProgram* slúži na pripojenie shaderov k *OpenGL*. Objekt sa vytvorí s funkciou *glCreateProgram()*. V aplikáciách sa môžu vyskytovať rôzne vytvorené *program objekty*, pri vykresľovaní je možnosť prepnutia medzi nimi. K vytvorenému objektu je možné pripojiť *shader objekty*, k tomu slúži funkcia *glAttachShader()*. Pri linkovaní *program objektu* k aplikácii s funkciou

*glLinkProgram()*, shadery musia byť kompilované. „Hotový“ program môžeme spustiť takto: *glUseProgram(prog)*, kde parameter prog je ukazovateľom na *program objekt*.

Pripojiť shader k aplikácii v *OpenGL* požaduje veľký počet funkcií. Jednotlivé kroky sú pochopiteľné, ale je ich veľa. Linkovanie v OSG je trošku jednoduchšie.

*OpenGL API* je úplne mapovaná do OSG nasledujúcim spôsobom:

$$\begin{aligned} glShader \quad object &\Longrightarrow osg :: Shader \\ glProgram \quad object &\Longrightarrow osg :: Program \\ glUniform^{*}() &\Longrightarrow osg :: Uniform \end{aligned}$$

Trieda *osg::Shader* je odvodená od *osg::Object*, a nahradzuje *glShader* objekt z *OpenGL*. Objekt tejto triedy slúži k manažmentu shaderov. Obsahuje metódy načítania zdrojových kódov shaderu zo súboru, alebo z reťazce. Podobne ako v *OpenGL* aj tento objekt má dva typy, *VERTEX* a *FRAGMENT*. Zdrojový kód nemusíme kompilovať, objekt po načítaní kódu vykoná tento krok automaticky. K jednému *program objektu* je možné pripojiť viac *shader objektov* analogicky ako v *OpenGL*.

Objekty z triedy *osg::Program* sú v OSG náhradou objektov *glProgram*. Prázdny objekt, tj. bez pridaných shader objektov, použije pri renderovaní fixný vykresľovací reťazec. K *program objektu* je možné pripojiť viac shaderov daného typu, rovnako ako u *glProgram*. Linkovanie a spustenie objekt vykoná automaticky, netreba zvlášť volať linkovačné a aktivačné funkcie. *Program objekty* zasielajú ešte *vertex atribúty* do shaderu, ale o tom trošku neskoršie.

*Program objekty* sa pripojujú k jednotlivým uzlom scény a pri vykresľovaní volajú *shader programy*. Shadery sú aplikované na uzly na ktoré sú pripojené *program objekty*, a na ich synovské uzly. Pripojenie program objektov k uzlom je veľmi jednoduché, lebo trieda *osg::Program* je odvodená od *osg::StateAttribute*. To znamená, že program objekty sú atribútom stavu uzlov. Ako atribút stavu je možné tieto objekty pridať k *StateSet* stavovým objektom, ktoré môžeme pripojiť ku každému uzlu scény. Preto shadery sú pripojiteľné ku každému bodu scény a pomocou vhodného nastavenia dedenia stavu je výsledná scéna veľmi variabilná.

Pri porovnaní metódy pripojovania shaderov v *OpenGL* a v OpenSceneGraph, metóda pripojovania v OSG je podľa mňa jednoduchšia a lepšia. Nie je totiž nutné zvlášť preložiť zdrojové kódy shaderu, nemusíme linkovať a spustiť program, všetko sa odohráva automaticky po načítaní kódu a pripojení programu k scéne. OSG je čisto objektovo orientovaný, preto funkcie alebo metódy pre spracovanie a pripojenie shaderov sú implementované vo vnútri objektov. *OpenGL* API je už trocha zastaraná, OSG zvládne pripojenie shaderov pohodlnejšie a niektoré nutné kroky vykoná automaticky.

### 5.1.1 Vstup do shaderu z OpenSceneGraph

Vertex a fragment shader programy sa pripojujú ku scéne podľa vyššie popísanej metódy. Okrem pripojení sú veľmi dôležité užívateľské vstupy do shaderu. Bez užívateľských vstupov, s použitím len zabudovaných premenných z *OpenGL* a z OSG nebolo možné vytvoriť tak širokú škálu efektov, ktoré poznáme dnes.

Existujú štyri spôsoby ako do shaderu poslať informácie. Prvým spôsobom je možnosť využitia jedného zo systémových stavov *OpenGL*. Napr. informácie o farbe svetla sú prenášané do shaderu pomocou systémových premenných. Druhým spôsobom je využitie pripojiteľných atribútov k vertexom, čiže *vertex atribúty*. Tretí spôsob je pomocou *uniform premenných*. Tento spôsob umožní poslať do shaderu ľubovoľné informácie. Pochopiteľne, do shaderu sa dajú poslať len také dáta, ktoré jazyk *GLSL* podporuje. Napr. reťazec v shaderu je nezmyselný. Štvrtý spôsob je použitie textúry pre poslanie dát. Pod pojmom textúra sa najčastejšie rozumie obrázok, ale je ho možné chápať aj ako pole dát.

**Systémové stavy** v *OpenGL* sú dostupné v *GLSL* ako zabudované systémové premenné, ktoré netreba deklarovať. Do tejto skupiny patria transformačné matice, informácie o svetelnom zdroji a materiálu povrchu. Takéto systémové stavy existujú aj v OSG. Tieto ale musia byť deklarované, a systém obnovuje ich hodnoty v každom snímku. Premenné sú automaticky dostupné v *GLSL*.

**Vertex atribúty** predstavujú informáciu, čo smeruje do *vertex shaderu*. Tieto informácie vo *fragment shadere* nie sú dostupné. Hodnoty atribút sú len na čítanie, zápis nie je povolený. Hodnota vertex atribútu môže byť pre každý vertex iná. Najpoužívanejšie *vertex atribúty* z *OpenGL* sú *gl_Vertex, gl_Normal, gl_Color, gl_SecondaryColor* a súradnice textúry *gl_MultiTexCoord0, gl_MultiTexCoord1*. Okrem týchto zabudovaných premenných programátor môže definovať aj vlastné vertex atribúty. To je možné aj z *OpenGL*, aj z OSG.

Najprv je nutné deklarovať vo *vertex shader*u premennú, do ktorej sa dáta budú posielať. Potom sa funkciou *glGetAttribLocation(program, premenna)* zistí umiestnenie premennej v pamäti. *Program* je ukazovateľ na *program objekt*, *premenná* je názov premennej vo *vertex shaderu*. Funkcia vráti umiestnenie v pamäti, čo je potrebné vedieť pri nastavení hodnoty premennej. Funkcia *glVertexAttrib1f(location, float)* priradí premennej (jej umiestnenie v pamäti je *location*) hodnotu *float*.

Ako vertex atribút je možné poslať do vertex shaderu aj pole vertexov. Túto funkciu je najprv nutné aktivovať ďalšou funkciou *glEnableVertexAttribArray(location)*, kde *location* je zase umiestnenie v pamäti. Funkciou *glVertexAttribPointer()* priradíme k premennej pole vertexu.

V OSG sa tento proces odohráva podobným spôsobom. *Vertex atribúty* je možné pripojiť k objektu *Geometry*, čo obsahuje model v jednom poli vertexov. Spôsob, akým chceme k vertexu priradiť atribúty, si môžeme zvoliť: buď priradíme ku každému polygónu alebo vertexu jeden atribút, alebo celá geometria „dostane“ jeden atribút. Metóda objektu *Geometry setVertexAttribArray (index,*

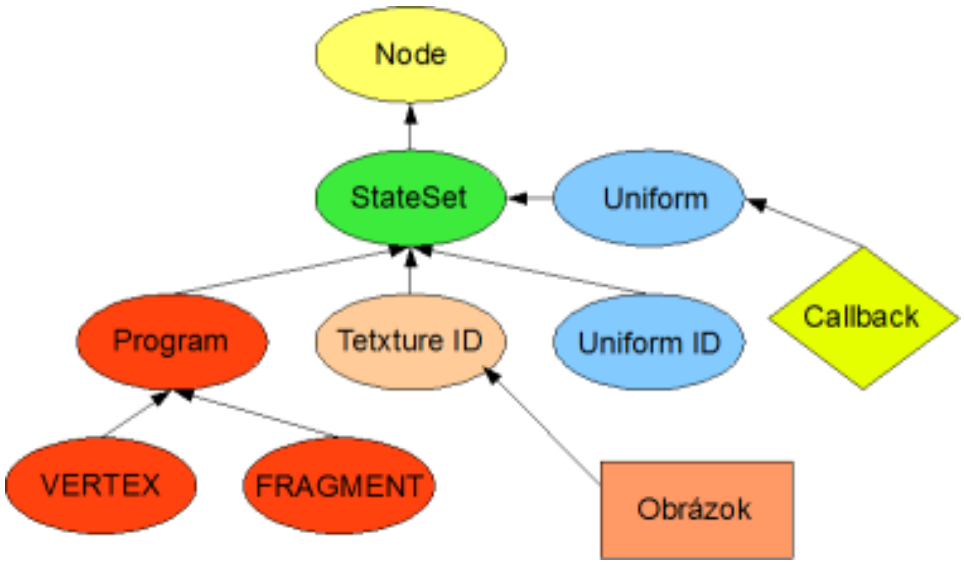
*pole)* pripojuje pole vertexov ku geometrii s indexom *index*. Nasledujúcim krokom je poslať dáta do shaderu volaním metódy *program objektu*, ktorá je linkovaná k uzlu *Geometry*. Táto volaná metóda je *addBindAttribLocation (name, index)*, kde *name* je názov premennej vo *vertex shadere*, a *index* bol definovaný v predchádzajúcom kroku.

**Premenné typu uniform** slúžia na prenos informácií, ktoré sú konštantné v danom snímku alebo v časovom intervale. Hodnota je dostupná vo vertex aj vo fragment shaderu iba na čítanie. V *OpenGL* sa pripojujú premenné rovnakým spôsobom ako *vertex atribúty*, rozdiel je v tom, že v názve funkcie miesto *VertexAttrib* je *Uniform*.

V OSG je pripojenie *Uniform* premenných veľmi jednoduché vzhľadom k *OpenGL* a vertex atribútom. Prvým krokom deklarujeme premennú a priradíme k nej hodnotu. Druhým krokom je vytvorenie *Uniform* objektu s dvomi parametrami. Prvý bude názov premennej v shaderu, a druhý bude vyššie deklarovaná premenná. Tretím krokom je pripojenie *Uniform* objektu cez *StateSet* k uzlu.

Hodnotu *uniform premennej* je možné aktualizovať z OSG. Na tieto účely slúžia objekty typu *Callback*. Takýto objekt sa môže pripojiť aj k objektu typu *Uniform*. Pri vytvorení *Callback objektu* je špecifikovaná aktualizačná funkcia.

**Textúry** z OSG sa predávajú do shaderu pomocou *uniform objektov*. Zo súboru sa načítajú obrázky do *image objektu*, a potom sa tento objekt pripojuje k objektu typu *Texture*D*, kde * značí počet rozmerov textúry. Textúrový objekt sa pridá k *StateSet* objektu s pomocou *texunit index*u. *Textunit index* sa používa pri vytvorení *Uniform* objektu, čo pripojuje textúru k shaderu.




Na obrázku sú zobrazené pripojenia shaderov, textúry, a *uniform* objektov.

## 5.2 Anisotropic lighting

V kapitole **Teória** sú podrobne popísané osvetľovacie modely *Phong* a *Blinn-Phong*. Pri návrhu anizotropického osvetlenia budeme používať vzorce tam odvodené.

Zmenený *Phongov osvetľovací model*, je

$$I_{Phong} = k_a I_a + \sum_{lights} (k_d (L \cdot N) I_d + k_s (R \cdot V)^{\alpha} I_s)$$

,

kde

$$L \cdot N' = (1 - (L \cdot T)^2)^{\frac{1}{2}}$$

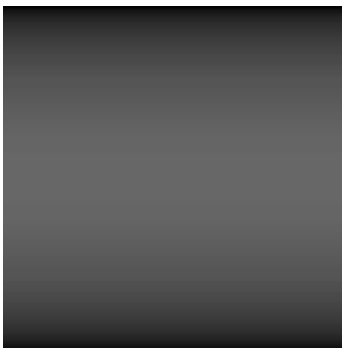
$$V \cdot R' = (1 - (L \cdot T)^2)^{\frac{1}{2}} (1 - (V \cdot T)^2)^{\frac{1}{2}} - (L \cdot T)(V \cdot T)$$

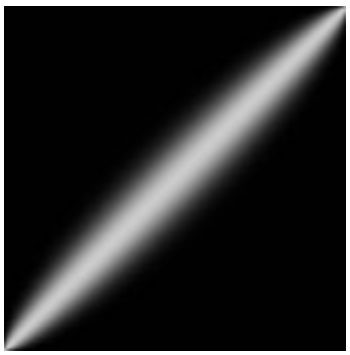
.

Výpočet osvetlenia vypadá náročne, a je k nemu nutné spočítať ešte aj hodnotu tangenty. Tangenta je špeciálny vektor, ktorý sa nachádza v tzv. *tangent space*. Tento priestor je na povrchu telies, tvoria ho tri vektory, *normála, tangenta* a *binormála*. Tieto tri vektory sú ortogonálne. Normála je totožná s normálou z modelového priestoru. *Tangenta* a *binormála* sa nachádzajú na styčnej ploche k povrchu modelu.

Efektívnosť výpočtov v počítačovej grafike je možné zvýšiť predbežným spracovaním. Po preskúmaní vzorcov je zrejmé, že sú funkciou dvoch skalárnych súčinov: $L \cdot T$ a $V \cdot T$. Tento fakt umožňuje predbežné spracovanie a ukladanie hodnôt $L \cdot N'$ a $V \cdot R'$ do 2D textúrovej vyhľadávacej tabuľky. K výpočtu *Phongového osvetlenia* sú nutné len dva skalárne súčiny, ktoré udávajú textúrovacie súradnice k textúre predbežne spracovanej. Týmto spôsobom je možné nahradiť dlhé matematické výpočty dvomi skalárnymi súčinmi a jedným prístupom k textúre.

Správne zostaviť predbežne spracovanú textúru je veľmi dôležité, pretože úspešnosť osvetlenia závisí skoro jedine na tomto procese. Difúznu zložku osvetlenia ($L \cdot N'$) je možné mapovať do farebných zložiek, spekulárnu zložku do alfa kanálu textúry.





Na prvom obrázku je zobrazená difúzna zložka vo farebných kanáloch, na druhom spekulárna zložka v alfa kanále.

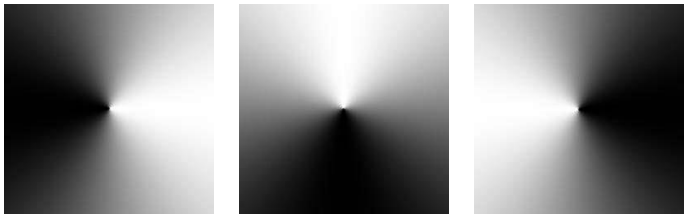
Skalárne súčiny $L \cdot T$ a $V \cdot T$ je možné vypočítať vo vertex alebo vo fragment shadere. Per-fragment osvetlenie je vždy hladšie než osvetlenie per-vertex, lebo vertex shader vypočíta hodnotu

osvetlenia len v každom vertexe, na rozdiel od fragment shaderu, ktorý túto hodnotu počíta v každom fragmente.

Výpočet osvetlenia vo fragment shadere sa koná v *tangent space*, kam sa vektory *V* a *L* musia transformovať z modelového priestoru. Táto transformácia sa vykoná ešte vo vertex shadere. Pri tejto transformácii sú vektory násobené maticou, ktorá je vytvorená z vektorov *tangent space*:

$$ {}_{e}M_{tangentspace} = [T, B, N] = \begin{bmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{bmatrix} $$

Pred násobením touto maticou je nutné vypočítať vektory *T* a *B*. Vektor *T* je možné vypočítať predbežne a uložiť do textúry, podobne ako sa počíta celé osvetlenie. Tri farebné zložky obsahujú tri súradnice vektoru. Prvý obrázok je červená zložka a súradnica x, druhý obrázok je zelená zložka a súradnica y, tretí obrázok je modrá zložka a súradnica z.



Z textúr získame tangentu, a vektor B (binormálu) vypočítame jednoducho: z dôvodu ortogonálnosti troch vektorov vektorovým súčinom normály a tangenty. Po skladaní *TBN matice* transformujeme vektory *L* a *V*. Tým je proces transformácie vo vertex shaderu hotový.

Po dokončenej transformácii, už vo fragment shaderu sa vypočítajú skalárne súčiny $L \cdot T$ a $V \cdot T$. Výsledkom súčinov budú dve reálne čísla *u* a *v* z intervalu <-1,1>. Textúrové súradnice sú ale z intervalu <0,1>. Preto k hodnotám *u* a *v* pripočítame jednu a vydelíme dvoma. Konečné osvetlenie dostaneme súčtom textúr difúznej a spekulárnej zložky (so súradnicami *u* a *v*).

Osvetlenie je možné vypočítať aj vo vertex shaderu [8]. K tomto výpočtu budeme používať *Blinn-Phong model*. Tento model je jednoduchší ako Phongov model, pretože nemusíme zistiť hodnotu tangenty.

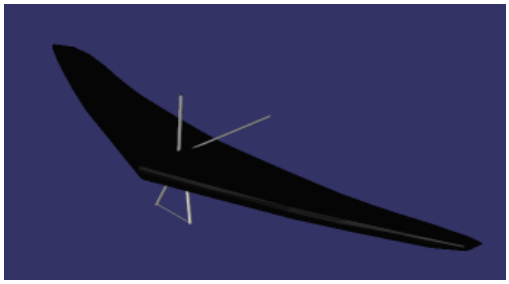
$$ I_{Phong} = k_a I_a + \sum_{lights} (k_d (L \cdot N) I_d + k_s (N \cdot H)^{\alpha'} I_s) $$

V tomto prípade súradnice *u* a *v* získame skalárnymi súčinmi $L \cdot N$ a $N \cdot H$. Vypočítané súradnice vo forme premenných *varying* pošleme do fragment, a textúrovanie vykonáme rovnakým spôsobom ako v predchádzajúcom prípade.

V scéne, ktorá je časťou tejto práce, budú tieto dve riešenia anizotropického osvetlenia umiestnené vedľa seba, aby rozdiel medzi výpočtom vo vertex a vo fragment shaderu bol jednoznačne viditeľný.

## 5.3 Vertex morphing

Pri návrhu *vertex morphing* som dlho rozmýšľal nad tým, na akom geometrickom telese alebo modeli by sa dal tento efekt najlepšie prezentovať. Z jednoduchých útvarov som skúšal guľu, kocku, valec ale výsledná animácia nevyzerala tak, ako som si ju predstavoval. Experimentoval som s väčšími komplexnejšími modelmi, ale pri aplikácii týchto modelov sa vyskytli problémy, preto som si zvolil model bezmotorového lietadla z príkladov OSG.



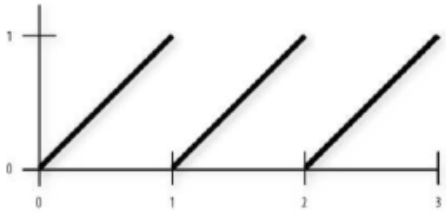
V kapitole **Teória** je rozobraný spôsob morfovania daného objektu do iného. Pozície vrcholov sú interpolované miešajúcim faktorom. Vypočítanie pozícií, do ktorých budú vrcholy modelu interpolované je náročný proces. Hlavnou otázkou bolo to, „kde“ tento výpočet vykonať: v OpenSceneGraph alebo vo vertex shaderu. Nakoniec som sa nerozhodol použiť tretiu možnosť výpočtu: predbežné spracovanie dát ešte pred interpoláciou. S predbežným spracovaním je možné zvýšiť efektívnosť a rýchlosť aplikácie. Jeho hlavnou výhodou je, že pri behu aplikácie *vertex morphing* nemusí počítať robiť nič iné, než interpoláciu samotnú.

Interpolácia sa vykoná vo vertex shaderu. Vektor zdrojovej pozície je pozícia vertexu, čo je k dispozícii v GLSL shaderu vo *vertexovom atribúte gl_Vertex*. Vektor cieľovej pozície je nutné nejakým spôsobom do shaderu dostať. Pole tohto vektoru je v OSG uložené v jednom vektorovom poli. S *uniform premennými* je možné poslať do shaderu aj pole vektorov, ale premenné sú aktualizované v každom snímku, a shader potrebuje iný vektor pre každý vertex.

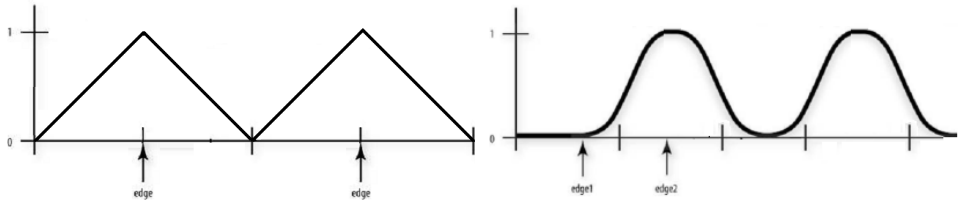
Vhodným riešením je poslať pole vertexov do shaderu ako vertex atribút. Hodnoty týchto premenných sú aktualizované per-vertex, takže ku každému vertexu je možné priradiť vlastný cieľový vektor.

Ďalšou funkčnou metódou by bolo poslať pole vertexov do shaderu ako textúrovacie súradnice. V triede *Geometry* je možné priradiť k vertexom vlastné textúrovacie súradnice. V prípade nefunkčnosti predchádzajúceho riešenia, je možné použiť túto metodu ako alternatívu.

Interpolácia vo vertex shaderu sa počíta pomocou miešaného faktoru. Tento faktor sa mení v čase, preto s *uniform premennou* treba poslať do shaderu informáciu o tom, ako dlho už aplikácia beží. Faktor je vlastne funkcia času, ktorá sa pohybuje medzi hodnotami nula a jedna. Funkcia je periodická, preto čas aplikácie musíme spracovať tak, aby bol faktor vhodný pre interpoláciu. Prvým krokom je normalizovať čas, aby bežal periodicky od nuly do jednej.




Vykreslený bude len jeden model, na rozdiel od kapitoly **Teória**, kde sú uvedené dva. Aby animácia fungovala správne, treba ešte opraviť funkciu faktoru, a tak už môžeme morfovať zo zdrojového modelu cieľový a späť. Výsledný vzhľad funkcie závisí na tom, akú interpoláciu potrebujeme:




Na ľavej strane je lineárna interpolácia, rýchlosť výslednej animácie je konštantná. Na pravej strane je interpolácia hladšia a rýchlosť animácie bude nižšia keď hodnota funkcie miešajúceho faktoru bude blízko k jej extrému.

Aktuálnu pozíciu získanú interpoláciou je potrebné násobiť s maticou *gl_ModelViewProjectionMatrix* aby vertex bol transformovaný do *image space*.

## 5.4 Animovaná tráva

V kapitole **Teória** je popísané ako vypočítať vo vertex shaderu reálne vyzerajúce pohyby trávy, teraz sa budeme zaoberať scénou na ktorú bude aplikovaný vertex shader.

Základ scény tvorí textúrovaný štvorec, ktorý predstavuje povrch zeme, kde rastie tráva. Steblá trávy sú modelované dvomi navzájom kolmými obdĺžnikmi, ďalej bude táto kompozícia dvoch polygónov nazývaná *objekt trávy*. Na polygóny bude pri vykresľovaní pridaná textúra s alfa kanálom, aby sme dosiahli efekt samostatných stebiel trávy. [6]

Čierne pixely na obrázku majú alfa hodnotu nula. Pri textúrovaní budú tieto oblasti prehľadné a vo výsledku budú viditeľné len steblá trávy.

*Objekty trávy* sú postavené z dvoch polygónov, ktoré majú štyri vertexy. Pohybovať sa budú len horné vertexy. Do shaderu je nutné poslať informácie o tom, ktoré vertexy nemajú byť posunuté. Na toto je možné použiť vertex atribút, alebo textúrovaciu súradnicu.

Ďalším krokom je vygenerovanie *objektov trávy* v potrebnom počte a náhodnom rozložení v scéne. Na túto úlohu je možné definovať funkciu, ktorá vráti ukazovatele na *objekty trávy*, ktorými môžeme ich pripojiť ku scéne. Generovanie náhodných x-ových a y-ových súradníc je možné vykonať v cykle. Vzhľadom na to, že renderujeme prehľadné polygóny, je nutné nastaviť správne poradie vykresľovania. Renderovanie sa musí začínať u najvzdialenejších uzlov, a postupne sa majú vykresliť objekty bližšie ku kamere. Bez tohto nastavenia sa po renderovaní objavujú artefakty. OSG renderuje uzly v poradí pridania ku scéne. V našom prípade sú uzly náhodne generované, teda poradie uzlov nie je správne, preto je nutné nastaviť vykresľovanie od najvzdialenejších objektov k bližším.

Osvetlenie vlniacej sa trávy sa zmení vzhľadom na to, že steblá trávy sa pohybujú a pri silnom vetre sa môžu aj otočiť. Zmeny v osvetlení môžeme riešiť so zmenou zelenej zložky farby, aby jednotlivé steblá trávy boli iného odtieňa zelenej farby.

# 6 Implementácia vybraných shader efektov

Predchádzajúce kapitoly popisujú stavbu scény v *OpenSceneGraph*, činnosť vykresľovacieho reťazca, základy jazyka *GLSL*, teoretické základy a návrh vybraných grafických efektov. V tejto kapitole budú k implementácii anizotropného osvetlenia, vertex morphing a animácii trávy využívané poznatky predchádzajúcich kapitol.

V kapitole **Návrh** je u každého efektu popísaný vzhľad scény a shaderov. Scény majú časti, ktoré sú podobné u každého efektu, napr. načítanie shaderov a textúry.

Základom každej scény je ***uzol root***, a ostatné uzly sa k nemu pripojujú. Uzol je objekt typu *Group*, a deklaruje sa nasledujúcim spôsobom:

```
osg::ref_ptr<osg::Group> root ( new osg::Group )
```

Trieda `ref_ptr<>` implementuje ukazovateľ na objekt odvodený od *Referenced*. Obsahuje čítač odkazov ukazujúcich na tento odvodený objekt. Objekt sa sám zmaže, keď počet odkazov na neho ukazujúcich je nula.

**Pripojenie uzlov** ku scéne je pomocou uzlov typu *Group*. Tento typ objektov organizuje uzly do stromovej štruktúry. Ku *Group* sa pripojujú uzly metódou `addChild()`:

```
root->addChild( uzol )
```

Metódu `addChild()` obsahuje ešte aj uzly *LOD*, *Switch* a *MatrixTransform*.

Modely v objekte *Drawable* sa ku scéne pripojujú pomocou uzlov *Geode*. Pridať objekt *Geometry* k uzlu *Geode* je možné nasledujúcim spôsobom:

```
Geometry_node->addDrawable( geometry )
```

Stavový objekt *StateSet* vytvoríme a pripojíme k uzlu takto:

```
osg::ref_ptr<osg::StateSet> stateset = uzol->getOrCreateStateSet()
```

**Načitanie a pripojenie shaderov** je pomocou objektov *Shader*, *Program* a funkcie `loadShaderSource()`, ktorá zo súboru načíta zdrojový kód shaderu do *shader objektu*, a hlási chybu keď súbor neexistuje:

```
Program->addShader( fragmentShader.get() );
Program->addShader( vertexShader.get() );
loadShaderSource( vertexShader.get(), "vertex.vert" );
loadShaderSource( fragmentShader.get(), "fragment.frag" );
stateset->setAttributeAndModes
                ( Program.get(),osg::StateAttribute::ON );
```

Posledným príkazom pripojujeme *shader program* k uzlu.

**Textúry** sa pripojujú ku *StateSet* objektom, do shaderu sú prenášané s *Uniform* objektmi.

```
texture->setImage( texture_image );
```

```
stateset->setTextureAttributeAndModes
                    ( 0,texture.get(),osg::StateAttribute::OFF );
stateset->addUniform( new osg::Uniform( "texture", 0 ) );
```

Textúra je načítaná do *Image* objektu `texture_image` zo súboru. `texture` je textúrovým objektom a *StateSet* uzly je `stateset`. Textúra v shaderu bude dostupná v *uniform premennej* `texture`.

**Vykresľovanie scény** prebieha vždy podobným spôsobom, pomocou triedy *osgViewer::Viewer*. Objekt z tejto triedy zobrazuje scénu na obrazovke v samostatnom okne. Príkazom `viewer.setSceneData(root)` sa nastavuje koreňový uzol scény. Metóda `run()` objektu *viewer* spustí okno pre renderovanie.

Aby manipulácia (**rotácia a posun scény, zoom**) s vykresľovanou scénou v samostatnom okne bolo jednoduché, je k zobrazovaciemu *viewer* objektu pripojený objekt *TrackballManipulator*:

```
viewer.setCameraManipulator(new osgGA::TrackballManipulator)
```

Implementácia týchto operácií je rovnaká u každej aplikácie, preto nebude popísaná zvlášť v popise jednotlivých efektov. Pre každú aplikáciu platí, že vykresľovanie prebieha pomocou triedy *osgViewer::Viewer*, manipulácia so scénou vo vykresľovacom okne je riešená objektom *TrackballManipulator*. Shadery a textúry budú pripojené k aplikácii podobným spôsobom ako sú implementované v predchádzajúcich odstavcoch.

## 6.1 Anisotropic lighting

### 6.1.1 Scéna

Výsledná scéna pozostáva z dvoch jednoduchých geometrických telies, ktoré budú osvetlené anizotropicky. Na prvé teleso bude aplikovaný shader s výpočtom osvetlenia vo fragment shaderu, na druhé shader s per-vertex osvetlením.

Z dôvodu dvoch telies v scéne je ich nutné umiestniť tak, aby sa navzájom neprekrývali. Na ľubovoľné transformácie časti scény slúži uzol typu *MatrixTransform*. V tomto prípade posunujeme obidve telesá, preto potrebujeme dva transformačné uzly, ku každému telesu jeden.

```
osg::ref_ptr<osg::MatrixTransform> transform1 =
                    new osg::MatrixTransform();
transform1->setMatrix( osg::Matrix::translate(-7.5,0,0) );
root->addChild( transform1.get() );
```

Transformačný uzol transformuje svojich potomkov podľa matice. *Matrix* objekt v OSG postaví z vektoru posunutia maticu, pomocou ktorej potom transformujeme synovské uzly transformačného uzla. Podobným spôsobom vytvoríme transformačný uzol `transform2` pre druhé teleso scény. Vektor posunutia bude `(7.5,0,0)`.

OSG poskytuje zabudované modely jednoduchých geometrických telies, napr. kocka, guľa, valec. Tieto telesá sú odvodené z triedy *osg::Shape*. Ku scéne budú pripojené dva z týchto objektov. Trieda

gule je *osg::Sphere*. Tento zabudovaný typ modelu je pripojený k objektu *ShapeDrawable*, ktorý je odvodený od *Drawable*. Objekt *ShapeDrawable* pripojíme k uzlu `geode1` typu *Geode*.
Aby osvetlenie fungovalo správne, je nutné normalizovať normály gule. V OSG je možné používať v *StateSet* objektoch premenné *OpenGL*, ktoré slúžia ku zmene stavu, metódou *setMode()*.

```
osg::ref_ptr<osg::Sphere> sphere1 =
                  new osg::Sphere(osg::Vec3(0.0,0.0,0.0), 5.0);
osg::ref_ptr<osg::ShapeDrawable> unitSphere1 =
                  new osg::ShapeDrawable( sphere1.get() );
osg::ref_ptr<osg::StateSet> drawableState1 =
                  unitSphere1->getOrCreateStateSet();
drawableState1->setMode
                   (GL_RESCALE_NORMAL, osg::StateAttribute::ON);
geode1->addDrawable(unitSphere1.get());
transform1->addDrawable(geode1.get() );
```

Týmto spôsobom vytvoríme aj druhú guľu a pripojíme k transformačnému uzlu `transform2`.

Pripojenie shaderov ku guliam je implementované podobne ako už bolo vyššie popísané. Prvá guľa dostane per-fragment anizotropické osvetlenie. Do *Shader* objektov `vertexShader1` a `fragmentShader1` sa načítajú zdrojové kódy vertex a fragment shaderov. Objekty sa pripojujú k `shaderProgram1`, čo pridáme do `geodeState1` (je *StateSet* objektom `geode1`). Podobne pripojíme aj per-vertex osvetlenie k druhej guli, v názvoch objektov miesto `1` bude `2`.

Do shaderov je ešte nutné poslať textúry. Ku `geodeState1` budú pripojené tri textúry: textúra difúznej a spekulárnej zložky, a textúra so smerom anizotrópie.
Ku `geodeState2` pripojujeme len jednu textúru, čím bude texturovaná druhá guľa.

V scéne bude ešte aj zdroj svetla. V OSG sa do scény pripojuje svetelný zdroj z triedy *osg::LightSource*. Pripojuje sa priamo ku koreňovému uzlu. Ku svetelnému zdroju je nutné pripojiť objekt typu *Light*, čo vlastne predstavuje svetlo svetelného zdroja. Objekt *Light* má viac zabudovaných premenných, ktoré nastavia vlastnosti svetla. Dôležitou vlastnosťou je index svetla, čo slúži k jeho identifikácii. V našom prípade bude mať tento index hodnotu `0`. V *GLSL* je možné získať informácie o vlastnostiach svetla pomocou zabudovanej premennej *gl_LightSource*. Implementácia pripojení svetla do scény je nasledujúca:

```
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;
osg::ref_ptr<osg::Light> light = new osg::Light();
ls->setLight( light.get() )
ls->setStateSetModes( rootstateset, osg::StateAttribute::ON );
root->addChild( ls.get() );
```

Predposledným príkazom „zapneme“ svetelný zdroj.

V OSG ešte pripojujeme *TrackballManipulator* a koreňový uzol scény k objektu `viewer`. Posledným krokom je `viewer.run()`.

### 6.1.2 Per-fragment shader

Celý výpočet anizotropického osvetlenia, výpočet tangenty a transformácia vektorov sa vykoná vo fragment shaderu. Vertex shader len pošle do fragment shadera normálu vo forme *varying* premenných, smer osvetlenia *L*, pozorovací vektor *V* a textúrové súradnice. Do zabudovanej premennej *gl_Position* zapíšeme hodnotu funkcie *ftransform()*, čo simuluje činnosť fixnej reťazca *OpenGL*.

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

Vo fragment shaderu z textúry `anisotex` získame hodnotu tangenty:

```
vec3 T = texture2D(anisotex, gl_TexCoord[2].st).rgb;
```

Binormálu vypočítame pomocou vektorového súčinu normály a tangenty, potom zostavíme *TBN maticu*.

```
vec3 B = cross(N, T);
mat3 TBNMatrix = mat3(T, B, N);
```

Pomocou *TBN matice* transformujeme smer osvetlenia *L*, pozorovací vektor *V*.

```
L *= TBNMatrix;
V *= TBNMatrix;
```

Potom vypočítame skalárne súčiny $L \cdot T$ a $V \cdot T$ a výsledky transformujeme do intervalu <0,1>.

```
float dotTL = dot(T, L);

float dotTV = dot(T, V);
vec2 look = vec2(dotTL, dotTV);
look = (look * 0.5) + 0.5;
```

Difúzna a spekulárna zložka osvetlenia je kódovaná v textúrach, zložku ambient je možné vypočítať z dvoch konštánt. Difúznu a spekulárnu zložku získame podobne ako tangentu, textúrovacími súradnicami budú vypočítané skalárne súčiny.

```
vec4 diffuse = gl_LightSource[0].diffuse * gl_FrontMaterial.diffuse;
diffuse *= texture2D(base, look);
vec4 specular = texture2D(specular, look) *
        gl_LightSource[0].specular * gl_FrontMaterial.specular;
vec4 ambient = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
```

Výsledná hodnota farby fragmentu bude súčet troch zložiek osvetlenia.

```
gl_FragColor = diffuse + specular + ambient;
```

Zápisom do premennej *gl_FragColor* končí činnosť fragment shaderu. Fragmenty alebo pixely sa zobrazia na obrazovke.

### 6.1.3 Per-vertex shader

Tento variant anizotropického osvetlenia je veľmi jednoduchý. Vo vertex shaderu sa zo zabudovaných premenných vypočíta smer svetla a normála.

```
vec3 L = normalize(gl_LightSource[0].position.xyz - gl_Vertex.xyz);
vec3 N = normalize(gl_NormalMatrix * gl_Normal);
```

Vypočítajú sa skalárne súčiny $L \cdot N$ a $N \cdot H$, ktoré budú slúžiť ako textúrovacie súradnice. Premenná `tex` je *varying* a slúži na prenos textúrovacích súradníc do fragment shaderu. Vektor *H* je *halfway vector*, v *GLSL* existuje zabudovaná premenná, ktorá túto hodnotu udáva.

```
tex.y = dot(N, gl_LightSource[0].halfVector.xyz);
tex.x = dot(L, N);
```

Vo vertex shaderu je ešte nutné zapísať výslednú pozíciu vertexu.

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

Textúry pre difúznu a spekulárnu zložku je možné zlúčiť do jednej textúry. Difúzne hodnoty sú kódované vo farbách, spakulárna hodnota v alfa kanále. Shader získa hodnotu farby z textúry pomocou textúrovacích súradníc z vertex shadera.

```
Vec4 color = texture2D(vtexture, tex);
```

Pred zápisom hodnoty do premennej *gl_FragColor* násobíme ju dvomi, aby spekulárna zložka bola výraznejšia.

```
gl_FragColor = color * 2;
```

## 6.2 Vertex morphing

Animácia vybraného modelu s metódou vertex morphing je implementovaná vo vertex shaderu. Základom scény je koreňový uzol *root*. Model je načítaný zo súboru *drak.osg* do uzla *Node* a priamo sa pripája ku koreňovému uzlu.

```
osg::Node* modelNode = NULL;
modelNode = osgDB::readNodeFile("drak.osg");
```

Vertex a fragment shader pripájame podobným spôsobom ako u anizotropického osvetlenia. *Program* objekt `morphShaderProgram` obsahuje dva shadery, pripojuje sa ku stavom uzla `modelNode`.

```
morphShaderProgram->addShader(morphVertexSh.get());
morphShaderProgram->addShader(morphFragmentSh.get());
modelStateSet->setAttributeAndModes
        (morphShaderProgram.get(), osg::StateAttribute::ON);
```

Vertex shader pre animáciu potrebuje časovú premennú, hodnota ktorej sa mení časom. Táto časová premenná je implementovaná pomocou *Uniform UniformCallback* objektov.

```
osg::ref_ptr<osg::Uniform> timeA =
             new osg::Uniform("time", time);
timeA->setUpdateCallback(new timeCallback());
modelStateSet->addUniform(timeA.get());
```

*UniformCallback* v každom snímku spustí funkciu, ktorá aktualizuje *uniform* premennú v shaderu. V OSG je možné získať hodnotu, ktorá udáva, ako dlho už beží aplikácia.

```
time = nv->getFrameStamp()->getReferenceTime();
```

Takým spôsobom je možné aktualizovať z *UniformCallback* premennú `time` v každom snímku. Vo vertex shadere premenná bude fungovať ako hodiny.

`modelNode` pripájame ku koreňovým uzlom, nastavíme a spustíme *viewer* objekt.

Aby interpolácia bola možná vertex shader potrebuje vektor cieľovej pozície. Tieto hodnoty sú prenesené do shaderu ako textúrovacie súradnice. Tieto súradnice obsahuje súbor z ktorého je model načítaný.

Ovládač grafickej karty od spoločnosti *ATI* neumožnil aby sa pole cieľových vertexov poslali do shadera pomocou vetex atribútov z OSG.

Vektor cieľovej pozície sa dostane do vertex shaderu ako aktuálne textúrovacie súradnice v premennej *gl_MultiTexCoord0.xyz*.

```
vec3 vertex1 = gl_MultiTexCoord0.xyz;
```

Do fragment shaderu je poslaný vektor normály a smer svetla *varying premenným*, aby bolo možné počítať jednoduché osvetlenie

```
normal = normalize(gl_Normal);
light = gl_LightSource[0].position.xyz;
```

Do vertex shaderu sa aktualizovaný čas dostane vo forme premennej `time`. Z hodnoty tejto premennej je nutné odčítať jej celú časť, aby sme dostali hodnotu menšiu ako 1.

```
float ttime = fract(time*0.5);
```

Pre animáciu potrebujeme hladšiu funkciu miešajúceho faktoru aby rýchlosť interpolácie nebola konštantná.

```
float factor = smoothstep(0.0, 0.5, ttime)-
                        smoothstep(0.5, 1.0, ttime);
```

S faktorom `factor` môžeme interpolovať vertexy medzi zdrojovou a cieľovou pozíciou.

```
position.xyz = mix(vertex1, gl_Vertex.xyz, factor);
```

Aktuálnu pozíciu transformujeme do *image space* a potom ju priradíme premennej `gl_Position`.

```
gl_Position = gl_ModelViewProjectionMatrix * position;
```

Vo fragment shaderu vypočítame difúznu zložku *Phongovho osvetlenia* a sčítame s vektorom zelenej farby. Výsledok zapíšeme do premennej `gl_FragColor`.

```
vec4 diffuse = dot(normal, light) *
                        gl_LightSource[0].diffuse;
gl_FragColor = diffuse + vec4(0.3, 0.7, 0.3, 0.0);
```

## 6.3 Animovaná tráva

V zdrojovom kóde sú definované dve konštanty: prvá je počet objektov trávy `MAX_QUAD_NUMBER`, druhá je `GROUND_SIZE`, ktorá udáva veľkosť základnej plochy na ktorej sú umiestnené *objekty trávy*.

Základnou plochou v scéne je objekt *Geometry* `groundGeometry`, ktorý je pripojený ku koreňovému uzlu cez uzly `baseGeode`. Plocha pozostáva zo štyroch vertexov, ktoré sú ukladané v poli vektorov `groundVertices` a sú pripojené ku `groundGeometry`.

```
groundGeometry->setVertexArray(groundVertices.get());
```

Na ploche bude textúra, preto je nutné nastaviť textúrovacie súradnice jednotlivých vertexov. Pole s textúrovacími súradnicami sa pripojuje ku `groundGeometry`. Prvý parameter je index súradnice pre vertex shader.

```
groundGeometry->setTexCoordArray( 0, texcoord );
```

Základnú plochu vykreslíme zo štyroch vertexov.

```
groundGeometry->addPrimitiveSet
                    (new osg::DrawArrays( GL_QUADS, 0, 4 ));
```

Premenná `GL_QUADS` znamená, že kresleným polygónom bude štvorec.

Načítame textúru `groundTex` a pripojíme ju pomocou `groundStateSet` k základnej ploche.

*Objekty trávy* budú generované, preto nie je možné ku každému vytvoriť vlastný *StateSet* a shadery pripojiť ku každému zvlášť. Preto najprv vytvoríme `grassStateSet`, stavový objekt, pripojíme k nemu shadery, a nastavíme všetky potrebné vlastnosti. Potom tento stavový objekt pripojíme ku každému objektu trávy

Najprv pridáme shadery a textúru stebiel trávy ku `grassStateSet`. Tieto operácie sa vykonajú podobne ako u predchádzajúcich efektov. Textúra v shaderu bude v premennej `grass`.

Aby objekty trávy boli priehľadné, musíme túto vlastnosť nastaviť pomocou *OpenGL* stavov.

```
grassStateSet->setMode
                    ( GL_BLEND, osg::StateAttribute::ON );
grassStateSet->setRenderingHint
                    ( osg::StateSet::TRANSPARENT_BIN );
```

Veľmi dôležité je správne poradie vykresľovania priehľadných objektov trávy. Je nutné nastaviť renderovanie od najvzdialenejších objektov k najbližším. Vytvoríme *RenderBin* objekt `renderbin`, a nastavíme renderovanie na `SORT_FRONT_TO_BACK`. Automatické *depth test* je nutné vypnúť, aby náš `renderbin` fungoval správne.

```
osgUtil::RenderBin* renderbin = new osgUtil::RenderBin;
renderbin->setSortMode
                ( osgUtil::RenderBin::SORT_FRONT_TO_BACK );
renderbin->setStateSet( grassStateSet.get() );
grassStateSet->setMode
                    (GL_DEPTH_TEST, osg::StateAttribute::OFF);
```

Animácia trávy potrebuje podobné časové premenné, akú sme používali v efekte vertex morphing. Pripojenie a aktualizácia sú implementované rovnakým spôsobom.

Generovanie *objektov trávy* v počtu `MAX_QUAD_NUMBER` je pomocou funkcie:

```
osg::Geode* createIQuad(pos, stateset)
```

Parameter `pos` je vektor so súradnicami x, y, a udáva, kam bude *objekt trávy* vykreslený. Druhým parametrom je vstup do funkcie pre `grassStateSet`. Funkcia pripojuje k uzlu *Geode* dva *Geometry* objekty, ktoré sú navzájom kolmé, pretínajú sa v polovici a ich priesečník má rovnaké súradnice ako parameter `pos`. Pred vrátením *Geode* uzlu, funkcia k nemu pripojuje `grassStateSet`.

*Objekty trávy* sú generované v cykloch *for*.

```
for (int i = 0 ; i < MAX_QUAD_NUMBER; i++){
      float x = (rand() % ((GROUND_SIZE-2)*100)) / 100.0; x += 1;
      float y = (rand() % ((GROUND_SIZE-2)*100)) / 100.0; y += 1;
      osg::Vec2 position( x, y );
      grassIQuads[i] = createIQuad( &position, grassStateSet.get());
      grass->addChild( grassIQuads[i] );
}
```

Uzol `grass` je *Group* uzol a pripojuje objekty trávy ku koreňovým uzlom. Objekt *viewer* sa nastavuje podobne ako pri predchádzajúcich efektoch.

Vertex shader pracuje s vertexmi objektov trávy. Animuje len horné vertexy polygónov, dolné vertexy pozície nemenia. Keď druhý komponent premennej `gl_MultiTexCoord1` sa rovná jednej, tak sa jedná o spracovanie horného vertexu.

```
float flag = gl_MultiTexCoord1.t;
```

Ofsety, ktoré budú pripočítané k pozícii vertexu sú vypočítané kombináciou sínusových a kosínusových funkcií.

```
float offsetX = sin(vertex.x+time)*cos(vertex.x+time)*
                sin(vertex.x+time)*cos(vertex.x+time)*
                sin(vertex.x+time)*cos(vertex.x+time);
float offsetY =sin(vertex.y+time)*cos(vertex.y+time)*
                sin(vertex.y+time)*cos(vertex.y+time)*
                sin(vertex.y+time)*cos(vertex.y+time);
vertex.x += offsetX;
vertex.y += offsetY;
```

Z ofsetov sa počíta hodnota pre zmeny osvetlenia do fragment shaderu.

```
windgreen = (offsetX+offsetY)/2;
```

Výsledná pozícia vertexu je transformovaná do *image space* a je priradená k premennej `gl_Position`.

```
gl_Position = gl_ModelViewProjectionMatrix * vertex;
```

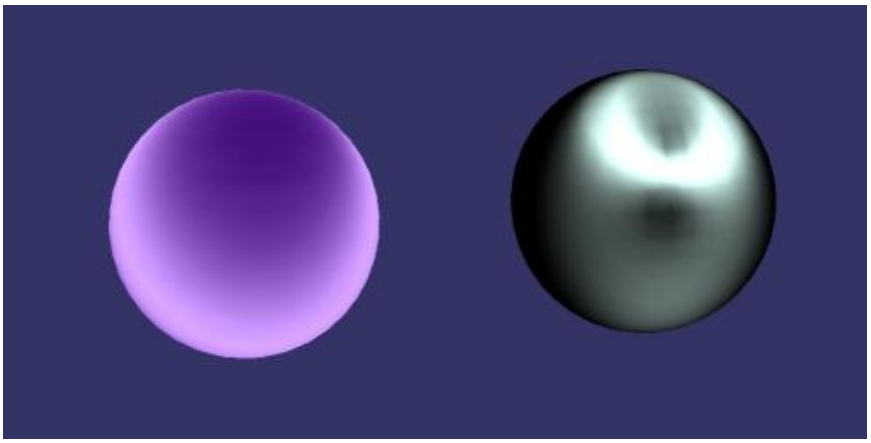
Fragment shader vykoná textúrovanie, pred priradením farby k premennej `gl_FragColor` zmení zelenú zložku farby. Hodnota premennej `windgreen` z vertex shaderu sa odčíta od zelenej zložky. Týmto procesom sú aplikované zmeny osvetlenia na trávu.

```
vec4 color = texture2D(grass, gl_TexCoord[1].st);
color.g -= windgreen;
gl_FragColor = color;
```

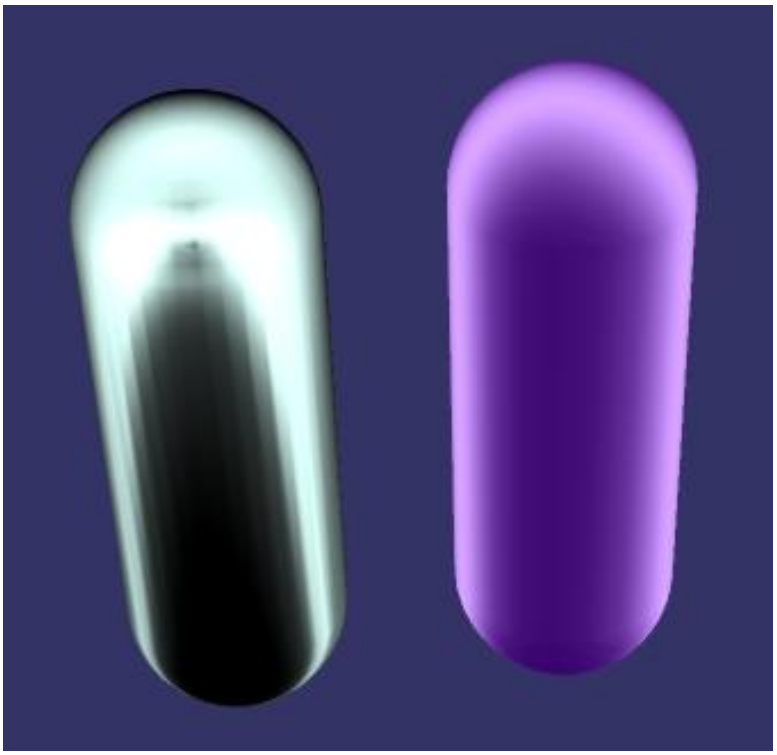
# 7 Výsledky

Výsledné grafické efekty budú zobrazené v tejto kapitole. Ku každému efektu sú uvedené dva obrázky. Skúsil som vybrať také obrázky, ktoré najviac demonštrujú výsledok mojej práce. U vertex morphing a u animovanej trávy nie je možné zobraziť animáciu na obrázkoch, preto som vybral snímky, na ktorých sú stavy animácie rozlíšiteľné.

## 7.1 Anisotropic lighting

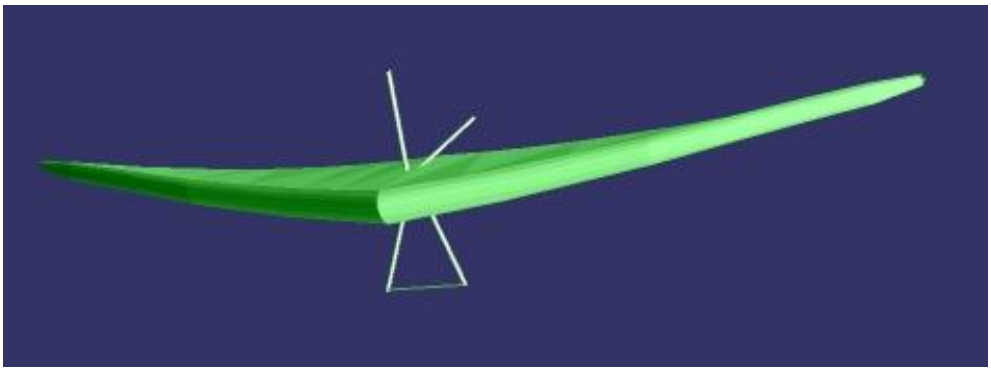


Gule s anizotropickým osvetlením vyzerajú ako ozdobné gule na vianočnom stromčeku.

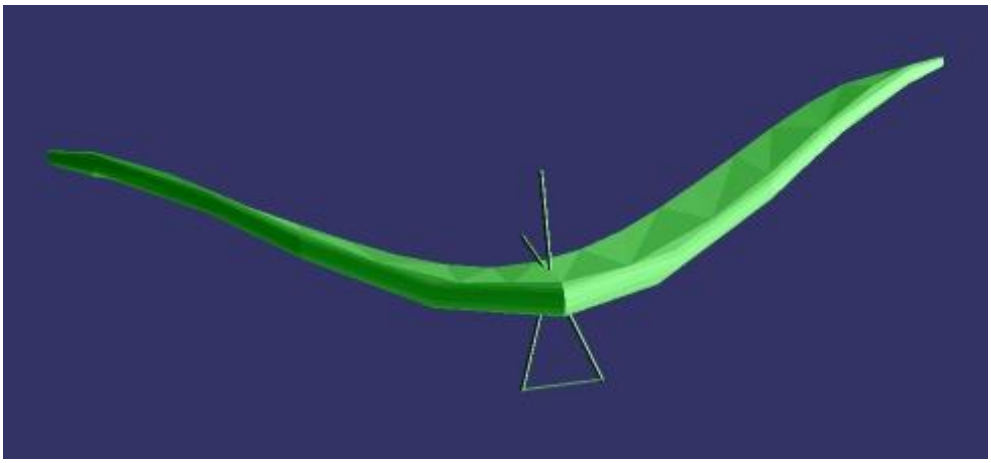


Na tomto obrázku veľmi dobre vidieť rozdiel medzi per-vertex a per-pixel implementáciou anizotropického osvetlenia.

## 7.2 Vertex morphing

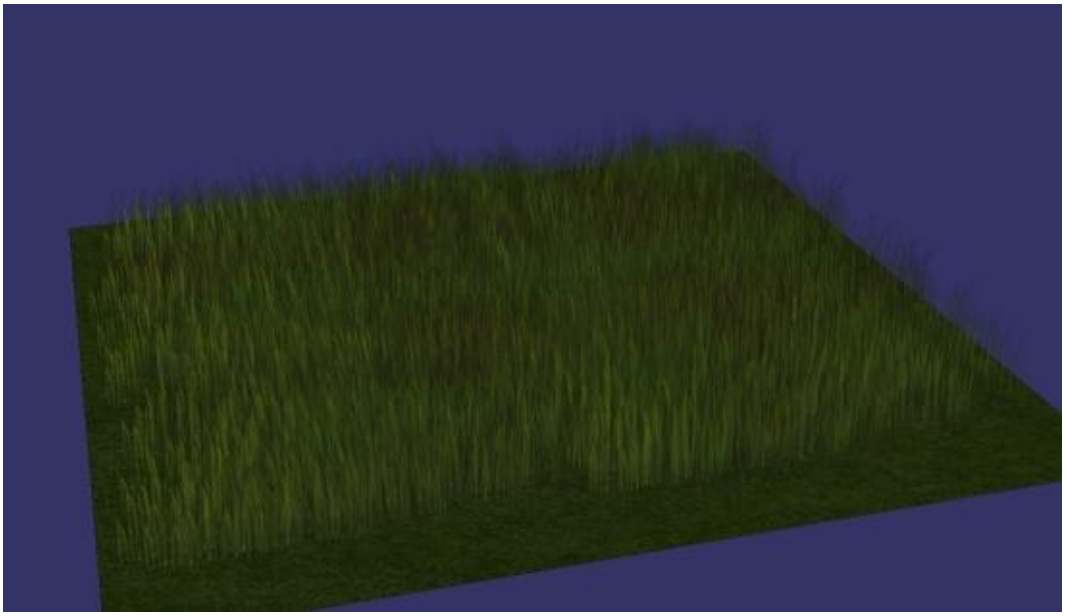


Na tomto obrázku sú krídla bezmotorového lietadla v zdrojovej pozícii.



Vertexy sú interpolované do cieľovej pozície.

## 7.3 Animovaná tráva



Animovaná tráva z diaľky. Na obrázku nie je vidieť pohyb stebiel trávy.

Animovaná tráva zblízka. Steblá trávy menia odtieň farby podľa ich pohybu.

# 8 Záver

Výsledkom tejto práce sú tri aplikácie, ktoré obsahujú implementáciu troch vybraných grafických efektov. V predchádzajúcej kapitole sú uvedené obrázky, ktoré predstavujú ako tieto moje aplikácie vyzerajú.

Zhodnotenie výsledných aplikácii ovplyvní aj fakt, že tieto grafické efekty sú základné a veľmi jednoduché. Stromové štruktúry grafov scény sú postavené zo základných uzlov a nevyužívajú pokročilé vlastnosti OpenScenerGaph. Neimplementoval som komplexné kompozície, len jednoduché scény s jedným alebo dvomi modelmi, dôležité bolo aby scéna prezentovala implementovaný efekt. Zhodnotenie výslednej práce môže byť podľa mňa pozitívne, podarilo sa mi implementovať základné efekty tak, ako som si ich predstavil pred vlastným programovaním.

Počas implementácie výsledných aplikácii tejto práce som potreboval základné vedomosti grafického rozhrania OpenGL. Knižnice OpenSceneGraph sú založené na tomto grafickom API a programovanie shaderov v jazyku GLSL súvisí s rozhraním. Táto práca bola veľmi dobrá na pochopenie a naučenie základov praktickej časti počítačovej grafiky. Tieto základné vedomosti by som si chcel rozšíriť v nasledujúcich rokoch návrhom a implementáciou komplexnejších scén. Implementované shaderové efekty tvoria dobrý základ k písaniu zložitejších, lepších a efektívnejších shader programov.

Vývoj výsledných aplikácií tejto práci v OpenSceneGraph bol veľmi pomalý, lebo som nenašiel vhodne vypracovanú referenciu o knižniciach rozhrania. Iné grafické programovacie rozhranie napr. OpenGL a Direct3D túto nevýhodu nemajú, preto na vývoj ďalších 3D aplikácií zvolím pravdepodobne iné rozhranie. Napriek tomu, že z času na čas som strávil hodiny hľadaním riešenia jednoduchého problému kvôli neexistujúcej referencii, zostavenie scény v OpenSceneGraph bolo veľmi jednoduché.

Možným pokračovaním tejto práce je zostavenie komplexnejšej scény z efektov, ktoré boli vytvorené skupinou študentov pracujúcich na bakalárskej práci v mojom ročníku s témou programovateľné shadery.

Implementácie vybraných grafických efektov počas vypracovania tejto práce som si užíval a v budúcnosti by som rád pokračoval v programovaní shaderov.

# Literatúra


[1] **Ros, Randi J.** *OpenGL Shading Language.* Second Edition. s.l. : Addison Wesley Professional, 2006. 0-321-33489-2.

[2] **Martz, Paul.** *OpenSceneGraph Quick Start Guide.* s.l. : Computer Graphics Systems Development Corporation, Mountain View, California, 2007.

[3] **Isidoro, John und Brennan, Chris.** *Per-Pixel Strand Based Anisotropic Lighting.* [Portable Document Format] s.l. : ATI Research.

[4] **Heidrich, Wolfgang und Seidel, Hans-Peter.** *Realistic, Hardware-accelerated Shading and Lighting.* [Portable Document Format] s.l. : Max-Planck-Institute for Computer Science.

[5] **Engel, Wolfgang.** *ShaderX3: Advanced Rendering with DirectX and OpenGL.* s.l. : Charles River Media, 2005. 9781584503576.

[6] **Isidoro, John, Vlachos, Alex und Brennan, Chris.** *Rendering Ocean Water.* [Portable Document Format] s.l. : ATI Research.

[7] **Dudash, Bryan.** *Anisotropic Lighting using HLSL.* [Portable Document Format] Santa Clara : NVIDIA Corporation , 2004.

[8] **Isidoro, John und Card, Drew.** *Animated Grass with Pixel and Vertex Shaders .* [Portable Document Format] s.l. : ATI Research.

# Príloha 1

## Manuál ku spustení demo aplikácie

Ku každému grafickému efektu implementovaného v rámci tejto práce existuje demo aplikácia. Aplikácie bežia na operačnom systému Microsoft Windows a potrebujú mať inštalovaný Microsoft Visual C++ 2005 Redistributable Package alebo MS Visual Studio. U každej aplikácie sú uvedené súbory potrebné ku správnemu spusteniu. U každej aplikácie sú priložené potrebné knižnice OpenScneGraph.

### Anisotropic lighting

Názov spustiteľnej demo aplikácie je aniso2_demo.exe. Súbory potrebné k aplikácie:

Textúry:

- anistotex.tga
- base.tga
- specular.tga
- tex.tga

Shadery:

- pixel_texture.vert
- pixel_texture.frag
- vertex_texture.vert
- vertex_texture.frag

### Vertex morphing

Názov spustiteľnej demo aplikácie je morphing_demo.exe. Súbory potrebné k aplikácie:

Shadery:

- morph.vert
- morph.frag

Model:

- drak.osg

### Animovaná tráva

Názov spustiteľnej demo aplikácie je quad_grass_demo.exe. Súbory potrebné k aplikácie:

Textúry:

- ground.tga
- grass.tga

Shadery:

- grass.vert
- grass.frag

V každej spustenej aplikácii je zobrazená scéna ovládaná myšou:

- scéna sa otočí s ľavým tlačítkom
- presunutie scény je pomocou prostredného tlačítka
- zoom je pomocou pravého tlačítka.

# Príloha 2

## CD s aplikáciami a s technickou prácou

Adresárová štruktúra:

**Demos** Demo aplikácie implementovaných efektov, jednotlivé aplikácie a k nim potrebné súbory sú v troch adresároch:

- **AnisoLight**
- **VertexMorph**
- **AnimGrass**

**Text** Technická správa v elektronickej podobe.

**Tutorial** Tutorial ku grafickým efektom vo formátu HTML stránok.

**Solution** Súbory ku kompilácii zdrojových kódov vo MS Visual Studio a vlastné zdrojové kódy.

**GLSLShaders** Shader programy v jazyku GLSL ku grafickým efektom.

**Textures** Textúry, ktoré sú potrebné pre správny beh aplikácie.

**Model** Model pre vertex morphing animáciu.

**Media** Videá s nahranými animáciami vertex mophing a animovanej trávy.

# Zoznam príloh