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

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ RŮŽIČKA
AUTHOR

BRNO 2014

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

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

GRAPHICS INTRO 64KB USING OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ RŮŽIČKA
AUTHOR

VEDOUCÍ PRÁCE Ing. TOMÁŠ MILET
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá problematikou tvorby grafického intra s omezenou velikostí. Popisuje jeho tvorbu a rozebírá metody a postupy, kterými se vyznačuje. Zabývá se procedurálním generováním, animací, vytvořením statických a dynamických textur a částicovými systémy.

Abstract

This work describes problem of creation graphic intros with limited size. It describes its creation and discusses the methods and procedures which characterize graphic intros. Main theme is about generation of procedural textures and models, animations, creation of static and dynamic textures and particle systems.

Klíčová slova

OpenGL, 64kB, Intro, procedurální generování, komprese spustitelného souboru, skybox, Phongův osvětlovací model, billboarding, GLSL, teselace

Keywords

OpenGL, 64kB, Intros, procedural generation, kompression of executable files, skybox, Phong reflection model, billboarding, GLSL, teselation

Citace

Jiří Růžička: Grafické intro 64kB s použitím OpenGL, bakalářská práce, Brno, FIT VUT v Brně, 2014

Grafické intro 64kB s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana inženýra Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Jiří Růžička
21. května 2014

Poděkování

Rád bych poděkoval Ing. Tomáši Miletovi za jeho rady a trpělivost v průběhu práce. Dále bych rád poděkoval rodině a přátelům za podporu při tvorbě a dokončování práce.

Obsah

1 Úvod **2**
1.1 Historie a současnost inter 2
1.2 Cíl práce 2

2 Teorie **3**
2.1 OpenGL 3
2.2 Grafické metody 6
2.3 Procedurální generování 8
2.4 Skybox 11
2.5 Generování planet 12

3 Implementace intra **14**
3.1 Knihovny 14
3.2 Skybox 14
3.3 Planeta 16
3.4 Asteroid 18
3.5 Slunce 21
3.6 Částicový systém 22
3.7 Animace 22

4 Spustitelný soubor **24**
4.1 Omezená velikost a kkrunchy 24
4.2 Experiment 25

5 Závěr **27**

A Plakát **29**

Kapitola 1

Úvod

Grafické intro je v podstatě upotávka či demo, které má ukázat schopnosti autora. Zpravidla se objevuje při startu aplikace a prezentuje jak programátorské schopnosti tak umělecké nadání autora ať již v hudbě nebo tvorbě 3D modelů. Intra většinou obsahují zajímavou scénu, 3D/2D efekty a zvukový doprovod.

1.1 Historie a současnost inter

Grafická intra začla vznikat v osmdesátých letech jako práce počítačových pirátů. Ti je vkládali před hry a programy, kterým odstranily ochranu proti kopírování. Intro sloužilo jako ukázka schopností a podpis daného člověka či skupiny. Intra častokrát vypada lépe jak samotné aplikace a jejich velikost byla 1 až 4kB.

Postupem času se z inter stala spíše umělecká díla. Autoři mezi sebou soupeří v několika kategoriích a snaží se udělat co nejlepší intro. Kdo by čekal, že taková intra nalezne při startu dnešních her a programů, tak by byl zklamán. Většinou se jedná o relativně velké videosekvence. Intra využívají nejnovejší možnosti grafických karet, aby měla co nejlepší efekty při stálé miniaturní velikosti.

1.2 Cíl práce

Cílem této práce je vytvoření jednoho takového intra s velikostí do 64kB. Scéna bude umístěna ve vesmíru. Procedurálně generovaný skybox zahrnuje hvězdy a mlhoviny. Dalším prvkem bude planeta podobná Zemi s oceány a kontinenty. Do této planety v závěru animace narazí veliký asteroid. Scéna bude osvětlena nedalekou hvězdou, pro efekt stínu a odlesku bude použit Phongův osvětlovací model. Cílem práce je zjednodušená simulace nárazu asteroidu do planety, kde spustitelný soubor nepřesahuje 64kB.

Kapitola 2

Teorie

Kapitola se zabýva prvky použitými pro vytvoření intra. Čtenář se zde seznámí s problematikou práce a možnostmi vytvoření grafického intra či jemu podobné aplikace. Jsou zde popsány prvky OpenGL, vysvětlení pojmu procedurální generování a jeho použití a grafické techniky, jako je Phongův osvětlovací model.

2.1 OpenGL

OpenGL [9] je soubor knihoven pro vytváření počítačové grafiky. V této kapitole se podívame na její součásti a zaměříme se na prvky použité v této práci.

2.1.1 GLSL - OpenGL Shading Language

GLSL je nedílnou součástí OpenGL API. Jedná se o programovací jazyk založený na syntaxi jazyka C obsahující specifické prvky pro grafiku. Grafická karta zpracovává části programu, ktere nazývame shadery. Shadery jsou zřetězeny za sebe a lze mezi nimy posílat data.

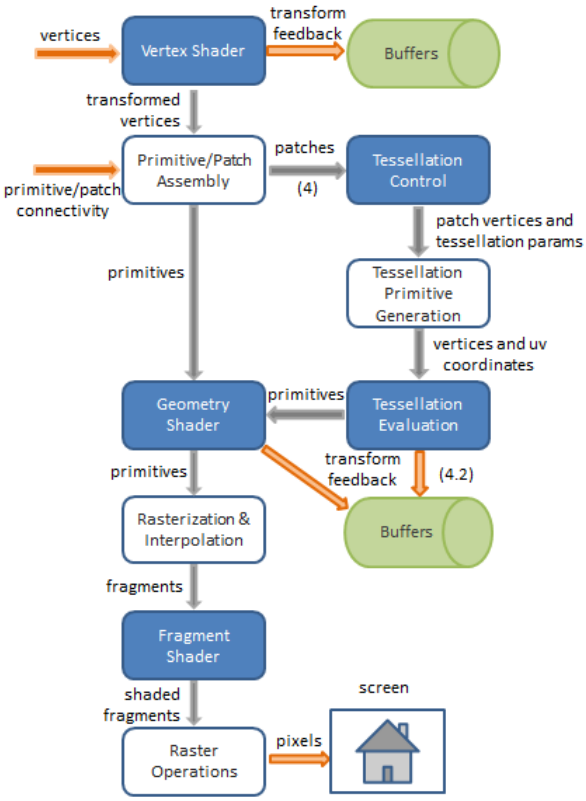
Pro generovaní grafiky je vhodné rychle zvládat operace s vektory a maticemi, GLSL tedy zahrnuje dvou až čtyř rozměrný typ vektoru (`vec2` až `vec4`) a matice (`mat2` až `mat4`). Nad těmito typy jsou přímo na grafické kartě implementovány základní operace s maticemi a vektory. Další datový typ je `sampler`, který slouží k uchování textury.

Kromě základních operací je implementován i rychlý přístup k prvkům vektoru (operátor swizzle). Pomocí tohoto operátoru lze přistupovat k prvků vektoru v daném pořadí. `Vector4.zxy` vytvoří vektor `vec3` z původního čtyřsložkového vektoru a naplní ho prvky původního vektoru v zapsaném pořadí.

Dále GLSL poskytuje širokou škálu funkcí pro míchání barev, matetické funkce, funkce pro práci s texturami a interpolační funkce.

2.1.2 VAO - Vertex buffer object

Drží v sobě odkaz na buffer(y) obsahující vertex data, tedy informace o pozici vrcholů primitiv, ale i texturovací koordináty, normály a další potřebné data. Uživatel si sám může definovat jaká data, s jakým posunem (`offset`) a v jakém pořadí zde budou uloženy. Jednotlivé typy bufferů dohromady tvoří VAO, v kterém je tak uložený celý objekt se všemy daty.



Obrázek 2.1: Grafická pipeline (převzato z [2]).

2.1.3 Shadery

Shadery jsou zpracovávány na grafické kartě jako sada zřetězených programů. Jedná se o malé programy, každy se specifickou funkcí a účelem. Tento způsob zpracování je velmi efektivní, grafická karta má zpravidla mnoho procesorů, v kterých jsou zpracovávána data paralelně pro jednotlivé vrchnoly, texely či fragmenty objektů z grafické scény[9]. Nyní si popíšeme některé typy shaderů.

Vertex shader

Základní shader, zpracovává vrcholy primitiv z VAO. Ty zde můžou být transformovány pomocí modelové, pohledové či projekční matice.

Tessellation control shader

Tento shader je odpovědný za řízení teselace, definujeme mu kolikrát se bude primitivum dělit. Také zde lze aplikovat úpravy pozice vrcholů primitiv (deformace objektu).

V počítačové grafice je teselací označován proces, kdy se z jednoduchého objektu vytváří složitější objekty s většími detaily, pomocí dělení primitiv. K tomuto účelu se používá například Catmull-Clark rozdělení povrchu[6]. Díky tomuto procesu můžeme z krychle vytvořit objekt podobný kouli, tedy s vyhlazeným povrchem. Díky tomuto principu lze u 3D modelu ušetřit mnoho dat a ty dopočítat podle potřeby při renderu scény.

Tessellation evaluation shader

Následuje po kontrolním shaderu. Jeho úloha je interpolovat data přijatá od kontrolního shaderu a přeposlat je do následujícího shaderu. Pro každé primitivum může být tento shader spuštěn několikrát.

Geometry shader

Tento shader slouží k vlastnímu přidávání a odebírání vrcholů v primitivech. Definujeme formát vstupních a výstupních dat a také počet vrcholů na výstupu. Tímto shaderem tedy ovlivníme tvar výsledného objektu.

Fragment shader

Tento shader je z hlediska obrazové informace nejdůležitější. Objekt zde má již jasný tvar a zbývá ho jen obarvit, určit stínování a odlesk (Phongův osvětlovací model). Dochází zde k obarvení rasterizovaných primitiv tak, že je pro každý pixel spuštěn program shaderu. Zpracování pixelů probíhá paralelně. Ta může být převzata z textury nebo vypočítána. Výstupem je barva a hloubka.

2.1.4 Textury

Textura je zpravidla dvourozměrné pole texelů se specifikovanou velikostí a formátem, kterým obalíme objekt, aby byl viditelný. Vzorek textury může být homogení či heterogení. Z textury lze brát data pro shader a nebo do ní vložit výstup z programu shaderu místo vykreslení výstupu přímo na obrazovku. Základní jednotkou textury je již zmíněný texel. Texely jsou při vykreslování mapovány do pixelů.

Rasterová textura

Klasický obrázek, komprimovaný nebo bez komprimace, je předem připravený. Pro grafické intro s omezenou velikosti je tento typ textury nevhodný, a pro svou velikost nelze využít.

Procedurální textura

Tento typ textury je generován po spuštění programu. Je proto ideální pro intro s omezenou velikostí. Rozměr a formát může být libovolný a kvalita textury je volitelná dle potřeby. Generování probíha pomocí různých algoritmů šumu, který je pak upraven tak, aby vypadal dle požadavků (např. jako mramor, dřevo, půda). Textůra může být generována pomocí algoritmu pro 3D šum a díky tomu lze obarvit objekty bez přechodu ze všech stran.

Cubemapy - kubické mapy

Speciální druh textury, který je vhodný pro 3D scénu, efekt, odrazy a osvětlení. Jedná se o šest dvourozměrných textur umístěnych do jednotkove krychle ve středu souřadnicového prostoru. Hodí se pro uložení textury např. pro skybox. Lze s nimy také obalit kulovité objekty tak, že objekt rovnoměrně obalíme.

2.1.5 FBO - Framebuffer object

Abychom mohli generovat procedurální texturu a uložit si jí do paměti, potřebujeme na to nastavit grafickou kartu. Framebuffer object se může skládat z několika 2D polí jako jsou barevné buffery, hloubkový buffer či stencil buffer.

K FBO si nejprve musíme vytvořit prázdné textury, těch může být i několik. Texturám nastavíme požadované parametry a připojíme je k FBO. Textury musíme nastavit jako attachment FBO. Výstupní hodnota fragment shaderu se po spuštění programu zapíše do nastaveného attachmentu. Poté aktivujeme textury a FBO je připraven k použití.

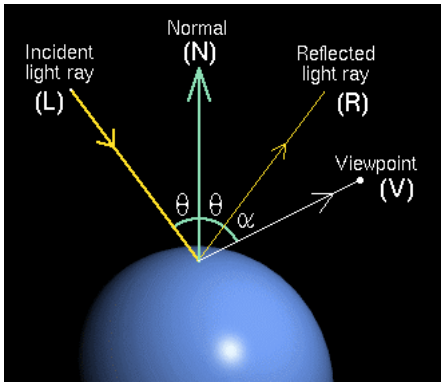
2.2 Grafické metody

Jedná se o způsoby, jak vylepšit zobrazovanou scénu tak, aby se více podobala reálnému světu. Také umožňují optimalizovat vykreslování.

2.2.1 Phongův osvělovací model

Osvětlovací model navržený Bui Tuong Phongem v roce 1973, založený na empirických zkušenostech nikoliv na fyzikálním modelu. Navzdory nepřesnostem vůči reálnému chování světa je výsledek velice dobrý a vypočetně málo náročný, proto se hodí pro real-time grafiku.

Ve výpočtu figurují 4 vektory. Vektor $\overrightarrow{N}$ je normála v bodě, který pozorujeme. K pozorovanému bodu vedeme vektor $\overrightarrow{V}$. Posledním prvkem je vektor paprsku dopadajícího světla $\overrightarrow{L}$ a jeho odraz vektor $\overrightarrow{R}$. Paprsků světla může být i více.



Obrázek 2.2: Zobrazení vektorů phongova modelu. Vektor světla L a jeho odraz L, normála v bodě odrazu N a vektor z místa pohledu V (převzato z [10]).

Odraz světla se v Phongově modelu skládá ze tří složek. Ambientní, difúzní a spekulární. Intenzita odrazu je pak dána součtem těchto složek.

$$C = C_a + C_d + C_s \tag{2.1}$$

Ambientní světlo

Je okolní rozptýlené světlo. Nelze určit jeho směr a tak se pouze vynásobí s odrazivostí daného materiálu.

$$C_a = I_a k_a \tag{2.2}$$

Difúzní světlo

Tento typ světla má konkrétní směr $\overrightarrow{L}$ a rozptyluje se rovnoměrně do prostoru a podle úhlu dopadu se mění intenzita. Pokud je vektorový součet normály a vektoru L roven nule nebo je záporný, pak je povrch odvrácen od zdroje světla.

$$C_d = I_d k_d(\overrightarrow{L} \cdot \overrightarrow{L}) \tag{2.3}$$

Spekulární světlo

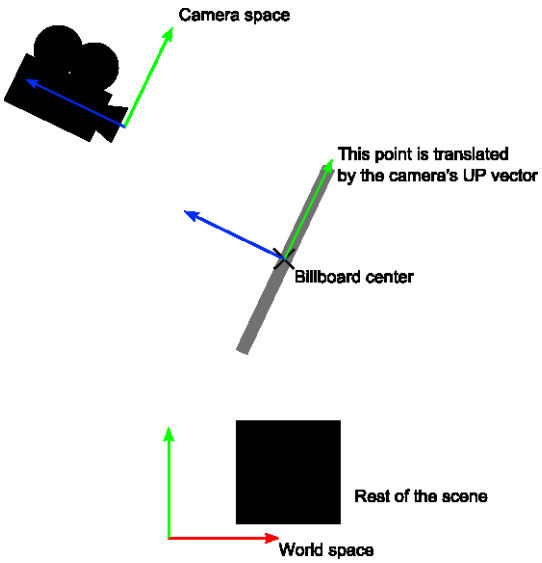
Světlá složka světla.Udává intenzitu světla odraženého převážně v jednom směru.

$$C_s = I_s k_s(\overrightarrow{V} \cdot \overrightarrow{R})^n \tag{2.4}$$

Kde `n` je Phongův exponent vyjadřující míru lesklosti.

2.2.2 Billboarding

Jedná se o postup kdy se do 3D scény zobrazují efekty nanesené na 2D plochu. Ty pak tvoří dojem 3D objektu, jelikož jsou stále natočeny směrem ke kameře. Tímto způsobem se pak generují například částicové systémy či ruzné zajimavé efekty jako kouř a oheň, protože jejich vykreslení ve 3D by bylo obtížné a efekt by byl velice podobný, ale náročnost výpočtu o mnoho větší. Lze tím tedy uspořit drahocený výpočetní výkon.



Obrázek 2.3: Ukázka generované hvězdné oblohy.

Na scéně generující vesmír tak lze jednoduše tvořit atmosféru planet, záři hvězd a další efekty například po srážce dvou objektů.

Pro vykreslení billboardu stačí mít jeden bod. V geometry shaderu si tento bod posuneme do správné pozice ve vykreslované oblasti pomocí ModelView matice, bod je získán z vertex shaderu:

```
vec4 clip_pos = Modelview * gl_in[0].gl_Position;
```

Z tohoto bodu potom generujeme 4 body, tak aby nám vytvořili čtverec. Přičteme/odečteme jedničku a získáme tak správné souřadnice. Body čtverce jsou násobeny projekční maticí, aby bylo docíleno správného natočení billboardu:

```
gl_Position = Projection*(clip_pos +
   vec4((-1+2*float(i%2)),(-1+2*float(i/2)),0,0)*0.1);
```

Tato metoda generování billboardu je velice snadná na implementaci a dá se lehko modifikovat, pro docílení velmi pestrých efektů.

2.3 Procedurální generování

S tvorbou grafiky, především s omezenou velikostí, souvisí generování textur a objektů pomocí algoritmů. Změny vzhledu celé scény pak lze dosáhnout velmi jednoduchou úpravou parametrů. U procedurálního generování je důležité aby výsledek práce byl pseudonáhodný. Tedy pro stejné parametry nám vygeneruje vždy stejný výsledek.

2.3.1 Funkce šumu a Perlinův šum

Nejčastějším prvkem v procedurálním generování jsou různě funkce šumu, ne všechny jsou však vhodné pro generování toho co potřebujeme.

Obyčejný šum generovaný běžným generátorem náhodných čísel, kterého se využívá běžně v informatice, je v této oblastí naprosto nevyhovující. Hlavní problém spočívá v tom, že šum je pokaždé náhodný a to i pro stejné vstupy. Při vykreslování scény bychom poté dostali pokaždé jiný výsledek, což je nežádoucí, protože pro některé případy by mohla scéna vypadat nevhodně. Dalším problémem je prudké střídání hodnot na výstupu, tyto generátory nemají spojitý výstup.

Příklad kongruentního generátoru čísel, generujícího čísla v intervalu $< 0, m)$:

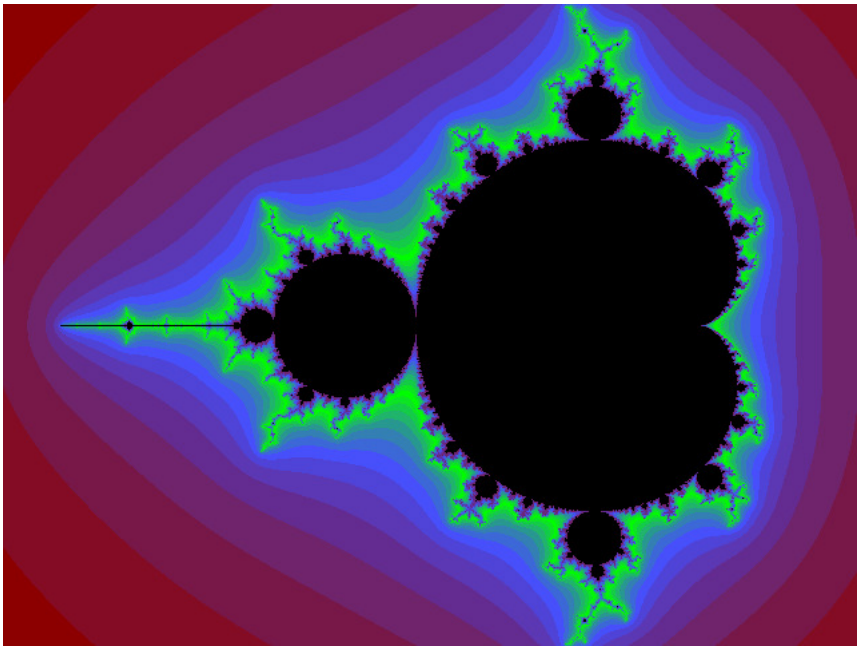
$$[h]x_{i+1} = (ax_i + c) \mod m \tag{2.5}$$

Kde x_i je vstupní hodnota, x_{i+1} je následující prvek v řadě, a je multiplikativní člen, c je aditivní člen a m omezuje interval zhora. Omezení rozsahu generátoru je podle architektury počítače.

V přírodě se u mnoha objektů dají objevit fraktální vlastnosti, viz [7]. Na první pohled se objekty jeví složité až neuspořádané, při podrobnějším zkoumání struktury bychom však objevili jisté vzory, jež jsou neměnné i pro různá měřítka. Jako ukázka může posloužit slavná Maldenbrotova množina.

Jednou z nejvhodnějších možností pro vizualizaci podobných jevů v počítačové grafice představuje Perlinův šum. Funkci představi roku 1985 Ken Perlin na konferenci o počítačové grafice a interaktivních technikách v New Yorku. Při této příležitosti představil šum, který je dnes nazýván jako *Pink noise*. Původně prezentovaná funkce šumu funguje na odlišných principech. U funkce Pink noise se dosahuje simulace fraktálních vlastností součtem několika funkcí šumu o různé frekvenci.

Ken Perlin dále vylepšoval šum, aby vyřešil problémy a omezení původní verze šumu. Vznikla tak funkce *Simplex noise* jejíž výhodou je menší počet interpolovaných hodnot, potřebných k získání hodnoty šumu v daném bodě. To znamenalo výrazné zrychlení výpočtu pro funkce s větším počtem dimenzí. Šum je také izotropní, nezávislý na směru, což může řešit mnoho problémů s anti-aliasingem.



Obrázek 2.4: Maldenbrotova množina, ukázka fraktálních vlastností.

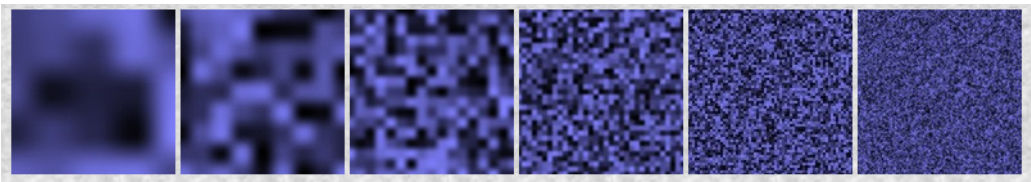
Perlinův šum [8]

Metoda generování Perlinova šumu je obdobná jako pro náhodné generátory. Vstupními hodnotami jsou pak reálné funkce, na jejíchž základě jsou generovány pseudonáhodné hodnoty. To znamená, že narozdíl od náhodných generátorů je výstup pro stejné vstupy je pokaždé stejný. Také pokud jsou rozdíly mezi vstupy malé, výstupní hodnoty si budou velice podobné.

Základem pro Perlinův šum je deterministický generátor pseudonáhodných čísel. Deterministický znamená, že vrací stejné hodnoty pro stejné vstupy, tím splňuje jeden z požadavků. Výstupem generátoru jsou čísla v intervalu $< -1; 1 >$.

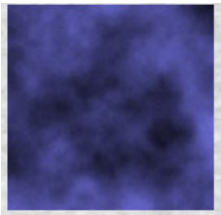
Čísla vygenerovaná tímto generátorem by nám, ale dávala moc hrubý výstup, proto se po vygenerování použije vyhlazení dat pomocí interpolace. Interpolací se také získávají hodnoty mezi jednotlivými prvky. Výsledná hodnota je průměrem aktuální hodnoty a hodnot okolních hodnot. Tím dojde k odstranění původní zrnitosti.

Dalším krokem je součet několika interpolovaných funkcí dohromady. Výsledkem součtu jednotlivých šumových funkcí je šum, který podle barevného provedení dokáže napodobit oblaka, dřevo, mramor, vodu, oheň a další přírodní prvky.



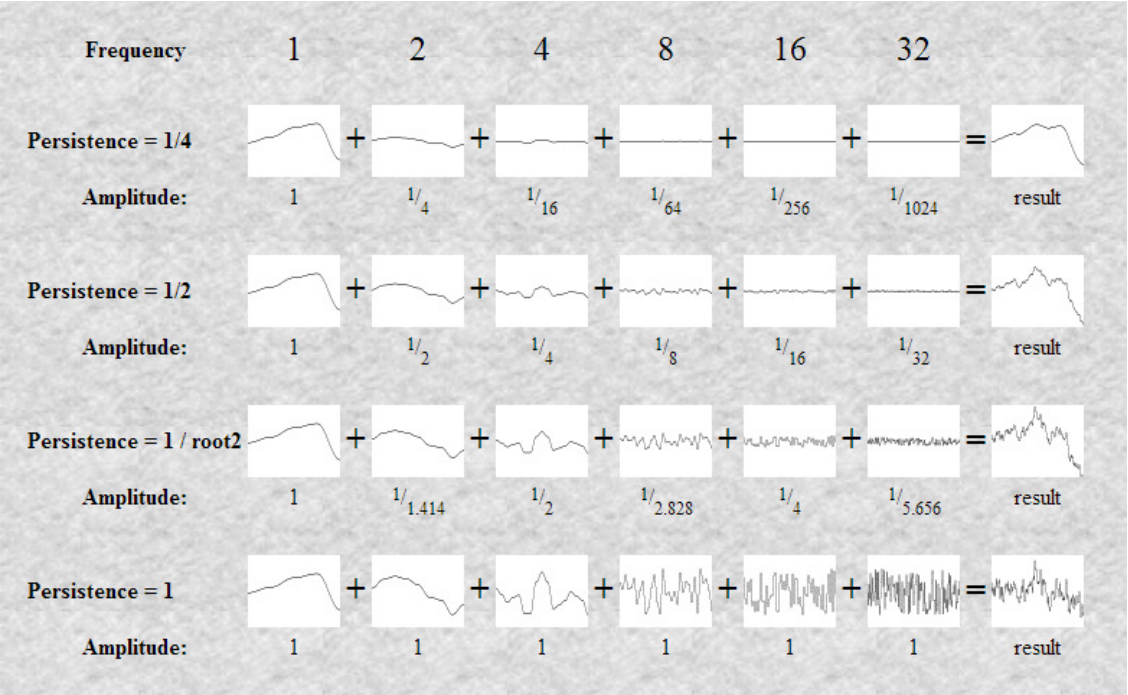
Obrázek 2.5: Několik 2D šumových funkcí, připravených na sloučení do výsledneho šumu (převzato z [3]).

Perlinův šum počítá každý pixel zvlášť, proto u velikých mnoho-dimenzionálních



Obrázek 2.6: Výsledný hladký šum po součtu funkcí z předchozích obrázků. V šumu lze zahlédnou tvary původních funkcí. (převzato z [3]).

aplikací může být výpočet velmi náročný. Základní náročnost algoritmu je $O(2^n)$. Při zvolení vhodných parametrů je zbytečné generovat mnoho oktáv, funkce se po několika oktávách měnit nebo naopak začne kmitat tak, že znehodnotí výsledek.



Obrázek 2.7: Vliv perzistence na generování amplitud oktáv (převzato z [3]).

Důležitým parametrem je i persistence. Ta zajišťuje rozmanitější průběh funkce. Ovlivňuje amplitudu a tak v později generováných oktávách dochází k ústupu nebo vzestupu kmitů s vysokou amplitudou.

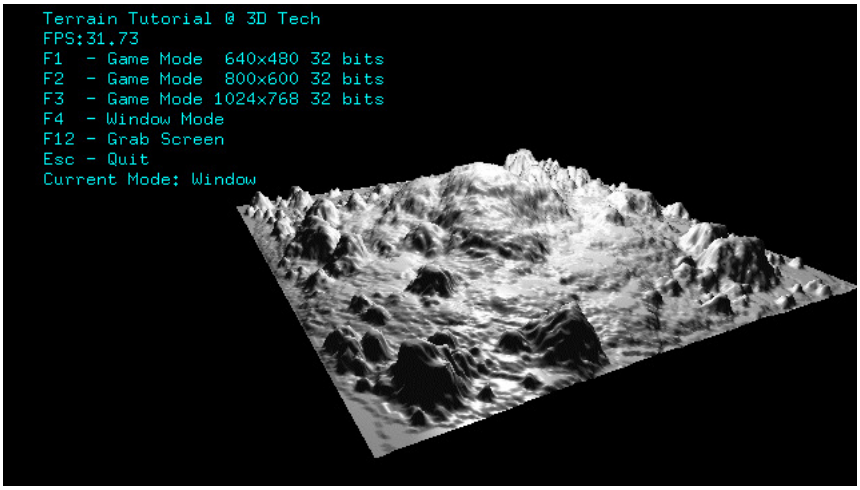
2.4 Skybox

V počítačové grafice je tak označována metoda vytvoření okolí scény, díky níž se docílí dojmu, že se zobrazená scéna jeví větší, než je její skutečná velikost. Myšlenka této metody je ve vykreslení vzdálených objektů, jako jsou hory, mraky, městské zástavby, okolního interiéru a podobně v závislosti na scéně. Tyto objekty nejsou složené z polygonů, ale jedná se pouze o texturu namapovanou na krychli nebo kvádr, který je vykreslený kolem scény. Skybox pak může být podle potřeby zvětšený nebo zmenšený.

Obrázek 2.8: Ukázka rozloženého skyboxu (převzato z [5]).

K namapování textury použijeme cube mapping. Textura pak může být předpřipravena nebo procedurálně generována jako v našem případě. Důležitu vlastností textur je jejich návaznost na hranách. Té lze docílit 3D generováním textur. Textury si lze předpřipravit nebo generovat pro každý snímek znova a tím tak měnit celou scénu.

Skybox se nejčastěji skládá ze 6ti textur, ale pro různé učely lze některé stěny negenerovat. Je to vhodé například u statické scény, či pro spodní stěnu krychle kterou překrývá terén. Dojde tím k ušetření výpočetních zdrojů, což je velice žádoucí.



Obrázek 2.9: 2D výšková mapa aplikovaná v 3D scéně (převzato z [2]).

2.5 Generování planet

Před vlastním generováním si je vhodné rozmyslet o jakou planetu půjde. Inspiraci v naší sluneční soustavě lze najít lehce. Planety zemského typu potřebují vygenerovat výškovou mapu, aby jejich povrch byl členitý a tuto mapu poté vhodně obarvit. Plynné planety lze generovat pomocí šumu podobně jako mraky nebo s víceméně jednobarevným povrchem (viz planety jako Uran a Neptun).

2.5.1 Výšková mapa

Jedná se o mapu výšek povrchu, která je s určitým stupněm hrubosti vyhlazena. Většina klasických scén je generována za pomoci 2D výškové mapy. Lze to provést s pomocí Perlinova šumu. Detailnější povrch je pak dále dotvořen například voxely.

Generování výškové mapy pro kulovitý objekt nám, ale přináší problém spojení hran 2D mapy. Není to nemožné, ale pro generování v rámci intra s omezenou velikostí je to nevhodné. Raději jsem se tedy zaměřil na 3D šum a vygenerovaní výškové mapy z něj.

Výšková mapa planety i asteroidu je generována pomocí 3D perlinova šumu. Transformace bodů jsou zaneseny do evaluačního shaderu a čistě kulatý objekt pak získá nepravidelný tvar. Tento postup by byl nevhodný pro generování větších detailů povrchu, pro takové detaily je vhodné aplikovat další deformace v rámci geometry shaderu s dotvořením dalších primitiv na povrchu.

2.5.2 Obarvení planet

Když je již výšková mapa hotová, musíme planetu ještě vhodně obarvit. Na volbu barev je vhodné použít stejnou funkci šumu se stejnými parametry jako při generování výškové mapy, která poslouží jako základ pro barevné přechody. Lze pak jednoduše rozmístit oceány a kontinenty na planetě podobné Zemi, přesně podle vygenerovaných prohlubní a výstupku.

Obarvení lze provéset vygenerováním 1D textury s barevným přechodem s vhodným rozložením barev. Při použití klasických funkcí jako `mix()` by výsledek povrchu byl neuspokojivý s minimálními přechody. Tato metoda by tedy vyhovovala pro tvorbu plynné

planety. 1D textura obsahuje přechod barev který je namapován na povrch podle funkce šumu z výškové mapy a tak může tvořit oceány, hory, břehy, pouště, zkrátka cokoliv.

Tento způsob generování barvy povrchu je velice jednoduchý a poskytuje uspokojující výsledky.

Kapitola 3

Implementace intra

V této kapitole se podíváme na vlastní implementaci této práce. Zmíníme použité metody, knihovny a způsob implementace.

3.1 Knihovny

Kromě standartních knihoven je nutno použít knihovnu s OpenGL, konkrétně GLEW. Mimo tu využívám knihovny pana Ing. Lukáše Poloka (používá ve zdrojových kódech přezdívku -tHE SWINE-) pro jednoduší ovládání OpenGL a operace s vektory a maticemi.

Pro vytvoření okna programu a procedury spojené se vstupem z klávesnice a práci s oknem používám pouze WinAPI, tedy základní funkce poskytované systémem Windows od Microsoftu.

GLEW - OpenGL Extension Wrangler

Multiplatformní knihovna psaná v pro C a C++, která pomáhá v v přístupu a ovládání OpenGL rozšíření. Umožňuje práci s grafickým řetězcem a efektivní ovládání požadovaných rozšíření.

3.2 Skybox

Vlastní skybox je generovaný před spuštěním animace celého intra a po dobu běhu se nemění. Jedná se tedy o statickou texturu. Ta je vygenerována do textury cubemapy pro jednoduší manipulaci a následné použití.

Skybox se skládá ze dvou prvků, tvoří ho hvězdy a mlhoviny.

3.2.1 Hvězdy

Základem pro vytvoření hvězdné oblohy je 3D Perlinův šum. Jelikož perlinův šum tvoří spojité textury mohl by se zdát nevhodný pro generování bodů v prostoru. Výsledný šum je tedy ořezán a hvězdy tvoří jen vrcholy amplitud šumu. Tento postup nazýváme prahování.

Šum pro vytvoření hvězd generuji následovně:

```
Noise(v*300+500,7,1.6,1.5,8)*.5+.5
```

Kde v je vektor vyjadřující pozici bodu v prostoru vůči počátku souřadného systému. Tento vektor je zvětšený a posunutý, protože perlinův šum je kvalitnější dále od středu souřadného

$$\begin{pmatrix} 0.01 & 0.08 & 0.01 \\ 0.08 & 0.64 & 0.08 \\ 0.01 & 0.08 & 0.01 \end{pmatrix} \tag{3.1}$$

Obrázek 3.1: Matice pro aplikované Gaussovo rozostření.

Obrázek 3.2: Ukázka generované hvězdné oblohy.

systému. Další parametry jsou frekvence, perzistence, amplituda a počet oktáv, v tomto pořadí. Výsledek vrácený touto funkcí je v intervalu $< -1; 1 >$. Jelikož funkci používám pro výpočet barvy, která je definovaná jen pro kladné hodnoty, musím výsledek posunou to správneho intervalu. Tedy interval vydělím dvěma a posunu o 0.5 do kladné části.

Dále aby hvězdy nevypadaly neuměle a místy hranatě je aplikován jednoduchý Gaussův filtr. Pomocí konvoluce tak dosáhneme mnohem lepšího výsledku. Viz zdrojový kód fragment shaderu 3.5

Pokud bychom počítali skybox pro každý snímek aminace znova, nešlo by použít mnou použitý postup, pro náročnost vykreslení. Gaussův filtr v každém texelu vyžaduje vygenerování okolních 8 bodů. To je výpočetně velice náročné, protože body jsou generovány pomocí Perlinova 3D šumu. Můj skybox je však generován staticky, před spuštěním samotné animace. Při prvním experimentu s filtrem jsem zaznamenal velkou počáteční prodlevu před startem animace. Textury, které generuji pro skybox mají veliké rozlišení a tak i pro výkonější grafické karty to není uplně snadný výpočet.

Po testech s gaussovým filtrem s maticí 3*3 jsem upustil od myšlenky použít větší, pro lepší kruhové rozmazání. Nároky na výpočet by byli neúměrné výsledku. Pro rozmazání by bylo vhodnější zvolit jiný postup generování textury.

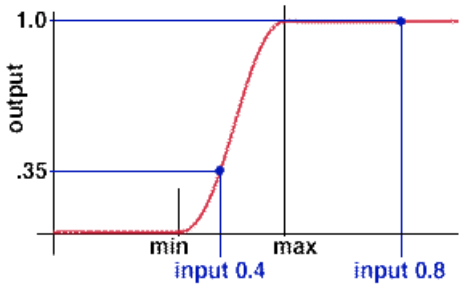
3.2.2 Mlhoviny

Aby nebyl prostor mezi hvězdami prázdný, je zde generováno množství mlhovin a mezihvězdných prachových mračen. Ty jsou generovány stejnou šumovou funkcí pouze s jinou frekvencí. Výsledný šum je obarvený s modrofialovým nádechem.

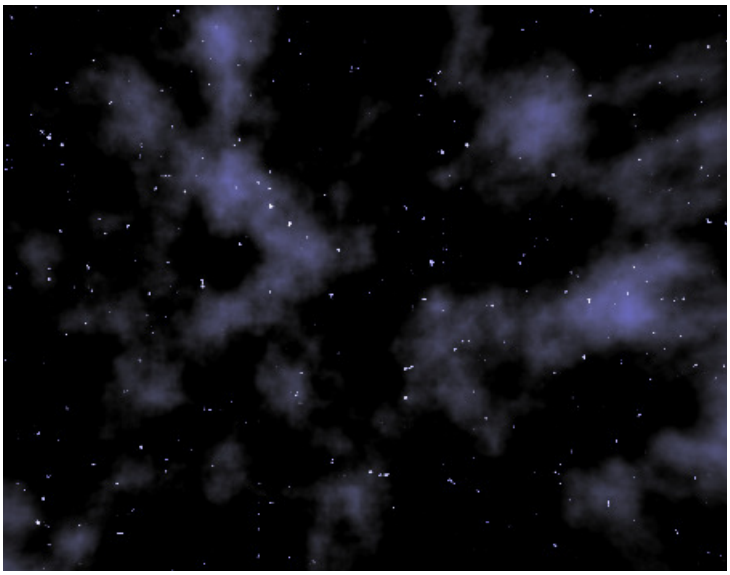
Postupného plynulého přechodu mezi barvami je docíleno pomocí funkcí `smoothstep` a `mix`, díky nímž lze udělat jednoduchý přechod barev jen na určeném intervalu.

```
smoothstep(0.3, 0.6, 0.4);

smoothstep(0.3, 0.6, 0.8);
```



Obrázek 3.3: Použití funkce smoothstep v praxi.



Obrázek 3.4: Ukázka generované hvězdné oblohy s mlhovinami.

Při obarvování jsou i některé hvězdy obarveny mlhou, vzniká tím tak lepší dojem z hvězdné oblohy.

3.3 Planeta

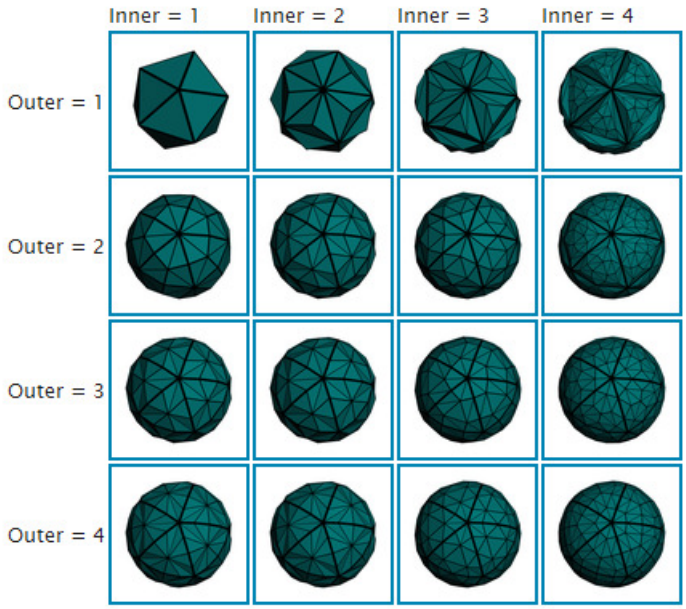
Pro vytvoření planety je nutné vygenerovat kouli. Základem pro generování mé planety je pravidelný dvacetistěn [4] (icosahedron). Z něj se postupně pomocí teselace generuje koule. Pro teselaci jsem zvolil experimentálně vnitřní i vnější parametr 12, protože při větším přiblížení bylo vidět značné zalomení povrchu planety. Bohužel parametr teselace ovlivňuje vypočetní náročnost a tak není vhodné volit čísla příliš vysoká.

Po experimentování s hladkým povrchem planety jsem dospěl k názoru, že povrch planety by měl být mírně deformován, aby reflektoval texturu povrchu, která zobrazuje hory a oceány. V evaluačním shaderu teselace byly vrcholy vynásobeny tak, aby se povrch zdeformoval přesně pod texturou. Docílil jsem toho tak, že jsem použil stejnou šumovou funkci jako pro generování textury, jen s nižší amplitudou výsledné hodnoty. Výsledek pak vytváří výškovou mapu v 3D.

Po několika experimentech bylo jasné, že přílišná výchylka povrchu planety nepůsobí

```
#version 400 // zde je vložen další kód pro perlinův 3D šum
in vec3 pos1;
uniform int face; // označení stěny krychle
layout(location = 0) out vec4 color;
void main(){
  float col = 0.0f;
  vec3 vector;
  // matice rozostření
  mat3 blur = mat3(vec3(0.01,0.08,0.01),vec3(.08,.64,.08),vec3(.01,.08,.01));
  // výpočet barvy bodu je složen z okolních osmi bodů s aplikací Gaussovy matice
  for(int i = -1; i<=1; i++)
    for(int j = -1; j<=1; j++){
      float c;
      vec3 v = vec3(0,0,0);
      vec2 coord = gl_FragCoord.xy;
      coord.x += i; coord.y+= j; // posunutí v matici
      //výběr stěny skyboxu a její otočení
      if(face == 0) v = vec3(-1,(coord.yx/vec2(1920,1920)*2-1)*vec2(1,-1));
      if(face == 1) v = vec3(1,(coord.yx/vec2(1920,1920)*2-1));
      if(face == 2) v = vec3((coord.x/1920.*2-1)*-1, -1, (coord.y/1920.*2-1)*1);
      if(face == 3) v = vec3((coord.x/1920.*2-1)*-1, 1, (coord.y/1920.*2-1)*-1);
      if(face == 4) v = vec3((coord.x/1920.*2-1)*-1,(coord.y/1920.*2-1),1);
      if(face == 5) v = vec3(((coord.xy/vec2(1920,1920))*2-1),-1);
      if(i == 0 && j == 0) vector = v;
      c = Noise(v*300+500,7,1.6,1.5,8)*.5+.5;
      //prahování a aplikace rozostření
      col += (c>.8) ? c*blur[i+1][j+1] : 0;
  }
  vector = normalize(vector); // pozice aktuálního bodu pro generování šumu
  float col2 = Noise(vector*300+500,1,1.7,1,8);
//výsledná barva texelu se skládá zde
  color = vec4(col,col,col,1.0) + //základní hvězdy bílé, rozostřené
0.5*mix(vec4(col2),vec4(32/255,100/255,1,0.5),smoothstep(0.0, 1., col2)) +
//mlhoviny
0.5*mix(vec4(col2),vec4(32/255,36/255,1,1),smoothstep(0.0, 1.0, col));
//dobarvení mlhovin a některých hvězd
}
```

Obrázek 3.5: Ukázka fragment shaderu pro generování stěn skyboxu s dodaným komentářem.



Obrázek 3.6: Ukázka vlivu teselačních konstant na výsledný objekt. (převzato z [4]).

dobře. Na první pohled z větší vzdálenosti se vysledek může zdát dokonale hladký, při bližším pohledu je ale patrná mírná deformace.

Planeta je generována jako terestrická planeta podobná Zemi. Na jejím povrchnu jsou opět pomocí šumu generovány kontinenty a oceány za použití výškové mapy. Kraje oceánu a země jsou tvořeny barevným přechodem (inspirováno z [1], stejně tak vrcholy hor. Severní a jižní pól planety je zabarven více do bíla a tvoří tak dojem polárních oblastí.

Planeta je osvícena nedalekou hvězdou, jedna její polovina je tak temné a druhá světlá. Povrch planety v místech, kde je vodní hladina, vytváří větší odlesk (spekulární složka světla) než povrch, kde je pevnina. Rotace je zajištěna transformací modelové matice.

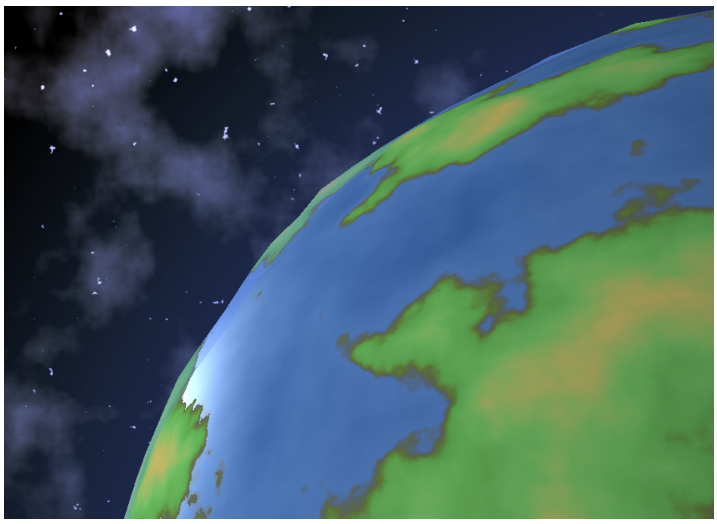
Kolem planety se generuje mírně namodralá atmosféra. Barva je zvolena vzhledem k pozadí, na kterém by bíla průhledná textura zanikala. Atmosféra je generována na čtverec, který je pořád natočený směrem ke kameře (viz billboarding). Je zde uplatněn blending, aby atmosféra působila průhledně vzhledem k hvězdnému pozadí. Pokud bychom ji měli přirovnat k reálné atmosféře naší planety, jednalo by se o nejvyšší vrstvy velmi řídkého vzduchu.

V průběhu animace se planeta promění. Po nárazu asteroidu, který má značnou velikost, se celý povrch planety přemění na roztavené horniny.

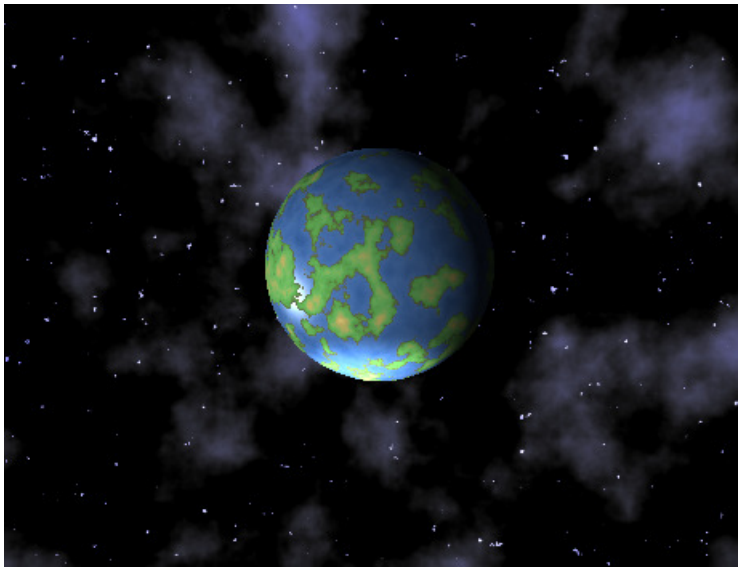
3.4 Asteroid

Stejně jako u planety, je základem pro generování asteroidu koule respektive dvacetistěn. Při teselaci je ale zavedena mnohem hrubší transformace povrchu. Vzhled asteroidu je podobný krystalům ledu s prachem usazeným na povrchu.

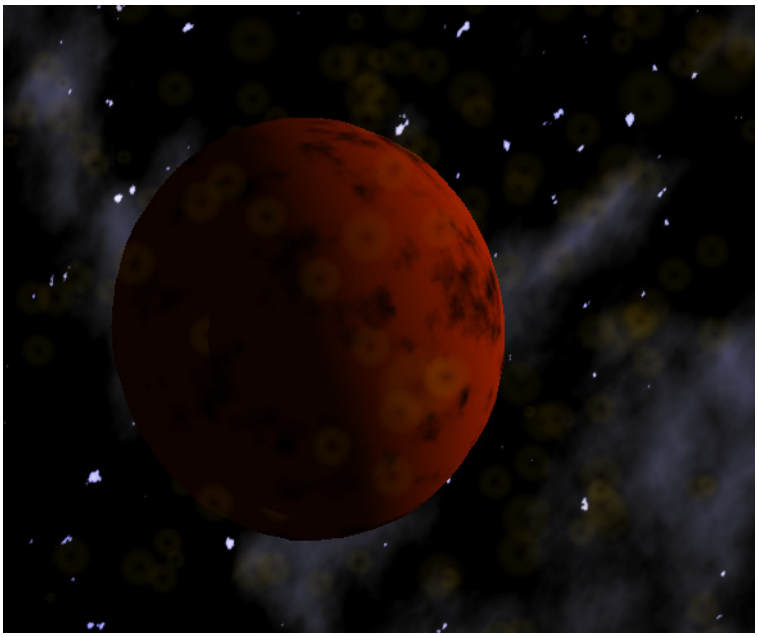
Pohyb asteroidu je v kolizním kurzu s planetou a je řízen globálním časem. V jistém okamžiku, kdy narazí do planety, je spuštěna animace částícového systému a po zanoření



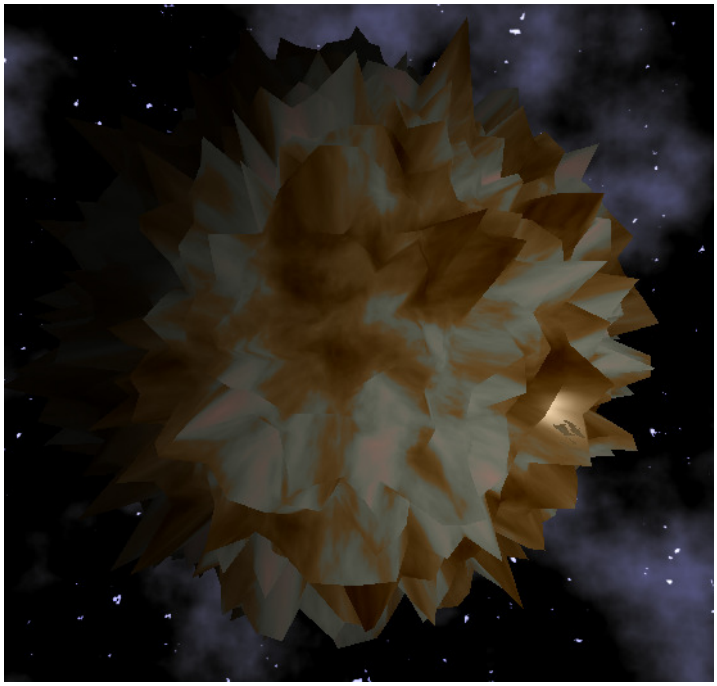
Obrázek 3.7: Detail nerovného povrchu planety.



Obrázek 3.8: Ukázka vygenerované terestrické planety.



Obrázek 3.9: Planeta po nárazu asteroidu.



Obrázek 3.10: Vygenerovaný asteroid.

do planety, tedy po tom co zmizí ze zobrazené scény, je vypuštěno i jeho vykreslování v rámcí intra.

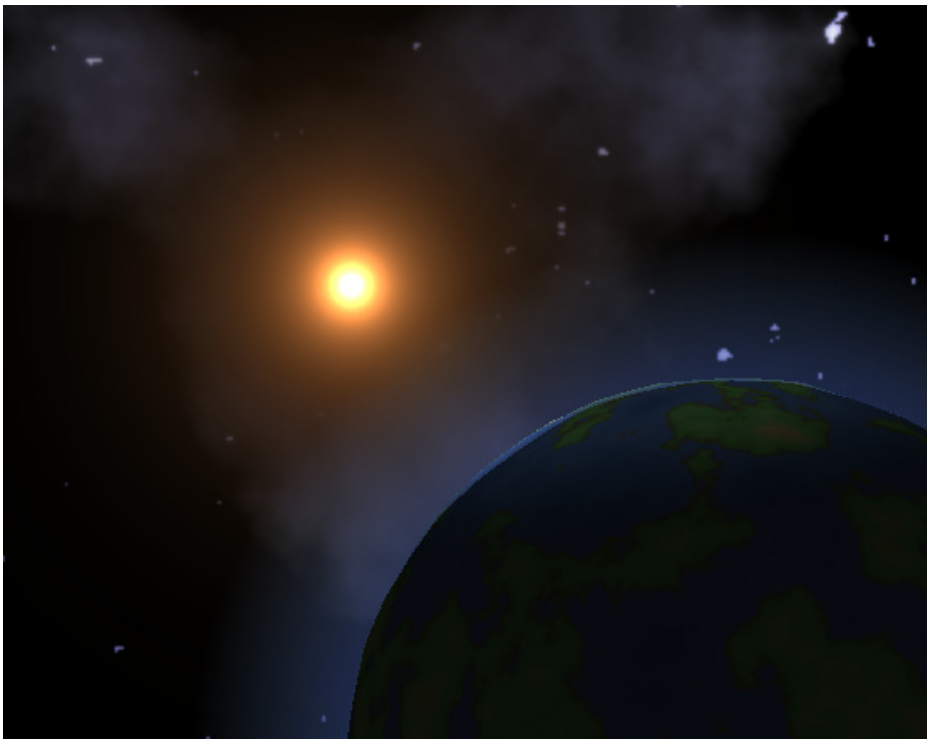
3.5 Slunce

V několika záběrech kamery se objevuje i blízká hvězda. Ta je umístěna na pozici světelného zdroje a dotvaří dojem osvětlené scény. Pozice hvězdy je po celou dobu animace neměnná. Je definována vektorem:

```
Vector4f sun(-50.0f,-1.0f,-1.0f,1.0f);
```

O vykreslení se stará jednoduchá sada vertex a fragment shaderu. Do fragment shaderu jsou zadány souřadnice pro dva trojúhelníky. Ty vytvoří čtverec, na který potom vykresluji texturu hvězdy. Pozice hvězdy je přenásobena View maticí tak, aby se nacházela přesně v místě zdroje světla a pomocí projekční matice je docíleno billboardingu - hvězda je tedy 2D textura natočená stále směrem ke kameře.

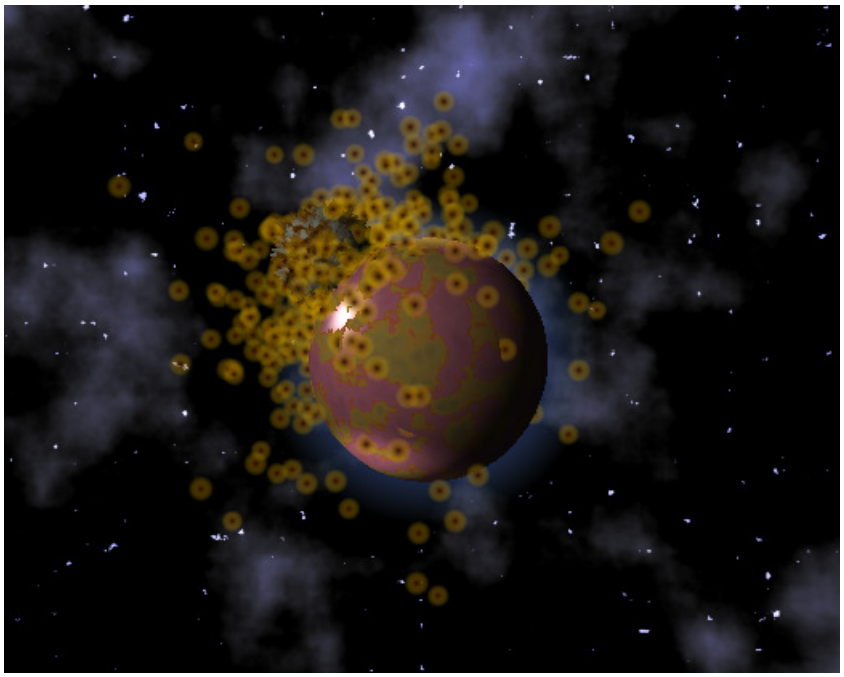
Textura hvězdy je složena ze žhavého kotouče a okolní koróny. Koróny je docíleno pomocí blendingu textury do ztracena. Pomocí přechodu je namapována barva, které určuje průhlednost vektor vzdálenosti od středu hvězdy. Žhavý kotouč je tvořen přechodem žluté barvy se zarudlým okrajem.



Obrázek 3.11: Vygenerovaná hvězdy v pozadí planety.

3.6 Částicový systém

Tento efekt je vykreslen až od času události srážky asteroidu a po uplynutí krátké doby zmizí. Částice jsou generovány z místa nárazu asteroidu do planety a pomocí billboardingu tvoří 3D dojem. Částice jsou generovány pomocí Perlinova šumu, kde jako vstupní parametr



Obrázek 3.12: Částicový systém.

je ID spuštění vertex shaderu s posunem. Díky tomu je pro každou částici generován jiný směrový vektor a ty se tedy rozletí do různých stran.

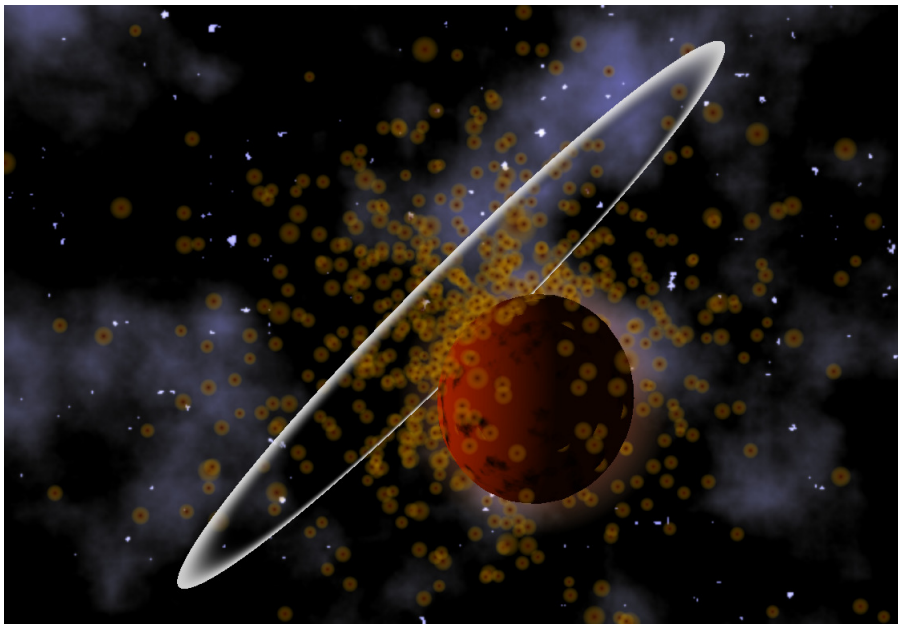
Částicový systém jsem doplnil rázovou vlnou šířící se z místa dopadu asteroidu do okolí. Vlna je tvořena obdobně jako slunce, nanesena na zvětšující se čtverec, umístěný na tečně mezi střede asteroidu a planety. Barvy vlny je generována obdobně jako slunce, s tím rozdílem, že barva je nanesena od ostrých krajů a směrem do středu je barva zprůhledněna.

3.7 Animace

Animace intra se skládá z několika záběrů kamery, rotace objektů, nárazu a výbuchu. Kamera se přemisťuje podle pohybu objektů v závislosti na aktulní části animace. Experimentálně jsem hledal vhodnou polohu kamery a její pohyb tak, aby animace působila zajímavě a byla zobrazena ze všech možných ůhlú.

Nastavení kamery provádím pomocí `View matice`, kterou si pomocí knihovny nastavím tak, aby se chovala jako OpenGL funkce `LookAt`. Požadovanou funkci mi poskytuje knihovna na transformaci matic a vypadá takto:

```
CGLTransform::LookAt(View,
   posX, posY, posZ,
   lookatX,lookatY,lookatZ,
```



Obrázek 3.13: Rázová vlna po nárazu asteroidu.

```
upX,upY,upZ);
```

Jde vlastně o tři vektory - pozice kamery, pozice bodu, na který hledíme a orientace kamery tzv. `up vektor`. Výsledek je poté vložen do View matice, která je prvním parametrem funkce. Tuto matici pak předávám všem objektům a upravenou matici používám pro natočení skyboxu.

Pohyb kamery a objektů na scéně je řízen modelovým časem. V animaci používám dva druhy času. Jeden je čistě modelový a jsou pomocí něj vykreslovány kroky animace, pohyb kamery, částicový efekt. Tento čas nesouvisí s reálným časem.

Druhý čas je počítán podle obrázků za sekundu (fps). Následuje přepočet na modelový čas a poté vykreslení simulace. Volitelně tedy mohu v rámci zdrojového kódu nastavit, jak rychle se budou promítat animace a to bez změny modelového času. Pokud by byl použit pouze modelový čas bez reflexe reálného výpočtu, animace by běžela zpomaleně nebo zrychleně podle výpočetního výkonu počítače.

Kapitola 4

Spustitelný soubor

Vývoj mého intra probíhal pouze na operačním systému Windows a proto je výsledkem spustitelný soubor právě pro tento systém (testováno na Windows 8 64-bit). Dalším požadavkem mohou být specifické parametry grafické karty. Teselační shadery, které používám, jsou dostupné až u grafických karet s OpenGL v4.0 a vyšší. Aplikace má toto ošetřené v ovladači grafiky a pokud požadavek není splněn, je to oznámeno uživateli v kontextovém okně. U slabších počítačů může být běh aplikace zpomalený a její start může trvat několik sekund, kvůli generování grafiky.

Program lze z konzole spustit se dvěma parametry. Jeden pro zapnutí fullscreen (–full) a druhý pro výpis FPS do konzole (–fps). Na jejich pořadí nezáleží.

```
Intro.exe --full --fps
```

4.1 Omezená velikost a kkrunchy

Grafická intra omezená prostorem mají velký problém, protože i za použití různých optimalizací často přesahují požadovanou velikost. To je způsobeno také standartními překladači, protože ty nejsou navrženy pro takovou funkci a musí se chovat korektně pro veškeré programy. Intra většinou obsahují specifický kód a proto by šlo některé náležitosti zanedbat a vynechat tak, aby velikost spustitelného souboru byla mensší. Naštěstí komunita kolem této tvorby je dostatečně široká a tak existují různé programy na komprimaci výsledného spustitelného souboru.

Německá kupina farbrauch, respektive její člen, naprogramoval malý komprimační prográmek kkrunchy, který dokáže zmenšit spustitelný soubor. Kromě zmenšení umí určit, o kolik je náš kód větší neř požadovaná referenční velikost.

Použití:

```
kkrunchy_k7.exe --best --refsize 64 --out cIntro.exe Intro.exe      (4.1)
```

Kde `--best` je parametr pro nejlepší způsob komprimace, `--refsize 64` je referenční velikost požadovaného výsledného souboru, my cílime na 64kB, `--out cIntro.exe` je jméno výstupního souboru a poslední parametr je název souboru ke komprimaci.

kkrunchy je určený jen pro 32-bitové aplikace a používajího i ostatní velké skupiny, které vytváří grafická intra různé velikosti.

Ukázka výstupu programu kkrunchy použítého na mé intro. Výstup byl zmenšen přibližně na třetinu velikosti:

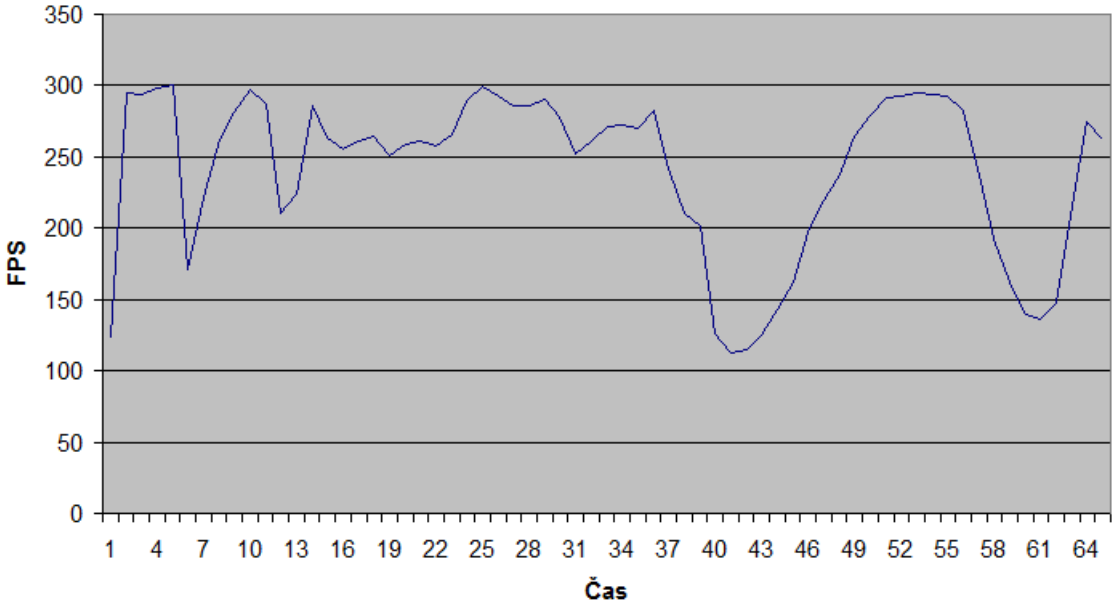
```
>kkrunchy_k7.exe --best --refsize 64 --out cIntro.exe Intro.exe
kkrunchy 0.23 alpha 2 >> radical exe packer (c) f. giesen 2003-2006

 - no symbol info present
 - preprocessing, filtering & reslicing
 - packing [################################] => 8056 bytes (in 0.03s)
 - WARNING: Manifest present and stored uncompressed.
 - WARNING: Resources present, this decreases compression a bit
 - WARNING: Out ImageBase 0x3e0000 lower than 0x400000, won't run under Win9x
 - writing output file cIntro.exe
 - packed executable 12288 -> 10752 bytes
 - delta to reference size: -55180 bytes
```

Pro rychlé vygenerování komprimované verze je přiložen spustitelný skpript `kkrunchy-compress.bat`, který obsahuje výše zmíněné parametry.

4.2 Experiment

Na hotové práci jsem chtěl otestovat závislost FPS na průběhu jedné periody animačního cyklu. Hodnoty FPS jsou velice vysoké, což značí, že bych mohl zvyšovat dále výpočetní náročnost. Pro ilustraci náročnosti výpočtu částicového systému a zrychlění běhu simulace následuje průběh FPS hodnost. Test byl proveden s parametem –full na notebooku MSI GE60 s grafickou kartou NVidia GTX 660M, rozlišení scény bylo 1920 * 1080 bodů.



Obrázek 4.1: Graf závislosti FPS na čase animace.

Začátek intra je ovlivněn generováním statických textur pro skybox. K největšímu poklesu FPS došlo při nárazu asteroidu do planety v čase okolo 40 sekundy a poté při zrychleném vracení se časem v závěru smyčky.

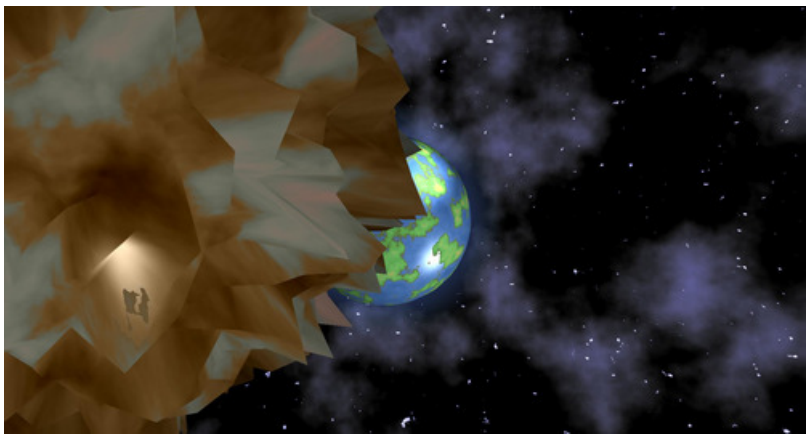
Kapitola 5

Závěr

Cílem práce bylo vytvořit grafické intro s omezenou velikostí. Pro účely demonstrace možností OpenGL jsem si vybral prostředí vesmíru a vymodeloval jsem jednoduchou kolizi planety s velkým asteroidem. Než jsem začal pracovat na této práci, měl jsem o tvorbě grafických aplikací a modelování velmi málo informací.

Při vytváření práce jsem se tedy seznámil s možnostmi OpenGL, s procedurálním generováním, použitím šumu v generované grafice. Dále jsem zkoumal další možnosti, které jsem nakonec nevyužil (ať již kuli složitosti implementace nebo špatné možnosti zapomponování do mojí scény) jako metody stínování, L-systémy či volumetrické generování.

Intro slouží jako základní ukázka práce s OpenGL a GLSL. Dalo by se dále rozšiřovat a upravovat. Lákavá je myšlenka vygenerování sluneční soustavy a průlet skrz ní až k místu kolize. Pro tento účel by se muselo dynamicky měnit množství generovaných detailů. Také by bylo vhodné doplnit intro o stínování a zvukový doprovod. K tomu by bylo zapotřebí mnohem více času, který jsem bohužel neměl a tak jsem se zaměřil na základní možnosti generování grafiky s omezenou velikostí.



Obrázek 5.1: Ukázka začátku intra.

Literatura

[1] GLSL Sandbox [online]. http://glsl.heroku.com/e#7662.3, [cit 2014-05-11].

[2] OpenGL tutorial - Lighthouse3D. http://www.lighthouse3d.com/opengl/, [cit 2014-05-11].

[3] Perlin noise. [online]. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm, [cit 2014-05-11].

[4] Triangle Tessellation with OpenGL 4.0 [online]. http://prideout.net/blog/?p=48, [cit 2014-05-11].

[5] Tutorial 25 - Skybox. http://ogldev.atspace.co.uk/www/tutorial25/tutorial25.html, [cit 2014-05-11].

[6] Catmul, E.; Clark, J.: *Recursively generated B-spline surfaces on arbitrary topological meshes*. 1978, doi:10.1016/0010-4485(78)90110-0.

[7] Mandelbrot, B. B.: *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1982, ISBN 0-7167-1186-9.

[8] Perlin, K.: Makeing noise. [online]. http://www.noisemachine.com/talk1/, 1999 [cit 2014-05-11].

[9] SEGAL, M.; AKELEY, K.: *The OpenGL Graphics System: A Specification (Version 4.3 (Core Profile))*. 2012.

[10] Tišnovský, P.: Phongův osvětlovací model. [online]. http://www.root.cz/clanky/opengl-19-phonguv-osvetlovaci-model/, [cit 2014-05-11].

Příloha A

Plakát