VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

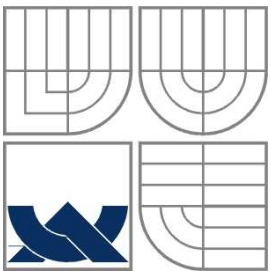
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ DEMO V 128KB

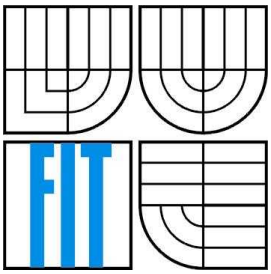
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN ZÁLETA
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ DEMO V 128KB

GRAPHICS DEMO IN 128KB

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN ZÁLETA
AUTHOR

VEDOUCÍ PRÁCE — ING. JURAJ VANEK
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce pojednává o klíčových prvcích v procesu tvorby jednoduchých 3D grafických animací (dem) s hlavním důrazem na minimalizaci množstva externích zdrojů jako dynamických knihoven, povrchů a modelů. Výsledná aplikace je složena z okna vytvořeného s pomocí WinAPI, které je naplněno plně funkční scénou, která využívá rozhraní OpenGL a OpenGL Shading Language pro kontrolu nad zobrazováním celé scény, její osvětlením a povrchy.

Abstract

This paper describes the key elements in the process of making simple 3D graphic animations (demos) with the main focus on minimalizing the amount of external resources such as dynamic libraries, textures and models. The resulting application is comprised of a window created using WinAPI, which is filled with a fully operational scene, that uses the environment of OpenGL and OpenGL Shading Language for the control over displaying the whole scene, its lightning and textures.

Klíčová slova

demo, 128kB, WinAPI, OpenGL, GLSL, VBO

Keywords

demo, 128kB, WinAPI, OpenGL, GLSL, VBO

Citace

Martin Záleta: Grafické demo v 128kB, bakalářská práce, Brno, FIT VUT v Brně, 2011

Grafické demo v 128kB

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval v souladu s respektováním intelektuálního vlastnictví a tedy zachoval autenticitu bez sklonů k plagiaritě, k čemu jsem přispěl uvedením všech pramenů a zdrojů, ze kterých jsem čerpal.

……
Martin Záleta
16. 5. 2011

Poděkování

Vzhledem k nedocenitelnému přínosu mého vedoucího jménem Juraj Vanek považuji za vhodné v těchle řádcích vznést poděkovaní za jeho odbornou pomoc a trpělivost ohledem práce. Taky bych chtěl rovným dílem poděkovat svým rodičům za vhodné prostředí k umožnení tohohle počinu.

Obsah

1 Úvod

Slovo *„demo“* si v oblasti počítačovej grafiky nadobudlo pomerne veľkú tradíciu, nakoľko sa už pri zrode tejto problematiky jednalo o jednu z prvých vecí, kde sa aj skutočne aplikovala vo svojom plnom rozsahu. Jednalo sa o krátke, obvykle niekoľkominútové animácie zvyčajne reprezentujúce správy poskytované šíriteľmi nelicencovaného softvéru alebo jednoducho iba inými nezávislými programátormi pohybujúcimi sa v tejto sfére bez akéhokoľvek pochybného pozadia alebo podtextu.

Oveľa dôležitejšou skutočnosťou však je, že už ako ich názov napovedá, tak veľmi často bolo už v ich samotnej podstate akýmsi spôsobom *demonštrovať* možnosti vtedajších technológií v oblasti počítačovej grafiky a predviesť tak rôzne funkcie a efekty z ich dispozície. Často sa dokonca jednalo o také, ktorých prevedenie bývalo pre širšiu oblasť programátorov nedosiahnuteľným a k bežnému používaniu a dostupnosti z hľadiska informácií sa dostalo až v neskoršej dobe.

Činnosť tvorenia týchto *dem* však stále ako kreatívna činnosť pretrváva, no v súčasnosti sa ich návrh a prevedenie viac blíži k moderným postupom v počítačovej grafike. Prevažne teda zahŕňa tvorbu modelov a povrchov vonkajšími zdrojmi a v samotnom kóde už rieši iba prepojenie a dynamiku týchto prvkov. Jedná sa o postup, aký ešte pred desiatkou rokov nebol možný a všetky podobné prvky sa museli robiť ručne. *

Hlavným úsilím v mojom programátorskom počine bola teda tvorba aplikácie, ktorej myšlienkou bolo predviesť a jasne znázorniť niektoré základné stavebné prvky poskytované možnosťami súčasných grafických knižníc a prostredí.

Celkový konečný vzhľad aplikácie som si rozdelil do niekoľkých tematicky odlišných funkčných celkov – scén, pričom som sa v nich usiloval poukázať na jednotlivé prístupy v tvorbe 3D animácií, konkrétne zavádzanie osvetľovacieho modelu v jazyku GLSL, procedurálneho generovania povrchov, využitia Vertex Buffer Objektov a spline kriviek pri modelovaní objektov, a iných. Počas ich tvorby som sa s nimi teda aj snažil dôkladne oboznámiť a získať tak i správne podklady pre budúcu prácu v tejto oblasti.

Pri návrhu aj implementácii tejto aplikácie som taktiež musel neustále myslieť aj na zadaním dané veľkostné obmedzenia, čo už do veľkosti, tak i do počtu výsledných súborov, takže som musel vypustiť niektoré dostupné možnosti využitia zložitých objektov generovaných ako výstupy moderných 3D modelovacích aplikácií, alebo dokonca aj len grafické súbory a výškové mapy rôznych externe modelovaných povrchov. Toto ma silno motivovalo k tomu, aby som vo výslednej podobe aplikácie ukázal, že aj za použitia pomerne jednoduchých prvkov je možné docieliť bohatého audiovizuálneho výstupu. Inými slovami istým spôsobom vyzdvihnúť myšlienku, že výsledný efekt je v oblasti multimédií dôležitejší než zložitosť použitých postupov.

Svoju prácu som aj dôkladne formálne rozpracoval ako v tomto dokumente, tak aj v programovej dokumentácii tvorenej komentármi zdrojového kódu. Logicky som túto technickú správu rozdelil podľa jednotlivých fáz vývoja aplikácie do častí o jej analýze, návrhu a implementácii.

Analýze prináleží kapitola 2, v ktorej sú dôkladne rozpracované využívané vývojové prostriedky, grafické rozhrania a ich možnosti. Opísané tu je ako rozhranie OpenGL, tak aj jazyk OpenGL Shading Language a jeho využitie pri tvorbe tejto aplikácie. Ohľad sa tu takisto berie aj na možnosti využitia jednotlivých vykresľovacích techník a ich výkonu, a nie je tu opomenutá ani úloha daných veľkostných obmedzení.

Kapitola 3 je venovaná takpovediac návrhu aplikácie, teda všetkým postupom týkajúcich sa problematiky tvorby objektov, osvetlenia, povrchov, modelovacích efektov atď. Hlavná pozornosť tu prináleží hlavne takým, ktoré sú v mojej aplikácii aj skutočne použité, teda konkrétne ich prevedeniu a zhodnoteniu v rámci výkonu, no viacero riadkov je venovaných aj možným zlepšeniam a ukážkam pokročilých techník, ktoré ostali v mojom projekte nezrealizované.

---

* Viac o *demách* a ich tvorbe je možné sa dozvedieť na portáli [6].

Napokon samotná implementácia a výstup práce sú objasnené v kapitole 4. Sú tu dôkladne popísané jednotlivé funkčné časti programu a zverejnené aj ukážky grafických výstupov. Na konci kapitoly sú taktiež použité metódy zhodnotené na základe ich praktických testov.

Všetky uvedené kapitoly sú navyše doložené množstvom súborov súvisiacich s týmto projektom na CD nosiči a súčasťou je tiež plagát poskytujúci zhrnutie výsledkov tohto projektu.

2 Analýza

Účelom tejto práce je vytvoriť grafické demo s ohľadom na jeho obmedzenú veľkosť. Z toho sa odvíjajú i bezpodmienečné požiadavky jej výstupov, menovite výsledná veľkosť pod 128kB, dĺžka aspoň 3 minúty a prítomná hudba v pozadí. V konečnej podobe má demo byť tvorené jediným exe súborom spustiteľným v čistej inštalácii Windows XP a teda nemusí prísne dodržiavať podmienku prenositeľnosti na iné platformy. Samotné prevedenie musí byť uskutočnené v jazyku C/C++ za pomoci knižnice OpenGL, prípadne Direct-X, kvôli ich možnostiam pre real-time zobrazenie grafických prvkov. Nasledujúcich pár riadkov bude teda venovaných tomu, čo tieto podmienky pri návrhu aplikácie z implementačných hľadísk znamenajú, čiže čo všetko je nutne potrebné dodržiavať a čomu je naopak potreba sa vyhnúť.

2.1 Vývojové prostriedky

Pri potrebe vývoja aplikácií v jazyku C/C++ v prostredí systémov Microsoft Windows sa naskytuje možnosť využiť IDE priamo určené na túto úlohu, a teda Microsoft Visual Studio, avšak dostupné sú aj isté alternatívne možnosti ako napr. PSPad a vim online, ktoré však nie je potrebné hlbšie skúmať a pre túto konkrétnu aplikáciu bude stačiť jedna z novších verzií prvej z navrhovaných možností – Microsoft Visual Studio 2008 Professional Edition.

Pre potreby riešenia implementačných detailov je však oveľa dôležitejšie sa rozhodnúť na spôsobe tvorby samotného okna určeného na vykresľovanie grafických prvkov. K tomuto účelu existuje viacero programátorských postupov, menovite Windows API (WinAPI), OpenGL Utility Toolkit (GLUT) a Software Development Kit (SDK), avšak iba prvý z menovaných nesie výhodu nezávislosti na vonkajších dynamicky prepojovaných knižniciach, ktoré nie sú štandardnou súčasťou Windows OS, a je okamžite dostupný aj v každej čistej inštalácii systému Windows XP a vyššie.

Rozhranie WinAPI teda nielenže je pre tu požadovanú aplikáciu najlepšou vhodnou možnosťou, ale umožňuje i tvorbu dialógového okna na výber rozlíšenia, čo je tiež z hľadiska návrhu k veľkému úžitku.

2.2 Rozhranie OpenGL

Na základe predchádzajúcich skúseností som sa rozhodol pre OpenGL a skúmanie iba ním poskytovaných možností, teda využitie Direct-X na vykresľovanie grafických prvkov nezvažujem. Toto prostredie má k dispozícii radu príkazov určených na ovládanie funkcií grafickej karty za účelom zobrazenia požadovaných výstupov. Na oboznámenie s nimi slúži okrem oficiálnej špecifikácie aj veľké množstvo poučných kníh, ako napríklad [2]. V svojej aplikácii používam predovšetkým tie, ktoré sú stručne vysvetlené v následujúcej tabuľke:

Funckia	Popis
glEnable() / glDisable()	Zapínanie a vypínanie stavových premenných prostredia.
glPushAttrib() / glPopAttrib()	Výber a vkladanie aktuálnych stavových premenných na zásobník.
glViewPort()	Prispôsobenie veľkosti zobrazovacieho priestoru.
glClear()	Vyprázdni určené buffery.
glMatrixMode()	Voľba modifikovanej matice (modelová / projekčná / textúrová).
glLoadIdentity()	Nastavenie súčasne zvolenej matice na jednotkovú.
glTranslatef()	Transformácia posunu matice daná vektorom.
glRotatef()	Transformácia rotácie matice daná počtom stupňov a vektorom.
glScalef()	Transformácia zmeny mierky matice.
glPushMatrix() / glPopMatrix()	Výber a vkladanie aktuálne modifikovanej matice na zásobník.
glBegin() / glEnd()	Zapínanie a vypínanie okamžitého módu vykresľovania.

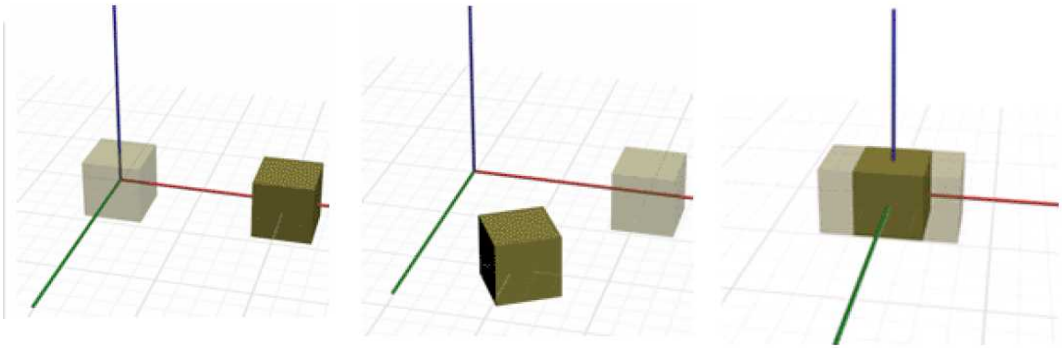
glFlush()	Vynútenie vykonania všetkých doterajších OpenGL príkazov.

Tabuľka 1: Základné funkcie rozhrania OpenGL.

Pred samotnou prácou v tomto rozhraní je však dôležité si objasniť, ako vlastne poskytnuté vykresľovanie 3D objektov a scén funguje, a na čo treba dbať pri používaní jednotlivých prístupov. Práve tejto problematike je venovaných nasledujúcich pár podkapitol.

2.2.1 Umiestňovanie objektov do scény

Pri vykresľovaní objektov je aktuálna pozícia určená pomocou modelovej matice, takže akýkoľvek pohyb v scéne sa musí uskutočňovať prostredníctvom transformácií tejto matice. Tie sa v prostredí OpenGL vykonávajú sadou príkazov, ktoré na práve aktívnej matici vykonávajú transformácie posunov, rotácií, či zmien mierky.



Obrázok 1: Vizuálne znázornenie trasformácií posunu, rotácie a zmeny mierky [1].

Takisto je dôležité poznamenať, že v prostredí OpenGL sa pracuje s práve jednou modelovou maticou, takže na spätný návrat vykonaných zmien je nutné používať príkazy na vloženie aktuálnej matice na zásobník a na výber matice z vrchu zásobníka.

2.2.2 Okamžitý mód (Immediate mode)

Najviac priamočiarym spôsobom vykresľovania objektov v prostredí OpenGL je tzv. „okamžitý mód“ (immediate mode), ktorý spôsobuje, že všetky v ňom dané príkazy sa vykonajú okamžite, čo z implementačného hľadiska znamená, že je vtedy nutné postupne špecifikovať pozíciu každého jedného z vrcholov práve vykresľovaného objektu a všetkých potrebných atribútov ako farba, normálové vektory, textúrové koordináty a iné. Pri vykresľovaní zložitejších objektov je teda takýto prístup veľmi náročný na počet vykonávaných operácií a teda i výpočtový čas. Dokonca v súčasnom vývoji OpenGL panuje úsilie takéto vykresľovanie úplne zamedziť a prenechať ho čisto do rúk Vertex Buffer Objektov, ktoré popíšem neskôr.

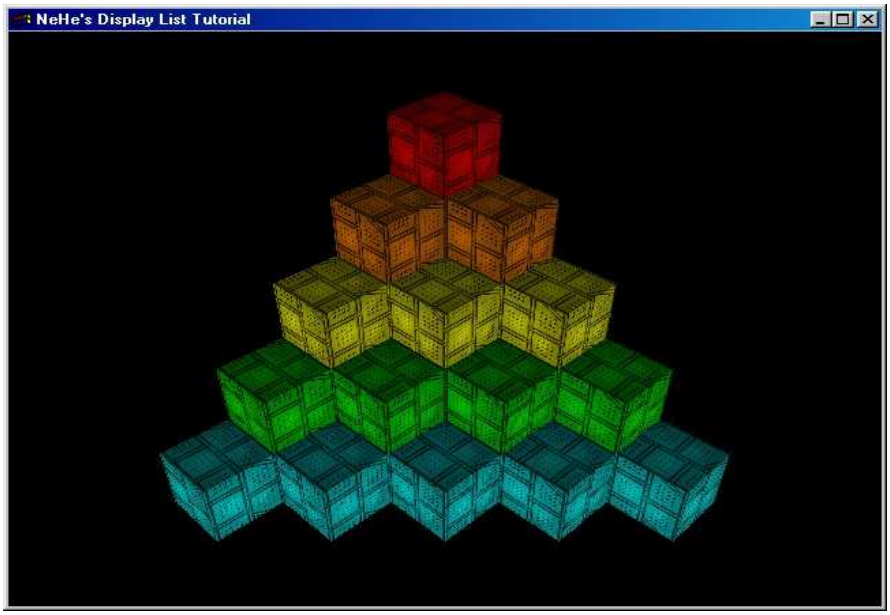
V tomto projekte však považujem za dôležité tieto tvrdenia aj prakticky overiť a hlavne odskúšať, či aj pri animáciách s menším množstvom objektov než by boli v iných, rozsiahlejších scénach a grafických projektoch, znamená spomínané množstvo a výpočtová náročnosť operácií vážnu hrozbu pre celkový výkon programu.

2.2.3 Display list

Display listy sú v OpenGL spôsobom, ktorý oproti okamžitému módu uľahčuje náročnosť výpočtov potrebných pri vykresľovaní objektov tým, že danú postupnosť príkazov po ich prepočítaní vzhľadom k nastaveným príznakom prostredia a maticovým transformáciám ukladá do video pamäti (VRAM)

namiesto toho, aby ich hneď vykreslil. Za behu programu už potom stačí len takto vygenerované display listy volať príslušnou funkciou, čím sa samotné vykreslenie na obrazovku značne zrýchli.

Ich použitie je však vhodné iba pri objektoch, ktoré sa za behu programu nijako značne nemenia, preto sú v animáciách vyžadujúcich častú zmenu objektov medzi jednotlivými zábermi nežiadúce. Sú však veľmi dobrým prostriedkom na prípravu procedurálne generovaných objektov ako napríklad písmen z fontov. Na ostatné účely však stále ostávajú Vertex Buffer Objekty vhodnejším riešením.



Obrázok 2: Display list umožňuje lepšiu prácu so statickými objektami, ktoré sa v scéne opakujú [7].

2.2.4 Vertex Buffer Objekt

Vertex Buffer Objekt (VBO) je rozšírením grafických kariet určeným na zlepšenie výkonu OpenGL. Pracuje podobne ako display list, s tým rozdielom, že vnútorné iba hromadne špecifikuje súradnice vrcholov a k ním pridruženým atribútom, ukladá ich v pamäti grafickej karty a všetky následné operácie vykonáva v ich optimalizovaných formách takisto priamo na rozhraní grafickej karty.

Veľkú výhodu oproti display listom poskytuje hlavne v tom, že takýto spôsob je rovnako prispôsobený na spracovanie údajov, ktoré sa môžu za behu programu často meniť. Jediná možná nevýhoda spočíva v tom, že všetky údaje o takto vykresľovaných objektoch musia byť vopred známe alebo spočítané, čiže sa nedá použiť na zapuzdrenie objektov vygenerovaných procedúrami rozhrania OpenGL, ktoré pracujú priamo s display listami alebo okamžitým módom.

Implementačne najvhodnejším riešením otázky o voľbe postupov pre vykresľovanie objektov v scénach je teda kombinácia všetkých dostupných možností a ich nevyhnutné následné zhodnotenie z hľadiska výkonu.

2.3 OpenGL Shading Language

Jednou z najdôležitejších častí každej 3D scény sú povrchy objektov, osvetľovací model, a iné efekty týkajúce sa ďalšieho spracovania vykresľovaných objektov bez ktorých by pôsobili ako dvojrozmerné placky na obrazovke a nedávali dobrý vizuálny efekt. Na ich tvorbu existujú veľmi rozsiahle možnosti a procedúry, pričom mnohé z nich sú priamo dostupné vo forme funkcií rozhrania OpenGL, avšak vektorové výpočty s nimi spojené sú v takomto prípade vykonávané automaticky a neposkytujú programátorovi žiaden priamy spôsob ich modifikácie a celkovej kontroly nad nimi.

Vtedy si treba pripustiť možnosť využitia OpenGL Shading Language (GLSL), ktorá je rozšírením väčšiny súčasných grafických kariet ako možnosť priamočiarejšieho a lepšieho ovládania vykresľovacích procesov skrz shader programy (skrátene shadery). Potrebné činnosti sú tak prístupnejšie a jednoduchšie, pretože GLSL poskytuje programátorovi vektorové operácie, ktoré nie sú v samotnom OpenGL prostredí dostupné, čo uľahčuje výpočty nutné v implementácii týchto prvkov.

2.3.1 Príkazy jazyka GLSL

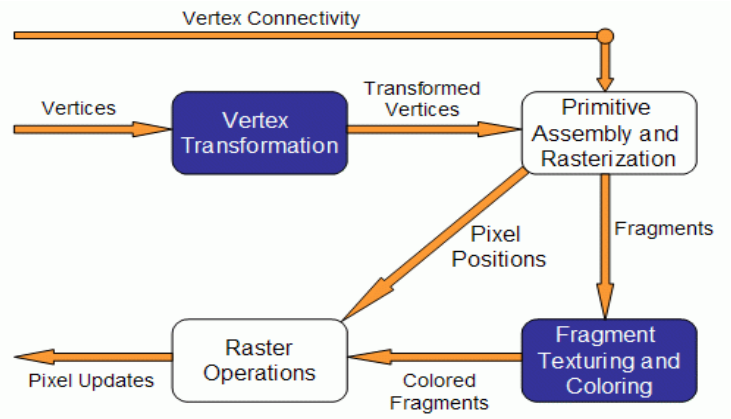
O tomto jazyku sa dá veľa dozvedieť napríklad z učebnice [3], ktorá vyšla v niekoľkých rôznych edíciách. Syntaxou a sémantikou je tento jazyk veľmi podobný C/C++, avšak hlavný rozdiel tu spočíva v používaných dátových štruktúrach a príkazoch.

Tento jazyk je hlavne zameraný na prácu s vektormi a maticami, čiže sa tu dajú priamo špecifikovať ako typ premennej a následne ich upravovať prostredníctvom optimalizovaných operácií ako napríklad skalárny súčin, normalizácia vektora alebo lineárna interpolácia.

Globálne premenné v rámci jednotlivých zložiek shadera programu tu takisto môžu byť špecifikované príslušnými kľúčovými slovami, ktoré určujú, či sa jedná o premenné určené k vzájomnému prenosu medzi zložkami, premenné určené v programe pred samotným vykresľovaním, alebo číselné atribúty vykresľovaných vrcholov.

2.3.2 Súčasti programu v jazyku GLSL

Konkrétny shader program teda predstavuje postupnosť príkazov určených k upresňovaniu jednotlivých vykresľovacích postupov. Takýto program vytvorený v jazyku GLSL sa skladá z niekoľkých funkčne odlišných častí, ktoré majú názov *vertex shader*, *fragment shader* a *geometry shader*. Prvé dva menované sú takmer povinnou súčasťou každého z nich, no posledný je novinkou predstavenou v OpenGL3.2 na ďalšie rozšírenie možnosti prispôsobenia vykresľovacieho procesu a na nižších verziách beží iba za podpory príslušného OpenGL rozšírenia. Celkovo vyzerá postup operácií vykonávaných pri vykresľovaní prvkov nasledovne:



Obrázok 3: Rendering pipeline; modré oblasti je možné priamo ovládať vertex a fragment shaderom.

Primárnou funkciou *vertex shadera* je transformácia obdržaných pozícií vrcholov podľa modelovej matice prostredia, čiže sa kód v tejto časti vykoná raz za každý daný vrchol. Takisto aj spracovanie všetkých pridružených atribútov vrcholov ako napríklad vektorov farieb a normál sa musí vykonať práve tu, čo otvára priestor pre rôzne špeciálne efekty týkajúce sa modifikácie týchto hodnôt, ako napríklad posun vrcholu v smere jeho normály (vertex displacement).

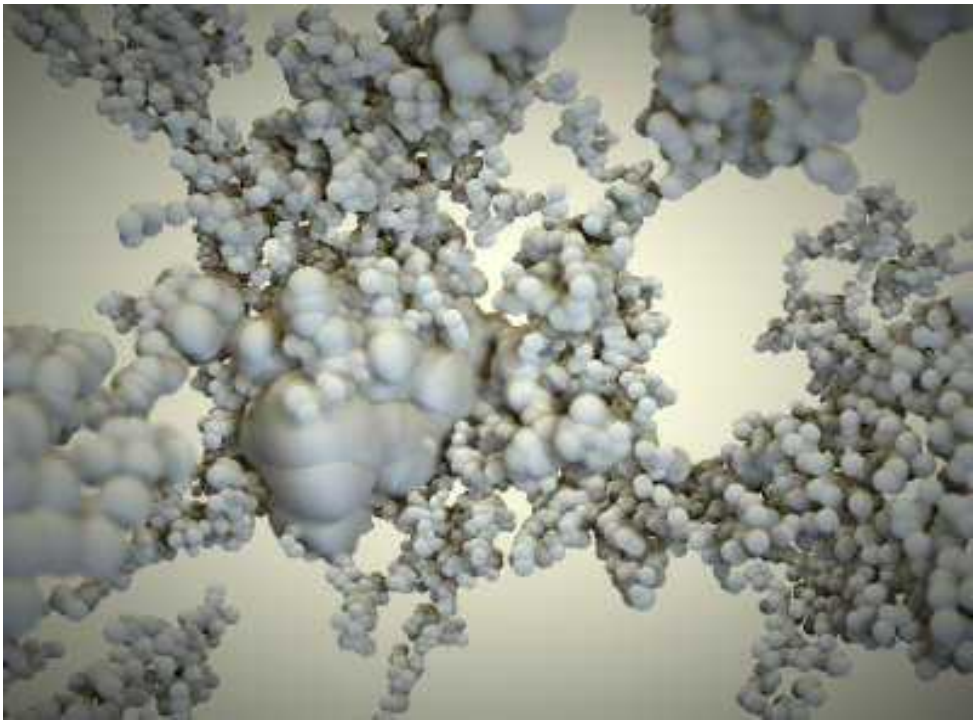
Typickou úlohou *fragment shadera* je zasa určenie farby výsledného fragmentu – pixelu, ktorý sa po vykonaní rastrových operácií umiestni na obrazovku, teda musí vedieť vykonávať všetky operácie súvisiace s počítaním použitej farby, jej hĺbky, alebo oboch. Hodnoty obdržané z predchádzajúcich zložiek sú teda interpoláciou hodnôt určených vrcholmi. Kód sa tu vykonáva raz

pre každý fragment, z čoho vyplýva, že za behu grafickej animácie sa vykonáva vo veľkých množstvách, a preto treba dbať na optimalizáciu implementovaného kódu, pretože veľmi ľahko tu môže dojsť k zahlteniu grafickej karty a teda i k podstatnému zníženiu celkového výkonu.

Geometry shader vstupuje do priestoru medzi predchádzajúcimi dvoma shadermi, kde má na vstupe všetky vrcholové atribúty vykresľovaných geometrických primitív (bodov, čiar, trojuholníkov) a má schopnosť popri povinnom predávaní údajov z vertex do fragment shadera takisto vytvárať úplne nové geometrické primitíva, ako napríklad kópie pôvodných primitív, iba na iných pozíciách, čo je konkrétne technika s názvom *geometry instancing* a využíva sa k veľmi efektívnemu a nenáročnému rozmiestňovaniu takmer identických objektov po scéne.

2.4 Veľkostné obmedzenia

Účelom veľkostných obmedzení u pôvodných dem bola prevažne obmedzená kapacita vtedajších prenosných záznamových médií a diskových priestorov, no požiadavky na ich dodržiavanie sa dochovali aj v súčasnosti ako druh výzvy pre tvoriaceho programátora.



Obrázok 4: Ukážka 4kB dema s názvom *kindercrasher [8]*.

Bez ohľadu na ich dôvod, zadaním dané veľkostné obmedzenia pre tento projekt sú neodbytne hlavným dôvodom, prečo je nutné opomenúť mnohé postupy používané v súčasnej profesionálnej počítačovej grafike, nakoľko pri maximálnej veľkosti 128kB nehrá možnosť uplatnenia veľkého diskového priestoru žiadnu úlohu a využitie externe definovaných zložitých modelov a povrchov nepripadá do úvahy. Jedinými prijateľnými možnosťami napokon ostáva buď procedurálne generovanie objektov a povrchov, alebo v prípade potreby začlenenie niektorých jednoduchších modelov do programu v rámci kompilácie v podobe hlavičkových súborov.

Pri návrhu teda bude najlepšie si nechať otvorené možnosti pre kombináciu obidvoch zo spomínaných prístupov, prípadne uplatniť iba také, ktoré budú zanechávať najlepší vizuálny efekt.

3 Návrh

Pred tvorbou grafického dema od samotných základov v prostredí OpenGL je nutné sa ustanoviť na jednotlivých programátorských postupoch, ktoré sa budú v rámci implementácie využívať. Toto zahŕňa mimo iného oblasť modelovania objektov, osvetľovacích modelov, použitia povrchov a písma v grafike, časticové systémy atď. Skladanie týchto prvkov dohromady umožňuje vytvárať animované scény figurujúce ako jednotlivé súčasti demo aplikácie.

Následujúce riadky teda pokrývajú problémy s tým spojené a riešia otázky týkajúce sa voľby vhodných smerov pri budúcej implementácii. Tu mi vo významnej miere pomohli pomimo knižných zdrojov tutoriály dostupné na internete, najmä [7] pre prácu s OpenGL prostredím a [9] spolu s [10] pre jazyk GLSL.

3.1 Objektový návrh aplikácie

Keďže sa nejedná o výpočtový problém s triviálnym riešením, je nutné si ho rozdeliť na viacero logicky odlišných celkov a zohľadniť túto skutočnosť aj pri samotnej implementácii. Inými slovami treba ešte pred samotnou tvorbou programu vytvoriť objektový návrh aplikácie, ktorý bude značiť, aké objekty je nutné zadefinovať.

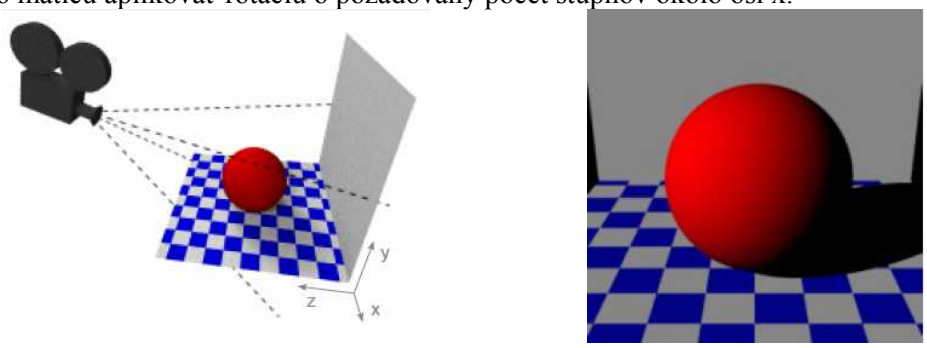
Jadrom programu a teda i jeho vstupným bodom jednoznačne musí byť inicializácia a spustenie potrebných vlákien a okien programu, teda hlavného okna a okna na výber rozlíšenia, pričom musí ostať priestor na tvorbu vlákna časovača programu a zaistenie jeho komunikácie s hlavným vláknom.

Ďalej je veľmi dôležitý objekt manažéra scény prispôsobený na priame zaobchádzanie s prostredím OpenGL, zapuzdrovanie práce s jednotlivými scénami a uskutočňovanie prechodov medzi nimi. Aj samotné scény všetky musia byť zvlášť tvorené svojim objektom, zdedeným z jednej abstraktnej triedy, ktorá zahŕňa základné funkcie na ich vykresľovanie a aktualizáciu dynamicky menených parametrov.

Podľa potreby sú ešte v programe definované aj ďalšie objektové triedy slúžiace na zapuzdrenie funkcií spojených s pridávanými rozšíreniami OpenGL prostredia. Tie sú formálne riešené v príslušných kapitolách.

3.2 Pohľad scény

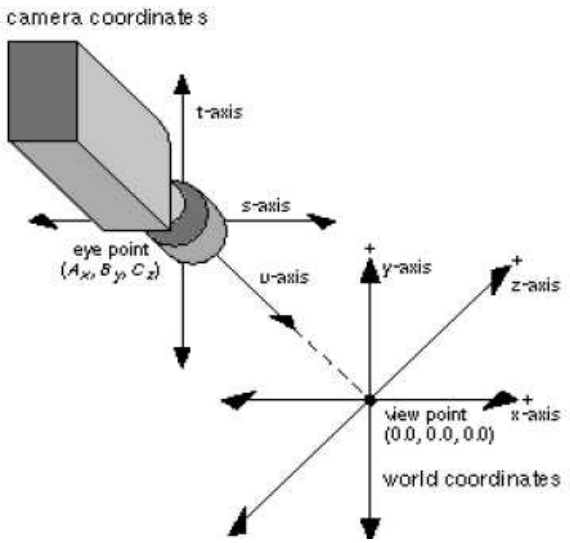
Pred umiestňovaním akýchkoľvek objektov do scény je nutné navrhnúť, akým spôsobom bude uskutočňovaný pohľad na ne, a teda umiestnenie a pohyb kamery. Najjednoduchším riešením tohto problému je posun jednotkovej modelovej matice o príslušnú vzdialenosť na z-ovej osi v zápornom smere, čím sa celý pohľad na následne vykresľovanú scénu oddiali, a pre lepší uhol pohľadu stačí už len na túto maticu aplikovať rotáciu o požadovaný počet stupňov okolo osi x.



Obrázok 5: Definícia statickej kamery pre jednoduchú 3D scénu [1].

Kamera vytvorená postupom popísaným v predchádzajúcich riadkoch je statická a pre lepšiu manipuláciu s jednotlivými scénami je nutné zaviesť rozšírenie o rotáciu pozdĺž zvislej osi a prenechať riadenie príslušných transformačných funkcií priebežne meneným premenným, čím sa docieli jednoduchej dynamickej kamery. Ňou sa dajú v jednoduchších scénach zaznamenať všetky dôležité prebiehajúce udalosti, avšak pre pokročilejšiu manipuláciu s kamerou, ako napríklad pohyb po zvolenej trase, je nedostačujúca.

Na flexibilnejšie modelovanie kamery scény slúži spôsob jej definície podľa súradníc jej polohy, polohy sledovaného bodu a vektora, čo určuje, ktorým smerom má byť pohľad otočený nahor. Rovnako, ako predchádzajúci prístup funguje na základe vykonávania transformácií na modelovej matici, takže pre správny beh je nutné túto funkciu volať pred akýmkoľvek vykresľovaním objektov, avšak pre docielenie špecifickej polohy a orientácie kamery pri tejto funkcii už nie je nutné poznať všetky transformácie nutné na presun kamery do žiadaného bodu a otočenie do správneho smeru.



Obrázok 6: Definícia kamery prostredníctvom funkcie `glLookAt()` [1].

Celkovo však voľba postupu pri manipulácii s kamerou závisí hlavne od jej účelu a celkovo od obsahu konkrétnej scény. V mnou navrhovanej aplikácii je táto skutočnosť aj patrične zohľadnená.

3.3 Modelovanie objektov scén

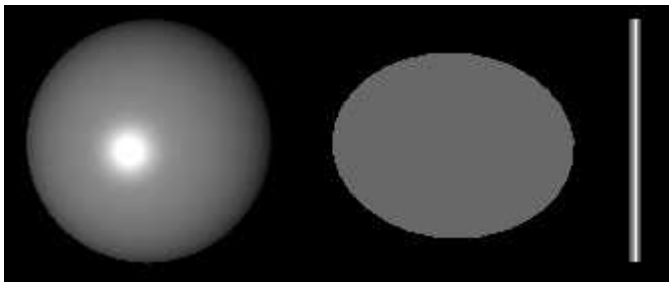
Trojrozmerné objekty neodbytne tvoria základ každej 3D animácie, takže vo veľkostne obmedzenom deme je v prvom rade nutné nájsť spôsob ich vykresľovania bez nutnosti importu z vonkajších súborov. Tu je množné uplatniť niektoré z procedúr priamo dostupných v rozhraní OpenGL.

Takisto použitie statických scén je v grafickom deme nežiadúce, takže zaistenie dynamickosti objektov je pri implementácii nutnosťou. Priamou možnosťou riešenia tohto problému je teda spolu s objektami jednotlivých scén navrhovať aj štruktúru s ich parametrami, ktoré sa za behu programu inkrementujú v tele metód vykonávaných medzi jednotlivými zábermi a pomáhajú tak vytvárať grafické sekvencie.

3.3.1 Kvadriky

Jednou z najviac priamočiarych možností je využitie objektov s kvadrickými povrchmi (kvadrikov), poskytovaných v nadstavbovej knižnici GLU ako príkazy na modelovanie niektorých bežne používaných 2D alebo 3D objektov ako sú kruhy, gule, alebo valce. Ich parametre sú ľahko

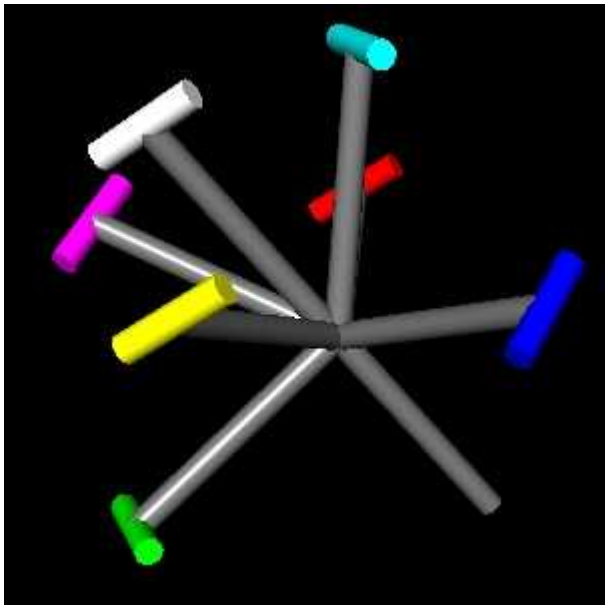
prispôsobiteľné, čo umožní priame ovládanie ich šírky a dĺžky a dáva príležitosť robiť objekty animácie zložitejšie a dynamickejšie než pri použití iných prístupov.



Obrázok 7: Niektoré z kvadrikov.

Hlavnou nevýhodou takto generovaných objektov je však to, že na ich vykreslenie používajú tieto procedúry okamžitý mód a teda môžu ohroziť výkon programu. Tomuto sa dá zamedziť buď redukciou počtu ich vrcholov daných parametrami funkcií a teda i zníženiu ich celkovej kvality, alebo zapuzdrením týchto objektov do display listov, no vzhľadom na to, že by tak stratili výhodou dynamických zmien ich parametrov medzi zábermi, je tento spôsob v mnou navrhovanom deme nežiadúci.

V rámci implementácii týchto objektov je teda dôležité sledovať, akým spôsobom negatívne ovplyvňujú beh animácie a teda i prakticky vyskúšať, koľko sa ich dokáže do scény zmestiť tak, aby bol beh animácie stále plynulý.



Obrázok 8: Zložitejší objekt vytvorený kombináciou valcov a kruhov.

3.3.2 3D texty

Pre lepší vizuálny účinok a interakciu s pozorovateľom je vhodné začleniť do scény objekty utvorené z fontov a vypisovať nimi správy použiteľné ako popis toho, čo presne sa v prebiehajúcej scéne deje. Na tento účel existujú funkcie v OpenGL, ktoré umožňujú vytvárať display listy z fontov v systémových adresároch. Tým pádom sú ich inicializácia a použitie jednoduché a flexibilné.

Tu sa naskytujú dva možné postupy. Prvým z nich je funkcia, ktorá vytvorí display list naplnený bitmapovými fontami, a druhým funkcia na tvorbu fontov tvorených buď čiarami alebo polygónmi, čím sa ukazuje byť pre účel dema vhodnejšou.

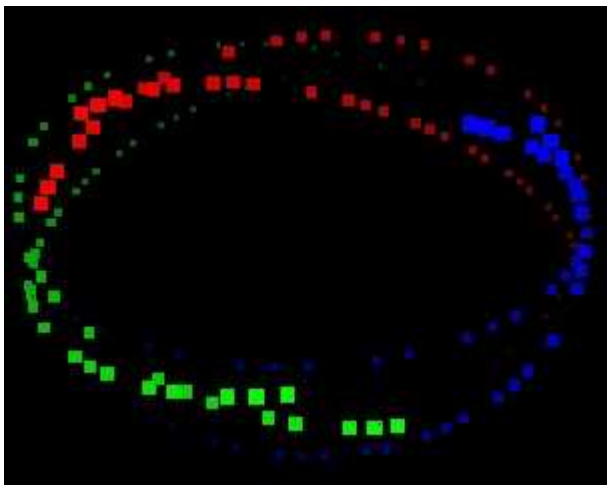
Vzhľadom na ich už spomenutú realizáciu prostredníctvom display listov sú tieto objekty málo náročné na výpočtový čas, a teda sa môžu bezpečne používať v akejkoľvek grafickej scéne, ba dokonca ich je aj možné v demách vídať celkom často. Jedine však treba dávať pozor na to, že v rámci svojich display listov môžu nastavovať premenné prostredia inak, než je pre zvyšok scény žiadúce, a teda v prípade akýchkoľvek problémov je nutné volania týchto display listov ošetriť úschovou a opätovným načítaním stavových premenných.

Obrázok 9: Otáčavý popisný text utvorený vhodným rozmiestnením objektov fontov.

3.3.3 Časticové systémy

Na podnet starých sporičov obrazovky, ktoré sa zakladali na blikaní hviezdičiek alebo iných jednoduchých objektov v rozličných častiach obrazovky, sa naskytuje možnosť zvážiť použitie častíc a navrhnúť vhodný časticový systém. Vkladať štvorce do scény je neefektívne kvôli zachovávaniu ich natočenia ku kamere. Na druhej strane, body a ich varianta s názvom point sprites sú pre túto úlohu oveľa vhodnejšie. Pozíciu častíc v scéne je možné priebežne ukladať a potom spätne vyvolávať napríklad pomocou translačných alebo celých modelových matíc.

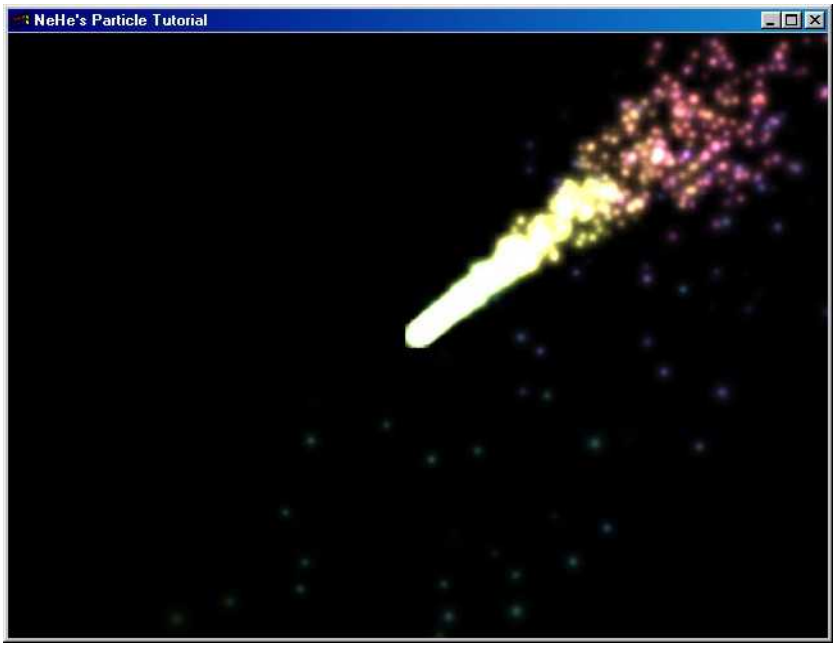
Tieto point sprites sa v programe umožňujú príkazom `glEnable(GL_POINT_SPRITE)`, ktorý spôsobí, že pri vykresľovaní bodov o veľkosti viac než 1 sa každému z ich fragmentov priradí rovnaký stav, čo zlepší ich antialiasing a konzistentnosť operácií s nimi prevedenými v rámci shaderu.



Obrázok 10: Point sprite častice v dynamickom pohybe.

Takto navrhnutý časticový systém so sebou prináša radu možných budúcich vylepšení a optimalizácií. Jeho hlavným nedostatkom je samotný tvar jednotlivých častíc, ktorý bez akejkoľvek vonkajšej úpravy ostáva ako štvorec. Toto je možné riešiť mapovaním gradientných textúr v iných tvaroch, ako napríklad kruhu, na povrch bodov, respektíve ich orezaním v programe shadera pomocou goniometrických funkcií.

Takisto ich priebežné ukladanie pomocou modelových matíc nie je úplne optimálnym riešením a znemožňuje dynamický pohyb kamery, čím by sa takto uložené body stali neaktuálne. Jediným riešením tohoto problému je však zlepšenie celkového používaného modelu tak, aby bol schopný zaznamenávať aj relatívnu pozíciu bodov oproti počiatku sústavy, v ktorej prebieha vykresľovanie. Keďže ale je časticový systém mojej animácie postavený iba do jednej z vedľajších úloh, súčasne navrhované riešenie je pre ne postačujúce a ako príklad zložitejšej práce s časticami uvádzam výstup jedného z tutorialov na [7].



Obrázok 11: Zložitejší časticový systém.

3.4 GLSL efekty

Jazyk GLSL má na modelovanie osvetlenia, povrchov objektov a špeciálnych efektov veľmi rozšírené možnosti. Umožňuje to mimo iného aj už spomínaná podpora datových typov pre vektory a matice. Samotné prevedenie a implementácia týchto funkcií však nie je jednoznačná, ani vykonávaná automaticky pomocou dostupných procedúr, a preto sa treba vopred rozhodnúť, ktorý z modelov treba pre daný shader zvoliť.

3.4.1 Osvetľovací model

V OpenGL sa rozlišuje viacero druhov osvetlenia, no pri návrhu sa budem zaoberať iba najčastejšie používaným z nich, a teda smerovým, v ktorom hrá dôležitú úlohu jeden z najzákladnejších osvetľovacích modelov, Phongov model, v ktorom konečná farba zobrazovaného fragmentu je výsledkom nasledujúcej rovnice:

$$I_f = I_a + I_d + I_s \qquad (1)$$

V rovnici (1) sa výsledná farba fragmentu $\mathbf{I_f}$ počíta ako súčet intenzity farby prostredia $\mathbf{I_a}$ (ambient), intenzity difúznej farby $\mathbf{I_d}$ (diffuse) a intenzity farby odlesku $\mathbf{I_s}$ (specular). Shader vykonávajúci tieto počty bude teda počítať takéto osvetlenie postupnou inkrementáciou jeho zložiek od farby prostredia až po farbu odlesku. Pri výkone týchto výpočtov vo vertex shaderi by teda výsledný objekt bol zobrazený nasledovne:

Obrázok 12: Osvetlenie rátanie pre každý vrchol postupnou inkrementáciou jednotlivých zložiek [9].

Tento prístup však už od pohľadu vykazuje značené nedostatky spôsobené tým, že zložky svetla sa v takomto prípade počítajú iba v jednotlivých vrcholoch a hodnoty v priestoroch medzi nimi sú vnútorne počítané lineárnou interpoláciou, čo je pre výsledný efekt nežiadúce, nakoľko niektoré zo zložiek osvetlenia pri svojom výpočte zohľadňujú uhol medzi polohou kamery a svetla s počiatkom v dotyčnom bode fragmentu. Takéto hodnoty je teda vhodnejšie počítať priamo v tele fragment shadera a výsledný objekt po uplatnení osvetľovacieho modelu podľa správnosti vyzerá takto:

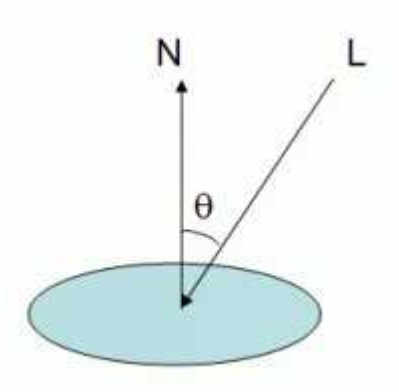
Obrázok 13: Smerové osvetlenie počítané pre každý fragment [9].

Čo sa teda týka samotného výpočtu už spomínaných jednotlivých zložiek, tak prvá z nich – ambient zložka je výsledkom násobku ambient zložky použitého svetla a ambient zložky použitého materiálu:

$$I_a = (A_l * A_m) + (A_s * A_m) \qquad (2)$$

V rovnici (2) teda prestavujú neznáme $\mathbf{A_l}$ ambient zložku použitého svetla, $\mathbf{A_m}$ zasa materiálu, a zohľadňuje sa tu aj globálna ambient zložka v podobe $\mathbf{A_s}$.

Difúzna zložka osvetlenia je v Phongovom modeli založená na Lambertovom kosínovom zákone, ktorý stanovuje, že jas difúzne žiariacej roviny povrchu je priamo úmerný kosínu uhla tvoreným líniou pohľadu a normálou povrchu. Grafické znázornenie tohto uhla a rovnice na výpočet difúznej zložky teda vyzerajú nasledovne:

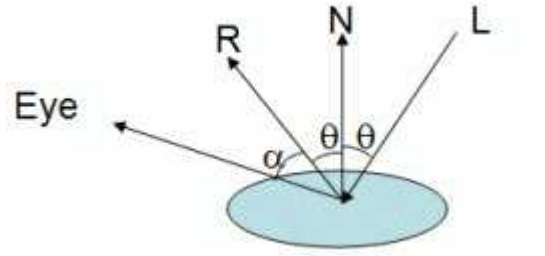


Obrázok 14:Uhol theta figurujúci v Lambertovom kosínovom zákone [9].

$$I_d = D_l * D_m * \cos(\theta) = D_l * D_m * (\overline{N} . \overline{L}) \quad (3)$$

Kosínus uhla theta je v rovnici (3) nahradený skalárnym súčinom vektorov **N** a **L**, ktorých význam bol vysvetlený v predchádzajúcich riadkoch. Okrem nich tu vystupujú neznáme $\mathbf{D_l}$ a $\mathbf{D_m}$ značiace difúznu zložku použitého svetla a materiálu.

Lesklá zložka osvetlenia je na výpočet najnáročnejšia a značí odraz svetelného zdroja na povrchu objektov. Je priamo úmerná kosínu uhla tvoreného vektorom odrazeného svetla a vektora smerujúceho k pozícii kamery. Ten je znázornený na tomto obrázku a použitý v nasledujúcej rovnici:



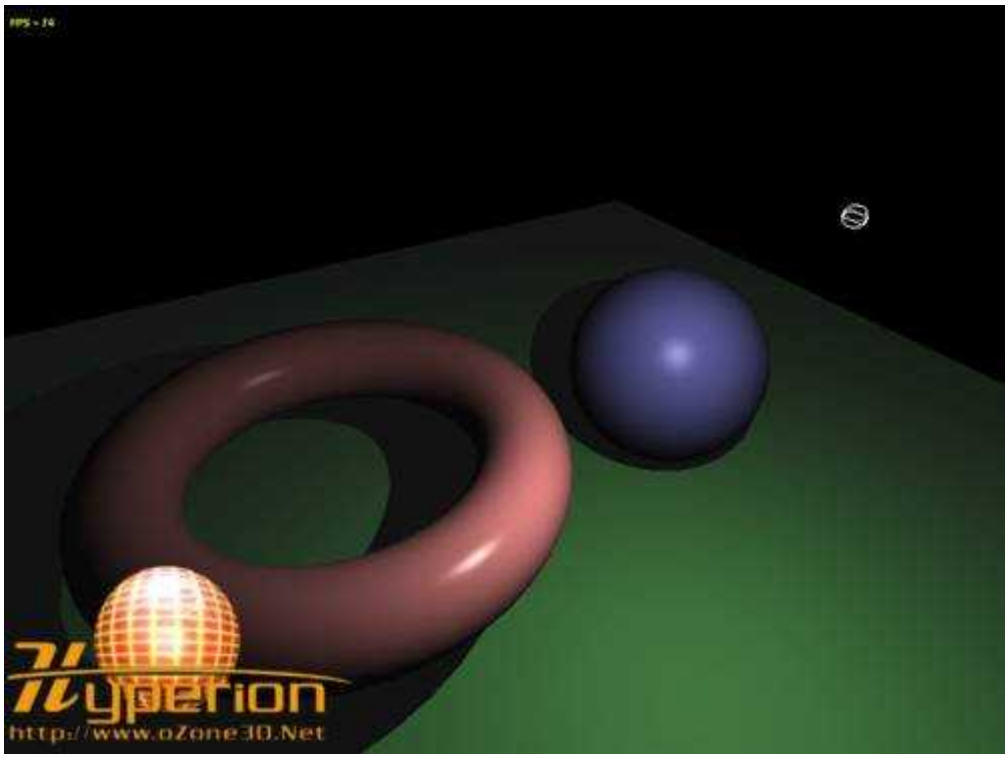
Obrázok 15:Uhol alfa figurujúci v rovnici lesklej zložky [9].

$$I_s = S_l * S_m * (\cos(\alpha))^S = S_l * S_m * (\overline{R} . \overline{Eye})^S = S_l * S_m * (\overline{N} . \overline{H})^S \quad (4)$$

V rovnici (4) figurujú neznáme $\mathbf{S_l}$ a $\mathbf{S_m}$ ako lesklé zložky svetla a materiálu, a následne **S** ako exponenčný faktor (lesklosť v OpenGL). Pre zložitosť implementácie tejto rovnice GLSL spôsobenou tým, že vektor **R** je v tomto vzorci vopred neznámy a treba ho dopočítať, sa využíva optimalizovaný Blinn-Phongov osvetľovací model využívajúci vektor **H** umiestnený v polovici medzi vektormi **Eye** a **L** a v rozhraní OpenGL priamo dostupný.

Takto vytvorený osvetľovací model v jazyku GLSL dôsledne napodobňuje smerové osvetlenie bežne používané v zafixovanom vykresľovacom postupe grafického hardvéru, takže je pre potreby osvetlenia úplne dostačujúce a vo všetkých prípadoch použité aj v mojej aplikácii.

Tento osvetľovací model takisto ponúka radu možných vylepšení pre lepšie napodobenie svetla z reálneho svetla, nakoľko v takejto forme nijako nerieši vytváranie tieňov, ani slabnutie intenzity svetla v závislosti od vzdialenosti fragmentu od zdroja osvetlenia. Pokročilejšie osvetlenie teda môže vyzerať napríklad takto:



Obrázok 16: Pokročilé osvetlenie založené na Phongovom osvetľovacom modeli [10].

3.4.2 Procedurálne povrchy

K tomu, aby sa vyhlo použitiu externe definovaných textúr, musia byť nahradené vhodnými druhmi procedurálne generovaných povrchov, napríklad takými, čo sú generované fraktálmi alebo šumovými funkciami.

Toto zahŕňa aj tzv. Perlin šum, čo je algoritmus na prevedenie hrubej šumovej funkcie na viac spojité a organizovanejšie polia hodnôt, ktoré sa hodia presne na účel tvorby povrchov. Aj v reálnom čase tak dokážu tvoriť rôzne tvarované materiály v rozumnej výpočtovej dobe. Tento algoritmus navrhol Ken Perlin a jeho podstata je dôkladne vysvetlená a objasnená v mnohých publikáciách a výukových knihách, menovite napríklad [4].

Základom pre implementáciu tohto algoritmu je teda šumová funkcia, ktorá nemusí spĺňať podmienku toho, aby zakaždým pre rovnaké vstupy dávala iné výstupy, takže pri jej definícii stačí spojiť všetky vstupujúce parametre skalárnym súčinom ich vektorov a výslednú hodnotu nechať prejsť goniometrickými funkciami, pričom sa do výslednej rovnice zakomponujú aj matematické operácie s prvočíslami, aby mali výsledné hodnoty dostatočne veľký rozptyl a boli lineárne nezávislé.

Výstupy takejto funkcie je potom možné použiť ako vstupy interpolačnej funkcie na vyhladenie hodnôt medzi jednotlivými fragmentami a zaistenie ich spojitosti. Najprv však treba z hrubého šumu dostať hladký šum, ktorý potláča extrémne hodnoty a k účelu spojitosti môže využívať buď 4-okolie alebo 8-okolie bodov a váhové priemery hodnôt hrubého šumu v ich koordinátoch. Spojením týchto dvoch postupov je možné zložiť funkciu na tvorbu interpolovaného šumu, ktorá najskôr zostaví hladké šumové hodnoty dvoch susediacich fragmentov a priestor medzi nimi vyplní vhodnou interpolačnou metódou. Pri generovaní šumu v reálnom čase je však rozumnou voľbou ostať pri 4-okolí a použiť iba lineárnu interpoláciu namiesto kosínovej alebo kubickej, čomu prispieva aj fakt, že jazyk GLSL má v sebe zakomponovanú optimalizovanú funkciu na lineárnu interpoláciu hodnôt.

Samotná funkcia Perlin šumu potom využíva takto obdržané hodnoty šumu na generovanie šumových funkcií s rôznymi amplitúdami, ktoré klesajú geometrickým radom podľa parametra vytrvalosti, a počet funkcií, ktoré takto treba vypočítať, je daný parametrom určujúcim počet oktáv,

kedy každá z funkcií sa volá jedná oktáva. Výsledné funkcie sú potom zložené dohromady a vzniká funkcia zvaná Perlin šum, ktorej prechodmi cez rôzne goniometrické funkcie a lineárne filtre je možné dostať veľkú škálu povrchov.

Obrázok 17: Perlin šum o vytrvalosti 0,25 a počte oktáv 4, napravo ešte päťnásobne oddialený.

S implementáciou tohto algoritmu v GLSL sa takisto viaže niekoľko úskalí. V prvom rade počítanie týchto funkcií za behu programu so sebou nesie aj značné množstvo potrebného výpočtového času, čo pri veľkých povrchoch môže spôsobiť výrazné spomalenie behu programu, a je dôvodom, prečo povrchy bývajú zvyčajne uložené v nemenných štruktúrach. Takýto prístup by však zamedzil využiť možnosť dynamickej zmeny charakteristík týchto povrchov medzi jednotlivými zábermi.

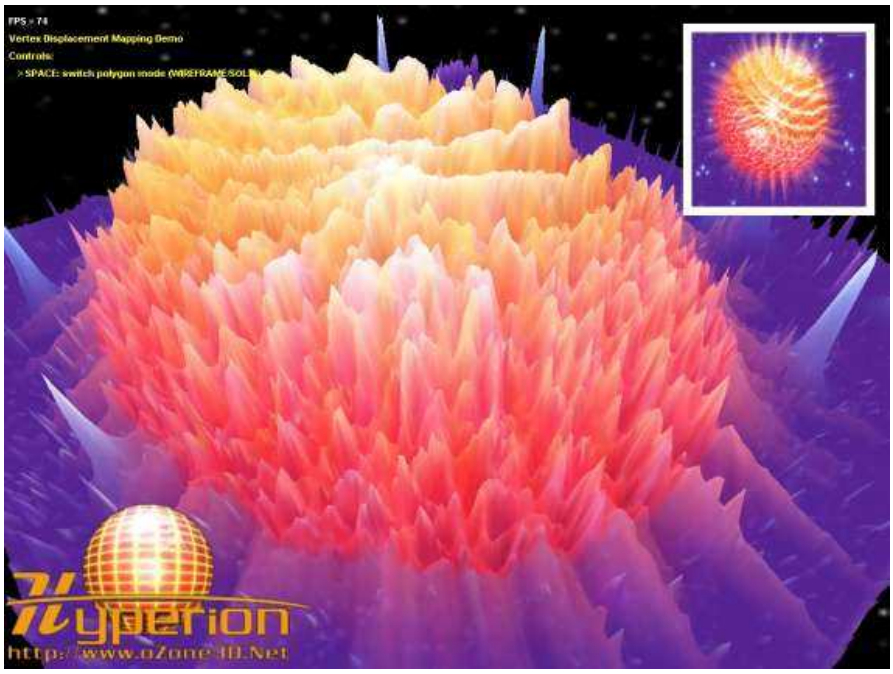
V druhom rade aj voľba premenných na vstupov tejto funkcie môže ovplyvňovať jej bežný chod, nakoľko napríklad povrchové koordináty objektov sa štandardne pohybujú medzi 0,0 a 1,0, teda ich najprv nutne treba namapovať na väčší rozsah.

Všetky tieto ohľady som pred zvažovaním možných optimalizácií pri implementácii prakticky odskúšal a stručne zhodnotil z hľadiska ich vplyvu na celý výkon v kapitole 4.5, no vzhľadom na rozsah môjho projektu a pomerne jednoduchú zložitosť jednotlivých scén, nebola žiadna kriticky požadovaná optimalizácia nutná.

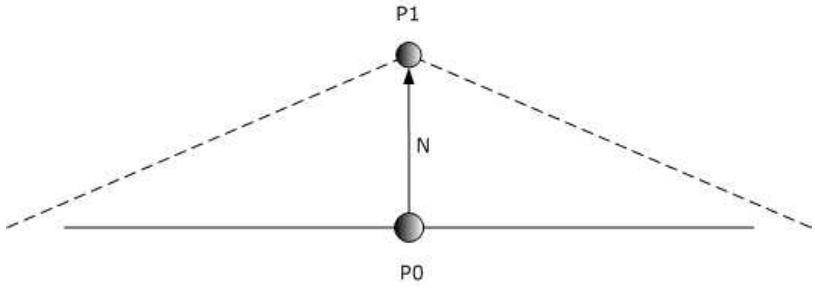
Obrázok 18: Povrch utvorený zložením dvoch funkcií Perlin šumu, jednej o vytrvalosti 0,25 a počte oktáv 4, druhej o vytrvalosti 0,5, generovaný v réálnom čase.

3.4.3 Vertex displacement

Vertex displacement je technika umožňujúca deformácie povrchov objektov, čím sa okrem modelovania zložitejších objektov z jednoduchších dajú takisto tvoriť objekty, ktoré svoju štruktúru dynamicky za behu programu menia. V princípe sa jedná o posuv vrcholových súradníc v smere ich povrchových normál a zvyčajné bývajú dané formou dvojrozmerných polí – máp, ktoré presne udávajú vzdialenosť nového bodu od pôvodného v smere normálového vektora.



Obrázok 19: Vertex displacement [10].

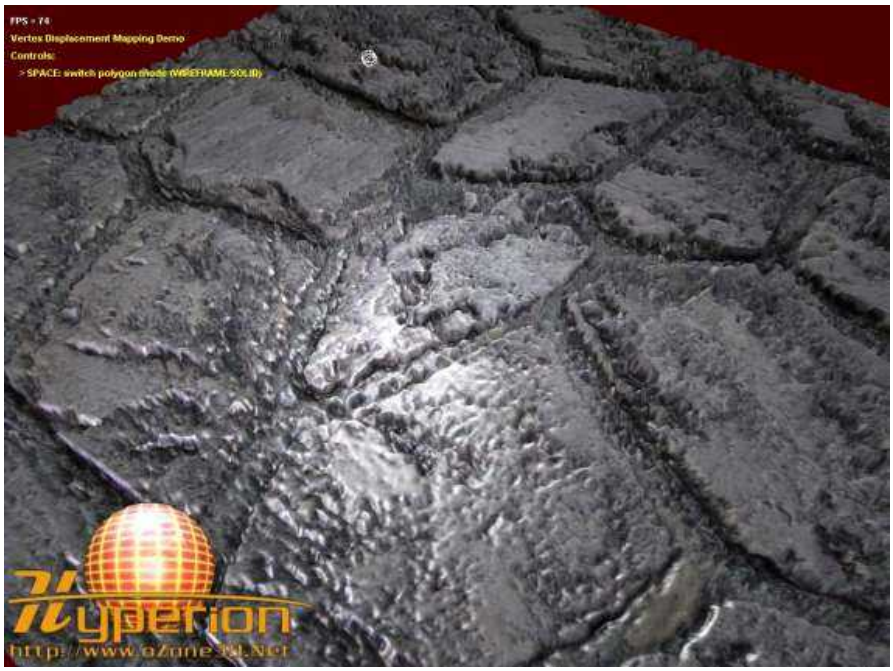


Obrázok 20: Princíp vertex displacementu [10].

$$P_1 = P_0 + (\overline{N} * d_f * u_f) \tag{5}$$

V rovnici (5) účinkujú neznáme $\mathbf{d_f}$ a $\mathbf{u_f}$, ktoré značia normalizovaný rozsah displacementu a užívateľsky definovanú mierku. Neznáma $\mathbf{d_f}$ býva zvyčajne takisto použitá aj na výpočet farby výsledného fragmentu, ba dokonca v niektorých prípadoch vopred spočítaná z požadovanej farby jej prevedením na hodnotu stupňa šedi. V mojej aplikácii však volím úplne iný prístup, kedy ako hodnotu tejto neznámej používam výstup funkcie Perlin šumu, ktorú je v programe už použitá pri generovaní povrchov, čo mi oproti prístupu s použitím namapovaných hodnôt umožňuje dynamicky meniť hodnoty deformácie za behu. Výstupné hodnoty rovnako zohľadňujem aj pri určovaní výslednej farby deformovaného povrchu.

Táto technika však v sebe obsahuje jedno veľké úskalie, a teda že pri zmenách pozícií vrcholov sa pôvodné normálové vektory stávajú neplatnými a v prípade ich použitia pri výpočte osvetlenia by nevracali správne hodnoty. Najviac priamočiarym riešením sa tu naskytuje využitie osvetľovacieho modelu, ktoré vo svojich výpočtoch povrchové normály nezohľadňuje, avšak v iných prípadoch neostáva nič iné, než predsúvať programu fragment shadera prepočítané hodnoty normálových vektorov. Pri hodnote displacementu danou poľom hodnôt sa tento prepočet vykonáva vypracovaním ďalšieho poľa hodnôt, a to pre normály platné pre už upravený povrch objektu, respektíve pri celom výpočte používať iba toto pole a z neho priamo rátať požadovaný displacement.

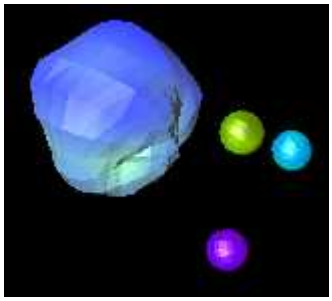


Obrázok 21: Oprava povrchových normál za použitia normálovej mapy [10].

Na implementáciu tohoto postupu a jeho modifikácií je však nutné programu vertex shadera dodávať pre každý vrchol aspoň hodnoty displacementu alebo požadovaných výsledných normál jeho susedných vrcholov, čo pri mnou zvolenom postupe využívajúcom Perlin šum v reálnom čase nie je dostupné. Preto je v mojom shader programe nutné prepočítavať normálové vektory úplne iným spôsobom.

Tu sa ponúka využiť výhody geometry shadera obdržiavať na svojom vstupe geometrické primitíva s už upravenými polohami vrcholov vykonaných za behu vertex shadera, a využiť ich na vypočítanie nových normálových vektorov. Samotný výpočet teda prebieha vektorovým súčinom ľubovoľných dvoch vektorov vychádzajúcich z jedného vrcholu primitíva a smerujúcich do aspoň ďalších dvoch, čím sa získa vektor kolmý na celý povrch.

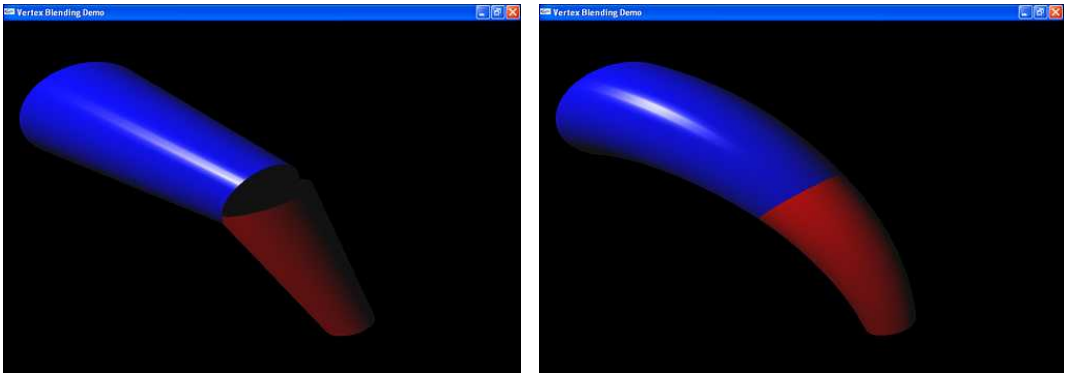
Jedinou nevýhodou tohto použitého postupu je, že takto vypočítané normály majú rovnaké hodnoty pre každý výsledný fragment v rámci jedného geometrického primitíva, čo spôsobuje, že po aplikovaní osvetlenia nepôsobí vymodelovaný povrch ako hladký, ale sú na ňom jasne vyprofilované hrany, čo môže, ale aj nemusí byť pre výsledný efekt nežiadúcim. To však v princípe závisí predovšetkým od osobného názoru pozorovateľa.



Obrázok 22: Oprava povrchových normál v geometry shaderi.

3.4.4 Vertex blending

Ďalšou užitočnou technikou v GLSL je vertex blending. Jedná sa o zjednocovanie dvojíc odlišných vrcholov za účelom vyplnenia priestoru medzi nimi a teda modelovanie jedného súvislého objektu z viacerých iných rozmiestnených v priestore. Lepšie tento princíp znázorňuje nasledujúci obrázok.



Obrázok 23: Objekt tvorený z dvoch valcov pred a po aplikácii techniky vertex blending [5].

Táto operácia sa teda vykonáva v tele programu vertex shadera, a to tak, že každému vrcholu je udelená nová pozícia lineárnou interpoláciou s príslušným vrcholom, kde požadovaná pozícia medzi týmito dvoma vrcholmi je udávaná parametrom s názvom *váha*. Výsledná normála v tomto bode sa rieši obdobným spôsobom.

Najdôležitejšie však je, aby mal každý vrchol jasne udanú pozíciu druhého vrchola, s ktorým má vertex blending vykonať. Toto sa dá riešiť buď prostredníctvom uniform premenných v kóde vertex shadera, kde je takto udaná modelová matica, v ktorej sa žiadaný vrchol nachádza. Tá však nemusí byť vždy presne známa a vo svojej implementácii tejto techniky riešim tento problém tak, že súradnice priľahlých vrcholov a normál predávam prostredníctvom vertex atribútov.

Keďže logicky je možné vykonať tento proces úplne mimo akéhokoľvek shader programu a prepočítavať nové pozície vrcholov a normál už pred ich samotným predkladaním k vykresleniu, jedná sa čisto o techniku vhodnú pre zlepšenie celkového výkonu programu, nakoľko pri prenechaní výpočtov programu shadera sa výrazne odľahčuje záťaž procesora.

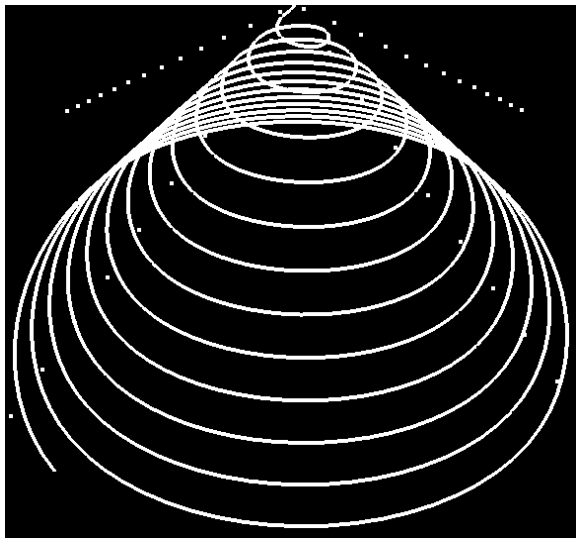
3.5 Spline krivky

Nie náhodou je spline krivkám venovaná kapitola sama o sebe. Hrajú totiž veľmi veľkú úlohu nielen pri modelovaní objektov, ale aj ich pohybových dráh a iných užitočných prvkov v súčasnej počítačovej grafike. Slúžia na popis krivých tvarov, nakoľko grafické prvky generované čisto prostredníctvom rovných čiar a oblých tvarov môžu ľahko pôsobiť zbytočne neprirodzene. V plnom rozsahu sa nimi zaoberá veľké množstvo kníh a publikácií, takže ich popis v nasledujúcich riadkoch je podaný v rámci stručnosti a zaoberá sa iba jediným druhom spline krivky a teda B-spline, pretože to je práve tá, ktorú som vo svojej implementácii použil pre jej vlastnosti.

Jedná sa o aproximačnú krivku, ktorá je určená n+1 koncovými bodmi, kde n je jej polynómový stupeň, a koncovými riadiacimi bodmi neprechádza. S touto krivkou sa pracuje po segmentoch a pridávaním ďalších kontrolných bodov sa takisto zvyšuje počet výsledných segmentov, no počet bodov prináležiacich výslednej krivke závisí čisto od implementačných detailov a nie je prakticky ničím výrazne obmedzený. Výpočet súradníc **S** bodu ležiaceho na v rozmedzí kubickej reprezentácie tejto krivky je znázornený v rovnici (6), kde **k** je z intervalu [0,1] a značí kľúč prináležiaci konkrétnemu bodu krivky určeného kontrolnými bodmi **C**.

$$S_i(t) = \begin{bmatrix} k^3 & k^2 & k & 1 \end{bmatrix} * \frac{1}{6} * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} * \begin{bmatrix} C_{i-1} \\ C_i \\ C_{i+1} \\ C_{i+2} \end{bmatrix} \quad (6)$$

Týmto postupom sa dajú ľahko vygenerovať krivky so segmentami oblých tvarov a vygenerované body krivky môžu byť použité napríklad pre určenie dráhy pohybu kamery, alebo jednoducho len pre vykreslenie čiar s vrcholmi v jednotlivých bodoch krivky.



Obrázok 24: B-spline krivka v podobe čiar v priestore, tvar špirály. Znázornené sú aj niektoré z jej kontrolných bodov.

Pre modelovanie zložitejších objektov za použitia spline kriviek je v tejto aplikácii uvedený ako príklad zaobalenie tejto krivky do prstencových útvarov navzájom pospájaných technikou vertex blending pre vytvorenie súvislého tunelu a následné použitie tej istej krivky pre plynulý pohyb kamery.

Vrcholy spomínaného objektu sú pri tomto prístupe počítané tak, že pre každé dva po sebe nasledujúce body krivky sa počíta normalizovaný vektor začínajúci v prvom z bodov a smerujúci do druhého z nich. K tomuto vektoru sú vytvorené dva vektory naň kolmé – prvý z nich jeho ortogonálnym priemetom do osi Z a druhý vypočítaný ako vektorový súčin predchádzajúceho a pôvodného vektora. Tieto dva vektory sú následne predĺžené o vhodný rozsah a za pomoci goniometrických funkcií z knižnice `math.h` sú vyrátané súradnice vrcholov opisujúce kružnice okolo dvoch použitých bodoch krivky.

3.6 Zrkadlové plochy

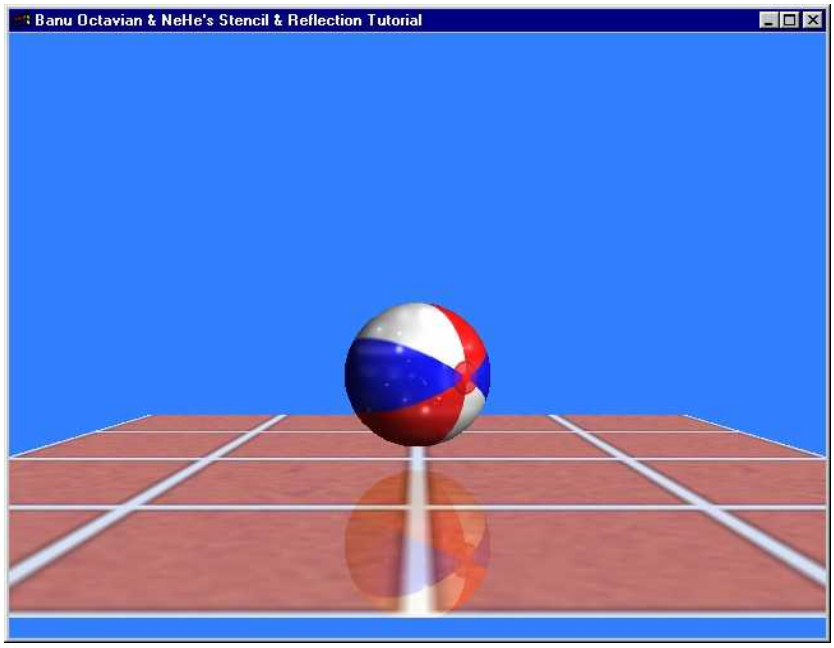
Tvorba povrchov s vlastnosťou odrážať okolité dianie ako zrkadlá je vítaným efektom v akejkoľvek 3D grafickej scéne. V prostredí OpenGL sa realizuje spojením dvoch funkčne odlišných techník, a síce vykresľovaním do stencil bufferu a zmiešavaním farieb (blending).

Stencil buffer slúži na zamedzenie vykresľovania iba do určitých častí obrazovky, čo v praxi znamená, že všetky fragmenty, ktoré by sa mali zobraziť mimo oblasť ohraničenú obsahom tohto bufferu, sú automaticky zahodené. Pri vytváraní rovných zrkadlových plôch je tento efekt užitočný presne v tom, že po vykreslení objektu do stencil bufferu stačí iba vykresliť identické kópie ostatných

objektov scény v opačnom smere jeho povrchovej normály a tie budú následne pomocou tohto bufferu orezané, aby sa zobrazili iba odrazy viditeľné z bodu pozorovania.

Následne zmiešavanie farieb (blending) slúži na zohľadnenie farby vykresľovaných odrazov v použitom materiáli odrazovej plochy. Dá sa určovať podľa alfa zložky farby, čo je vhodné pre vytváranie priesvitných plôch, no pri zrkadlových plochách stačí, a aj som pri implementácii použil, zmiešavanie podľa farieb.

Mnou implementované zrkadlenie je znázornené pri implementácii v kapitole 4.2.3, avšak pri požadovaní vyššieho realizmu je možné tento prístup optimalizovať rôznymi funkciami pre zmenu osvetlenia na povrchu odrazených objektov, ktoré pre menší rozsah a náplň mojej scény neboli nutné, rovnako ako ani zavedenie orezávacích rovín pre prípad ošetrenia prípadov, kedy by sa niektorý z objektov scény nachádzal za zrkadlovým povrchom.



Obrázok 25: Zrkadlová plocha s vysokou úrovňou realizmu [7].

3.7 Fyzikálne založené modelovacie efekty

Pri návrhu svojej demo aplikácie som sa rozhodol nazrieť aj do problematiky modelovania efektov spojených s dynamickým pohybom objektov tak, ako by približne prebiehal v skutočnom svete, čiže tak, aby zohľadňoval vplyv jednak príťažlivej sily podobnej gravitácii a na druhej strane i vedel realizovať zrážky medzi jednotlivými objektami.

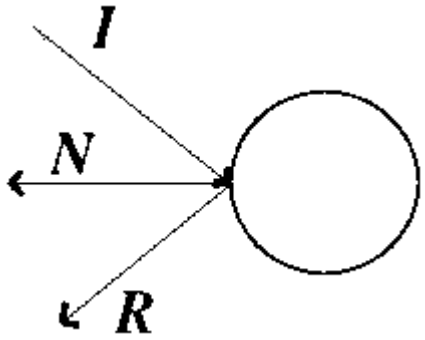
Pre vytvorenie fyzikálneho a kolízneho modelu je v prvom rade nutné, aby každý statický i dynamický objekt mal jasne definovaný priestor, ktorý svojim objemom pokrýva. Pohyblivé objekty musia mať taktiež okrem svojej pozície definovaný aj súčasný vektor rýchlosti, udávajúci ich pozíciu v následnom zábere alebo inak zvolenom časovom kroku.

Vplyv príťažlivej sily sa následne uskutočňuje neustálou dekrementáciou zvislej zložky týchto rýchlostných vektorov a náraz do jednotnej vodorovnej podložky včasnou inverziou tej istej zložky. Na tomto vektore zohľadnený aj istý úbytok pohybovej energie spôsobenej nárazom do podložky realizovaný jeho skrátením.

Modelovanie zrážok medzi ostatnými objektami je v princípe o niečo zložitejšie, avšak pre jednoduchosť som vo svojej aplikácii použil už len objekty v tvare gúľ a valcov so zaguľatenými koncami, aby sa dali pre potreby zrážok vykonať ich dočasné abstrakcie na taktiež guľaté objekty.

Toto zjednodušenie pri implementácii kolízneho modelu pomôže konkrétne v tom, že prípad zrážky dvoch takýchto objektov sa dá následne ľahko určiť zo vzdialenosti medzi stredmi jednotlivých gulí.

Ak by sa v určitom zábere scény mali nachádzať dva guľové objekty s vzdialenosťou ich stredov menšou, než je súčet ich polomerov, je nutné pred vykreslením tohto záberu zrealizovať medzi týmito dva objektami zrážku, a to tak, že sa pre ne vypočíta nový vektor rýchlosti za využitia princípu, že uhol dopadu sa rovná uhlu odrazu.



Obrázok 26: Uhol dopadu sa rovná uhlu odrazu [7].

$$\overline{R} = 2 * (-\bar{I} . \overline{N}) * \overline{N} + \bar{I} \qquad (7)$$

V rovnici (7) je znázornený výpočet odrazeného vektoru **R**, ktorý prevezme úlohu nového rýchlostného vektoru objektu. Do tejto rovnice vstupujú vektory **N** a **I**, z ktorých prvý je normálovým vektorom v bode kolízie a druhý je pôvodný rýchlostný vektor. Obidva tieto vektory musia byť pre správnosť výsledku normalizované, takže pre zachovanie rovnakej rýchlosti objektu, akú mal pred odrazom sa musí dĺžka výsledného vektora ešte nastaviť na dĺžku pôvodného rýchlostného vektora.

Takto navrhnutý fyzikálny model schopný spracovať kolízie objektov je pre správne fungovanie scény úplným minimom a obsahuje množstvo nedostatkov a možných zlepšení, ktoré pre výsledný grafický efekt mojej implementácie neboli nutné, ba dokonca ich uplatnenie by pri veľkom množstve objektov znamenalo výrazné zvýšenie potrebného výpočtového času. Pri malom rozsahu scény navrhovanej pre jej grafický výstup treba skutočne myslieť aj na to, či vynaložená záťaž vyprodukuje patričný vizuálny efekt, takže modifikácie modelu zrážok sú v následujúcich riadkoch rozobrané z čisto teoretického hľadiska.

V prvom rade tento model nedostatočne rieši zrážky jedného objektu s viacerými inými medzi jednotlivými zábermi a prázdny priestor medzi jednotlivými kolidujúcimi objektami považuje pri výpočtoch za zanedbateľný. To je spôsobené práve tým, že korektný model by podľa správnosti mal jednotlivé zrážky riešiť v presných okamihoch a pozíciách objektov scény v čase týchto zrážok, čo vyžaduje delenie časového intervalu, s ktorým sa práve pri práci s modelom scény pracuje, na menšie časti, čo však vyžaduje veľký počet sprievodných výpočtov a pri veľkom počte objektov negatívne ovplyvní plynulý beh programu, ba dokonca môže počítanie a následné vykresľovanie stavu scény v reálnom čase úplne znemožniť.

Druhým nedostatkom použitého modelu je fakt, že dva objekty v pohybe sa pri zrážke podľa skutočných fyzikálnych zákonov neriadia iba podľa pravidla o uhle dopadu a odrazu, ale zohľadňujú pri výpočte aj súčasnú pohybovú energiu, čiže v našom prípade dĺžky vektorov rýchlosti, ktorá sa pri náraze medzi objekty prerozdeľuje a zo zrážky teda nevychádzajú s rovnakou veľkosťou rýchlostného vektora, akú mali pred zrážkou. Dôvodom vypustenia tohto ohľadu z implementácie mojej scény bolo to, že všetky pohyblivé guľové objekty v nej majú rovnakú veľkosť, a teda z fyzikálneho hľadiska i váhu, a ich vzájomné zrážky sú skôr ojedinelé, než prísnym pravidlom diania v scéne, takže aj tento efekt by sa v grafickom výstupe prejavil pre pozorovateľa nepostrehnuteľným spôsobom.

3.8 Hudba do 128kB

Vytvoriť hudbu ako súčasť programu, ktorý sa musí veľkosťou zmestiť do 128kB je samo o sebe rozsiahlou problematikou. V prvom rade treba myslieť na to, že použiť bežné hudobné formáty ako .mp3 a .ogg a prehrávače na ne je v tomto prípade krajne nevhodné a bezstratové formáty typu .wav už vôbec nepripadajú do úvahy. Hlavným bodom sústredenia je teda v tomto úsilí štandard MIDI a k nemu pridružené súbory s koncovkou .mid, pretože sa jedná len o popis úplných nutností ako rozloženie jednotlivých tónov a nástrojov, ktoré majú na daných hudobných kanáloch hrať. K ešte väčšiemu zníženiu použitého diskového priestoru potom existujú rôzne MIDI syntetizátory, ktoré si takto uložené skladby prevádzajú do ďalej zjednodušených foriem.

Príkladom takto používaného syntetizátoru v oblasti dem je [11], ktorý spolu s nástrojmi na syntézu MIDI vstupov z viacerých kanálov vo forme VST zásuvných modulov ponúka aj C++ knižnice na prehrávanie takto vygenerovaných hudobných súborov vnútri programu. Hlavnou výhodou tu je nízka pamäťová náročnosť výstupných súborov a knižníc a takisto malý procesorový čas potrebný na prehrávanie hudby v programe. Pre prípad, že by hudobný súbor aj napriek tomu zaberal príliš veľa miesta, autor v pokynoch k obsluhe píše, že spôsob uloženia dát je optimalizovaný k vykonávaniu kompresie, avšak v mojom prípade je prenechanie polovičnej veľkosti súboru jeho audio zložke prijateľnou možnosťou, takže 64kB sa ukazuje byť viac než benevolentným limitom.

Obrázok 27: Grafické rozhranie VST zásuvného modulu V2 synthesizer.

Ďalšou možnosťou, ktorá stojí za zmienku je knižnica [12], ktorá bola špeciálne navrhnutá k tomu, aby zaberala čo najmenej pamäťového priestoru, nakoľko je celá napísaná v asemblerovskom kóde a na samotné prehrávanie zvukov využíva viacero rôznych možností vrátane DirectSound a OpenAL, kedy má k dispozícii aj možnosti na prevod nutných funkcií z dynamicky prepojovaných knižníc do objektových súborov, ktoré je možné priamo kompilovať pri preklade programu a dosiahnuť tak väčšej prenositeľnosti výsledného súboru.

Knižnica uFMOD slúži na prehrávanie súborov typu .xm, ktoré už sami o sebe ponúkajú výhodu veľmi nízkych pamäťových nárokov. Viac o nej sa však dá dozvedieť na jej domovskej stránke, nakoľko v implementácii svojho projektu sa budem skôr orientovať na prvú z uvedených možností, a teda libv2.

4 Implementácia

Pri implementácii tohto projektu som sa hlavne zameriaval na účelovosť napísaného projektu, a teda všetky uvádzané triedy a metódy často neberú do úvahy možnú znovu-použiteľnosť kódu. Je to spôsobené hlavne snahou o ušetrenie diskového priestoru tým, že objektové triedy nemajú implementované tu nevyužité rozšírenia a funkcie, ako napríklad zaistenie všeobecnosti využitia triedy pracujúcej s VBO, alebo ošetrenia validity vstupných premenných u metód. Celkovo je však beh programu počas celého jeho trvania ošetrený tak, aby prebiehal bez akýchkoľvek postranných problémov, čo považujem za postačujúce pre takýto typ aplikácie.

4.1 Štartovací bod programu

Ako už bolo naznačené pri návrhu aplikácie v prostredí WinAPI, po spustení sa začína zobrazením dialógového okna definovaného v hlavičkovom súbore `resource.h`, ktorý je vygenerovaný z grafického návrhu okna vo Visual Studiu a v aplikácii už má len definovanú funkciu na obsluhu správ. Ďalej aplikácia rieši vytváranie vlákien aplikácie a potom ich spúšťa. Konkrétny význam a funkčnosť týchto vláken popíšem v nasledujúcich pár riadkoch.

Prvé z vlákien prináleží hlavnému oknu aplikácie, a jeho kľúčovou súčasťou je obslužná slučka správ hlavného okna. Ešte pred kontrolou správ v zásobníku a volaniu tejto funkcie však prebieha vytvorenie samotného okna a jeho naplnenie OpenGL scénou vo funkcii, pričom následne je vytvorený objekt manažéra scény podľa triedy. Samotná obslužná slučka teda napokon prebieha tak, že dokým demo aplikácia nedosiahne konca, tak obsluhuje správy a v stave nečinnosti vykresľuje aktuálny stav aktívnej scény.

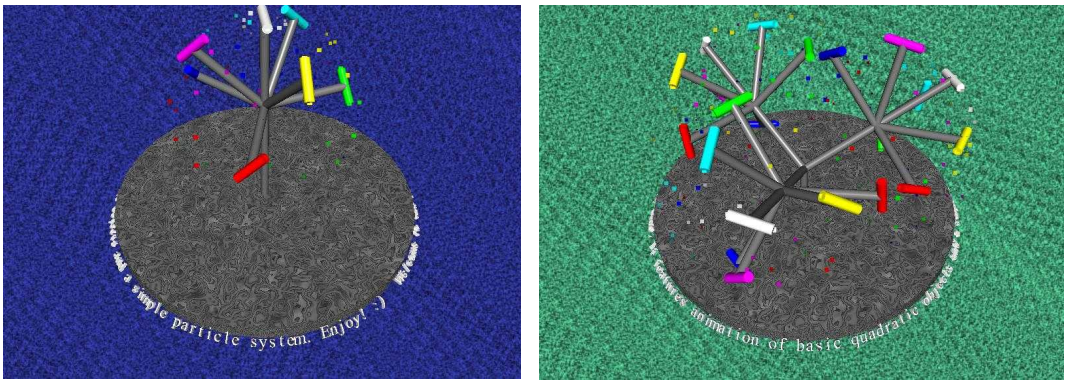
Vedľajším vláknom aplikácie je tzv. vlákno časovača, ktoré prostredníctvom pár príkazov periodicky posiela správy hlavnému oknu aplikácie, ktoré vyvolávajú update dynamických parametrov scén a prípadné prechody medzi scénami.

4.2 Realizácia scén

Samostatné triedy scén sú zdedené zo spoločnej abstraktnej triedy. Sekvencie prvkov v animáciách sa riešia inkrementáciou jej parametrov s tým, že zakaždým sa zvyšuje len jeden a keď dosiahne určitej hodnoty, tak sa prejde na ďalší parameter. Po uplynutí týchto parametrov sa navráti príznak ukončenia scény, ktorý vyvolá buď prechod na ďalšiu scénu alebo ukončenie aplikácie. Funkcie jednotlivých scén predstavím v nasledujúcich sekciách.

4.2.1 Úvodná scéna

Úvodná scéna dema pozostáva s postupného utvárania a pohybu objektov pripomínajúcich konštrukcie podobné stromom. Predovšetkým sa prekryje pozadie štvorcom na výplň a nastaví sa statická kamera scény. Potom sa prejde k vykresleniu spodného kotúča pokrytého povrchom z Perlin šumu a sprievodného textu okolo neho. Ďalej sa rieši už len vykresľovanie jednotlivých komponentov za použitia kvadrikov a tie majú parametre určené v štruktúrach definovaných pomimo triedy.

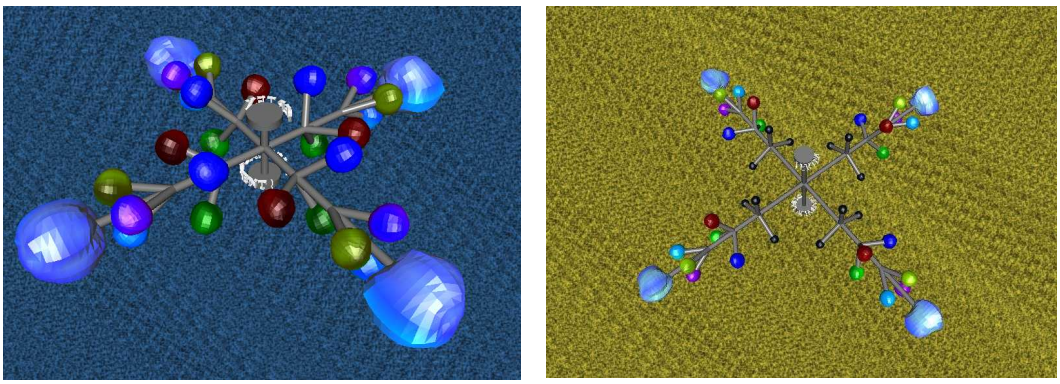


Obrázok 28: Úvodná scéna dema.

4.2.2 Vertex displacement scéna

Táto scéna bola navrhnutá ako demonštrácia vertex displacement efektu v shader programe. Okrem toho bola navrhnutá podobne ako úvodná scéna, teda vykresľované objekty sú vo forme kvadrikov s tým rozdielom, že namiesto výpočtu textúr sa tu Perlin šum využíva na počítanie vertex displacementu určitých guľových objektov.

Scéna začína tam, kde predchádzajúca skončila. Rovnako ako úvodná scéna postupuje pri vykresľovaní aktuálnych stavov scény nastavením pozadia a kamery a kreslí základ scény s popismi. Ďalej pokračuje vykresľovaním objektov v tvare tyčí vychádzajúcich zo stredu scény a zakončených guľami, na ktorých je aplikovaný displacement efekt. Postupom času sa pôvodné tyče rozširujú za postupného odďaľovania kamery a pridávajú sa k nim postupne ďalšie ako experiment s tým, koľko sa ich na obrazovku zmestí bez toho, že by tým bol negatívne poznačený výkon. Celý priebeh je sprevádzaný kyvadlovým pohybom objektov scény. Scéna končí postupným vyblednutím farieb (fade efekt), čím necháva po sebe prázdny priestor.



Obrázok 29: Vertex displacement scéna dema.

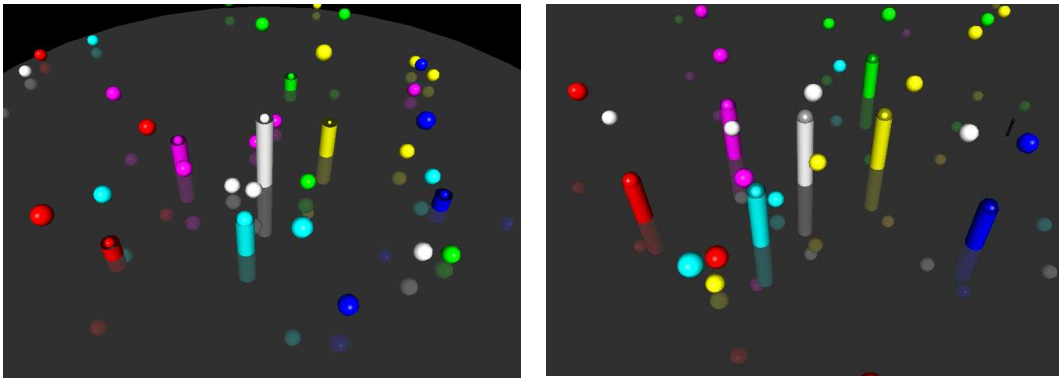
4.2.3 Fyzikálna scéna

Po predchádzajúcich dvoch scéna nasleduje scéna zameraná na použitie fyzikálnych modelovacích efektov, konkrétne kladie hlavný dôraz na nárazy guľových objektov s prostredím a vytváranie zrkadlových obrazov prostredníctvom stencil bufferu. Kamera je v tejto scéne plne dynamická a počas priebehu scény sa otáča okolo zvislej osi, aby bolo vidno, čo sa na scéne deje, z viacerých uhlov pohľadu.

Rovnako ako predchádzajúce scény je i táto tvorená kvadrikmi a figurujú tu konkrétne 3 funkčné druhy – podložka, nad ktorou prebieha dianie v scéne, a takisto ho zrkadlovo odráža; gule riadené vlastnými vektormi rýchlostí a neustále priťahované smerom k podložke, pričom reagujú na nárazy s ostatnými objektami v scéne; a valce z ktorých priebežne vychádzajú nové gule, pre jednoduchosť výpočtovými operáciami s nárazmi tiež zakončené priesvitnými statickými guľami.

Poloha a farba takto vykresľovaných valcov je vopred daná pri volaní konštruktora objektu a za behu scény sa len postupne inkrementujú dĺžky a šírky jednotlivých valcov, pričom tie, čo dosiahli svojej konečnej veľkosti začínajú produkovať guľové objekty a púšťať ich do scény s počiatočnými vektormi rýchlosti vygenerovaných podľa výstupu goniometrických funkcií s argumentami poradového čísla práve vytvorenej gule. Zvislá zložka rýchlosti je zasa počítaná podľa veľkosti daného valca – čím je valec nižší, tým je zvislá zložka vyššia.

Vykresľovanie aktuálneho záberu po nastavení kamery prebieha nasledovne – podložka sa vykresľuje do stencil bufferu, čo sa okamžite potom využíva na vykreslenie odrazu modelu scény, pričom po jej dokončení sa využitie stencil bufferu vypne a zapne sa mód zmiešavania farieb, v ktorom sa prostredníctvom metódy vykreslí podložka a prekryje tak už vykreslený odraz modelu. Ten sa nakoniec taktiež vykreslí opätovným volaním príslušnej metódy, čo zaistí jeho vykreslenie nad podložkou. Scéna končí oddialením kamery do prázdna.



Obrázok 30: Fyzikálna scéna dema.

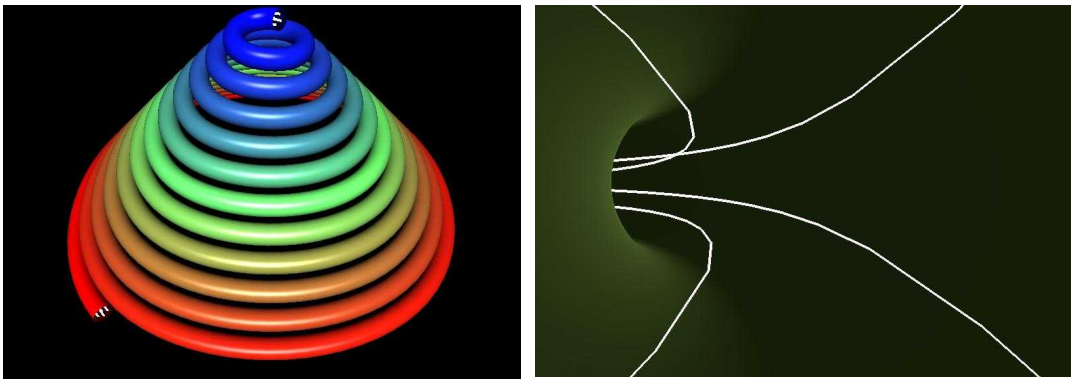
4.2.4 Spline scéna

V poslednej scéne ide o praktické znázornenie využitia spline kriviek pri modelovaní objektov a ovládaní pohybu kamery. Konkrétne sa jedná o jediný B-Spline, ktorého kontrolné body sú inicializované v rámci volania konštruktora objektu a následne sú podľa nich vypočítané súradnice bodov samotnej krivky. Všetky tieto body sú ukladané v príslušných poliach.

Za behu tejto funkcie sú najprv tieto body postupne obkolesené narastajúcim tunelom v tvare špirály, ktorého vrcholy sa rátajú podľa parametrického vyjadrenia kružnice za pomoci vektora smerujúceho od jedného bodu krivky k druhému a dvoch vektorov naň kolmých. Pre každý segment krivky sa takto vyrátané vrcholy predávajú do VBO, kde sa následne vykresľujú, pričom sa dbá na to, aby aj polia hodnôt potrebných pre vertex blending boli naplnené správnymi hodnotami.

Po vykreslení celej krivky nasleduje v tejto scéne naklonenie kamery do polohy nad špirálu a prechod jej vnútrom takisto daný súradnými bodmi krivky. Vnútorný priestor špirály je vyplnený jednoduchými čiarami natáčanými v smere hodinových ručičiek na lepšie zvýraznenie priestoru, ktorý je kamerou vidno.

Napokon je táto scéna zakončená priblížením správy „The End“ a jej následným vyblednutím a ukončením programu.



Obrázok 31: Spline scéna dema.

4.3 Vedľajšie triedy

Okrem triedy pre objekt manažéra scény a jednotlivé scény sú v programe implementované aj niektoré ďalšie triedy.

V prvom rade sa jedná o triedu `TCoord` určenú pre zjednodušenie práce so súborom koordinátov v 3D priestore, čo môže byť využité ako pre samotné body, tak pre rôzne vektory, ktoré tu takisto majú definované aj operácie špecifické pre prácu s nimi ako normalizácia vektora, skalárny a vektorový súčin a iné.

Pre beh programu však dôležitejšiu úlohu hrá trieda `TExt`, ktorá rieši manuálne načítanie tiel funkcií figurujúcich ako rozšírenia rozhrania a sú nezbytnou súčasťou metód v objektových triedach pre shader program a VBO. Tie potom tieto funkcie volajú ako metódy objektu tejto triedy, ktorej objekt je rovnako ako aj objekt shader programu a VBO vytvorený za behu programu iba raz, a teda v tele konštruktora objektu manažéra scény.

4.3.1 Shader program

Implementovaná trieda `TShader` zapuzdruje všetku prácu spojenú so shader programom od kompilácie jednotlivých zložiek a jeho linkovanie až po nastavovanie uniform premenných a predávanie lokácie vertex atribútov. Funkcie pre kompiláciu a linkovanie sú volané už samotnom konštruktore objektu tejto triedy a akékoľvek chyby v týchto procesoch sú ukladané do súboru log.txt. V rámci šetrenia pamäťových prostriedkov je tiež maximálny počet výstupných vrcholov geometry shadera nastavený na 3, keďže viac než spracované trojuholníky sa na výstupe v tomto programe neočakávajú.

Čo sa týka jednotlivých shaderov, tak z dôvodu eliminácie vonkajších zdrojov sú implementované ako súvislé reťazce v hlavičkovom súbore `shaderdefs.h`, avšak rovnako ako aj zvyšný kód programu sú vhodne okomentované v priestore medzi jednotlivými funkčnými celkami. Stručne však popíšem ich funkciu aj v následujúcich riadkoch.

V tele implementovaného vertex shadera sa nachádzajú funkcie potrebné pre výpočet hodnôt Perlin šumu, ktoré sú tu využívané pri vertex displacement efekte. V main funkcii tohto shadera sú potom vykonané hlavne výpočty varying premenných súvisiacich s osvetlením a potom podľa zopnutia spínačov pre vertex displacement a vertex blending sú vykonané príslušné operácie súvisiace s týmito efektami, zakaždým zakončené transformáciou pozície vrcholu do súradnej sústavy modelu.

Geometry shader následne plní špecifickú úlohu korekcie normálových vrcholov povrchu po vykonaní vertex displacementu tým, že ich nahradí normálou jednotnou pre celé geometrické primitívum, teda v tomto prípade trojuholník. Výsledné osvetlenie takýchto objektov je teda potom

menej prirodzené a sú zobrazované ako hranaté, no z hľadiska potrebného výpočtového času sa jedná o výhodné riešenie.

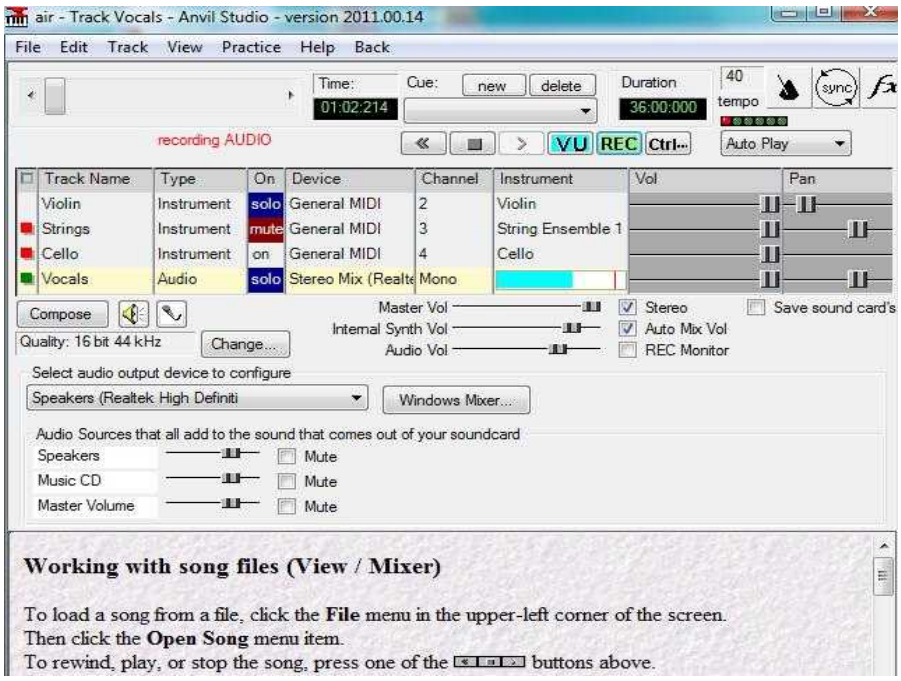
Napokon fragment shader rovnako ako aj vertex shader obsahuje vo svojom tele funkcie súvisiace s Perlin šumom, avšak ich používa na počítanie šumových textúr objektov. Vo svojej main funkcii potom normalizuje prichádzajúce vektory súvisiace s osvetlením objektu a pomocou nich ráta výslednú farbu fragmentu, pričom pri zapnutom prepínači šumu zakomponuje jeho hodnotu do difúznej zložky farby. Takisto vo výslednej farbe zohľadňuje aj jej požadované vyblednutie používané ako efekt prechodu medzi niektorými scénami.

4.3.2 Vertex Buffer Object

Trieda `TVBO` zapuzdrujúca prácu s Vertex Buffer Objektom používaným na vykresľovanie objektov v poslednej scéne dema, bola implementovaná čisto pre tento konkrétny účel, a teda má v sebe definované iba tie polia súradníc, ktoré sa priamo v tejto scéne využívajú, a žiadne iné. Takisto čiary napĺňajúce vnútro špirály v tejto scéne majú svoje vlastné dynamicky vytvárané polia na naplnenie príslušnými hodnotami a samostatný buffer, do ktorého sa ukladajú pred ich vykreslením.

4.4 Hudba dema

Ako už bolo objasnené v návrhovej časti, na tvorbu audiálnej časti svojho projektu som využil nástroj [11], ktorému som za vstup vložil vlastnoručne vytvorený MIDI súbor v programe [13]. Pri rozmiestňovaní melódií* a voľbe nástrojov na ich prehrávanie som sa orientoval na to, aby výsledná skladba obsahovo sedela k už hotovej animácii a V2 syntetizátorom vytvorený súbor s koncovkou v2m som následne predal do svojho programu prostredníctvom hexadecimálneho výpisu súboru do dátovej štruktúry v súbore *track.h*.



Obrázok 32: Grafické rozhranie programu Anvil Studio 2011.

* Za samotných melódiami stojí kolektív autorov Andreas & Savas Oulassoglou (aaristid@logos.cy.net).

4.5 Výsledky aplikácie

Výsledná aplikácia vrátane hudby zaberá v konečnom štádiu po samotnej kompilácii niečo málo nad 100kB, čím spĺňa stanovený veľkostný limit aj s rezervami. Následne čo sa týka výkonu, tak aplikácia bola testovaná pod viacerými zostavami ovplývajúcimi podporou OpenGL2.1, konkrétne týmito:

- Intel Core 2 Duo @ 2,00 GHz; 3GB RAM; NVIDIA GeForce 8400M GS 128MB
- Intel Core 2 Duo @ 2,17 GHz; 3GB RAM; NVIDIA GeForce 9300M GS 512MB
- Intel Core 2 Duo T9400 @ 2,53Ghz; 4GB RAM; NVIDIA GeForce 9650M GT 1024MB.

Vo všetkých týchto prípadoch bol zaznamenaný plynulý beh dema počas celej dĺžky jeho trvania a pamäťové nároky aplikácie sa celý čas pohybovali v rozmedzí od 30MB do 40MB, čo vylúčilo nutnosť nevyhnutných optimalizácií pre zlepšenie výkonu, avšak z podrobnejšieho prieskumu framerate hodnôt počas jeho behu sa o použitých technikách dajú o ich výkone vyvodiť oveľa smerodajnejšie dôsledky.

Najväčšiu záťaž pre výpočtový čas predstavovala aj napriek použitiu VBO pre vykresľovanie posledná scéna, čo bolo hlavne spôsobené nutnosťou opätovného rátania objektových vrcholov pri každom vykresľovaní scény, ktorý sa jednak medzi jednotlivými zábermi dynamicky menil, a na druhej strane vyžadoval veľké množstvo číselných údajov potrebných na zobrazenie každého jedného zo segmentov, čo značne znemožňovalo akúkoľvek snahu o vyváženie záťaže medzi grafickú kartu a procesor.

Čo sa však týka prvých dvoch scén, teda tých, čo v sebe zahrňovali vykresľovanie kvadrikov v okamžitom móde a počítanie hodnôt šumových funkcií v reálnom čase, v požadovanom rozsahu túto záťaž zvládali lepšie než scéna posledná. Dôvodom bola pomerne nízka zložitosť celkových modelov použitých v tejto scéne, z čoho vyplýva, že pri súčasných možnostiach dostupných pre osobné počítače, ktoré sú schopné podporovať potrebné rozšírenia, je aj takéto použitie shader programov a iných možností rozhrania OpenGL, kedy mnohé z nich sú považované za zastaralé a nežiadúce, vhodným riešením návrhu jednoduchších animovaných scén.

Pre projekty menšieho rozsahu, ako konkrétne môj s veľkostným limitom 128kB, takéto tvrdenie platí obzvlášť, pretože zachytiť komplikované modely a scény by na takomto obmedzenom priestore bolo samo o sebe značným implementačným problémom, a takisto som presvedčený, že aj pri interpretácii slova „demo“ ako „demonštrácia“ som možnosti rozhrania OpenGL demonštroval v rámci troch minút celkom úspešne.

5 Záver

Minimálnym požiadavkom vo výstupe tohto projektu bola trojminútová animácia s hudbou v rámci jediného spustiteľného súboru pod 128kB, čo je cieľ, ktorý sa mi podarilo splniť a dokonca si z toho odniesť i veľa užitočných poznatkov o práci v OpenGL prostredí, ako z hľadiska návrhu, tak aj implementácie v C++.

Náplňou sa táto práca však dotýka predovšetkým nevyhnutných základov nutných pre tvorbu logicky konzistentných 3D scén a neuplatňuje žiadne pokročilé techniky a optimalizácie ako v rámci osvetlenia, tak i samotných objektov, čo necháva veľa priestoru pre môj budúci prieskum možností tohto rozhrania a takisto aj vytvorené demo by sa dalo vo veľa ohľadoch rozviesť.

Menované zmeny by sa veľmi ľahko mohli v súčasne implementovaných scénach ukázať ako nevýrazné a zmeny oproti súčasnému stavu by dokonca mohli byť úplne nepostrehnuteľné, no ako už bolo spomínané, tak použitý osvetľovací model je možné ďalej rozvinúť tak, aby vykonával aj tvorbu tieňov a slabnutie svetla podľa vzdialenosti od zdroja. Vytvorené scény majú však abstraktný koncept namiesto toho, aby sa usilovali napodobňovať prostredia a javy z reálneho sveta, takže takéto zmeny by výsledný efekt mohli poznačiť skôr v negatívnom smere než pozitívnom.

Takisto aj častice môžu byť modelované zložitejším spôsobom, než je použité v prvej scéne dema, no to by sa zasa mohlo negatívne poznačiť na plynulosti behu aplikácie. Od toho existuje veľké množstvo optimalizácií použitého kódu a techník, ktoré síce pre plynulosť behu súčasných výstupov neboli nutné, ale v možnom ďalšom vývoji tejto aplikácie by napríklad povrchy z Perlin šumu nemuseli byť generované v reálnom čase, ale uložené do dátovej štruktúry a za behu animácie volané priamo z nej, čím by sa výkon zníži na úkor pamäťových nárokov.

Napokon aj samotné scény mohli byť koncipované úplne inak a aj ich počet by sa dal neustále zvyšovať, čím by bola pokrytá napríklad oblasť modelovacích techník založených na procedurálnych výpočtoch vrcholových koordinátov objektu za pomoci iných už vopred daných a znázorniť tak proces postupnej transformácie kocky na úplne iný objekt ako guľa.

Možností, o ktorých by sa dalo takýmto spôsobom písať je v oblasti počítačovej grafiky priam neobmedzené množstvo, avšak so svojou prácou v tomto poli som spokojný a som rád, že som mohol aspoň v takomto rozsahu nazrieť do veľmi dôležitej časti z nich a implementovať ju do ucelenej formy, ktorá by sa z istého uhla pohľadu dala nazvať drobným umeleckým dielom.

Literatura

[1] Kršek, P. a Španěl, M.: *Základy počítačové grafiky*. Brno: Študijná opora FIT VUT v Brne, 2008.

[2] Schreiner, D.: *OpenGL programming guide : the official guide to learning OpenGL, versions 3.0 and 3.1.* Boston: Pearson Education, 2010.

[3] Rost, R. J.: *OpenGL shading language - 3rd ed*. Boston: Pearson Education, 2010.

[4] Randima, F.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Boston: Pearson Education, 2004.

[5] Wright, R. S. Jr. a Lipchak B.: *OpenGL SuperBible, Third Edition.* Sams Publishing, 2004.

[6] Demoscene, The: *demoscene.info*. [online]. URL: http://www.demoscene.info/ (Máj 2011).

[7] Molofee, J.: *NeHe Productions.* [online]. URL: http://nehe.gamedev.net/ (Máj 2011).

[8] Inspire: *kindercrasher*. [online]. URL: http://pouet.net/prod.php?which=50526 (Máj 2011).

[9] Fernandes, A. R.: *OpenGL @ Lighthouse 3D – A Resource for Programmers.* [online]. URL: http://www.lighthouse3d.com/opengl/glsl/ (Máj 2011).

[10] Guinot, J.: *oZone3D.Net – GLSL Programming Tutorials*. [online]. URL: http://www.ozone3d.net/tutorials/index_glsl.php (Máj 2011).

[11] Hinrichs, T.: *V2 synthesizer system.* [online]. URL: http://www.1337haxorz.de/ (Máj 2011).

[12] Asterix and Quantum: *uFMOD*. [online]. URL: http://ufmod.sourceforge.net/ (Máj 2011).

[13] Willow Software: *Anvil Studio.* [online]. URL: http://www.anvilstudio.com/ (Máj 2011).

Seznam příloh

Příloha 1. CD s .exe súborom, zdrojovými kódmi, programovou dokumentáciou a záznamom dema.
Příloha 2. Plagát