VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

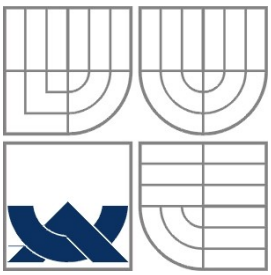
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

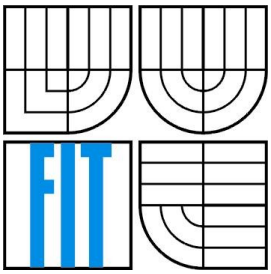
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAN PINTER
AUTHOR

BRNO 2007



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 — JAN PINTER
AUTHOR

VEDOUCÍ PRÁCE — Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2007

Zadání

Seznamte se s fenoménem grafického intra s omezenou velikostí.

1. Prostudujte knihovnu OpenGL a její nadstavby.
2. Popište vybrané techniky použitelné v grafickém intru s omezenou velikostí.
3. Implementujte grafické intro s použitím OpenGL, aby velikost spustitelné verze nepřesáhla 64kB.
4. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; prezentujte projekt plakátkem.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačnich technologií Vysokého učení technického v Brně.

Abstrakt

Práce se zabývá problematikou tvorby grafického intra (dema) s velikostí omezenou hranicí 64 kB. Rozebírá a popisuje problémy spojené s efektivním využitím poskytnutého minimálního prostoru, prezentuje základní techniky běžně používané u programů uvedeného charakteru. Také popisuje způsob realizace některých zajímavých grafických efektů a prvků, jejich možné optimalizace a vizuální vylepšení.

Klíčová slova

demo, intro, OpenGL, Catmull-Clark, optimalizace, syntetizátor, delta modulace, reflexe, radiální rozmazání, Perlinův šum, částicový systém

Abstract

This paper deals with graphics intro (demo) 64kb creation issues. It analyses and describes problems involving effective use of minimum data space assigned and presents basic techniques used with those specific types of programs as mentioned above. It also describes a way of specific, interesting graphic effects and elements realization, their possible optimization and visual improvement.

Keywords

demo, intro, OpenGL, Catmull-Clark, optimalization, synthesizer, delta modulation, reflection, radial blur, Perlin noise, particle system

Citace

Jan Pinter: Grafické intro 64 kB s použitím OpenGL, bakalářská práce, Brno, FIT VUT v Brně, 2007

Grafické intro 64 kB s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. A. Herouta, Ph.D, a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal poznatky při zpracovávání této práce.

……
Jan Pinter
15.5.2007

Poděkování

Dovoluji si vyslovit poděkování Ing. A. Heroutovi, Ph.D., který mi jako vedoucí projektu byl ochoten věnovat čas a poskytl nejednu cennou radu. Dále bych rád poděkoval A. Grygarovi za poskytnutí trojrozměrného modelu stromu, L. Kulhavé za vlastní báseň a L. Petrové za její překlad a přebásnění do anglického jazyka.

Obsah

1 Úvod

Fenomén grafického intra (dema) je jedním z nejstarších aspektů kultury digitálního umění. Původně malé a jednoduché programy sloužící jako drobné osobní prezentace počítačových pirátů postupem času prodělaly neuvěřitelnou evoluci a v současnosti kulturně sjednocují desetitisíce lidí.

Demo slouží převážně jako prostředek prezentace vlastních schopností, popř. schopností grafika, zvukaře apod. Existuje nespočet kategorií, které tvorbu dem různě ovlivňují. Jsou dema, které běží pouze v terminálovém okně, jsou dema, které slouží jako pozvánky na různé akce pořádané lidmi z demoscény (síť osob s aktivním zájmem o tuto formu digitálního umění).

Existují také dema, jejichž hlavním kritériem je limit velikosti. Tato práce se pokusí nastínit základní techniky a postupy při tvorbě dema s velikostí omezenou hranicí 64 kB. Vzhledem k doporučenému rozsahu nejsou v tomto dokumentu popsány podrobně veškeré implementační podrobnosti (vytvoření dialogového okna pomocí WinAPI, jednoduchá interpolace po Bézierově křivce atd.). Namísto toho jsou zde podrobněji rozvedeny nejčastější techniky používané při tvorbě velmi malých programů, od nastavení kompilátoru až po volbu vhodné reprezentace ukládaných dat a jejich minimalistického vyjádření.

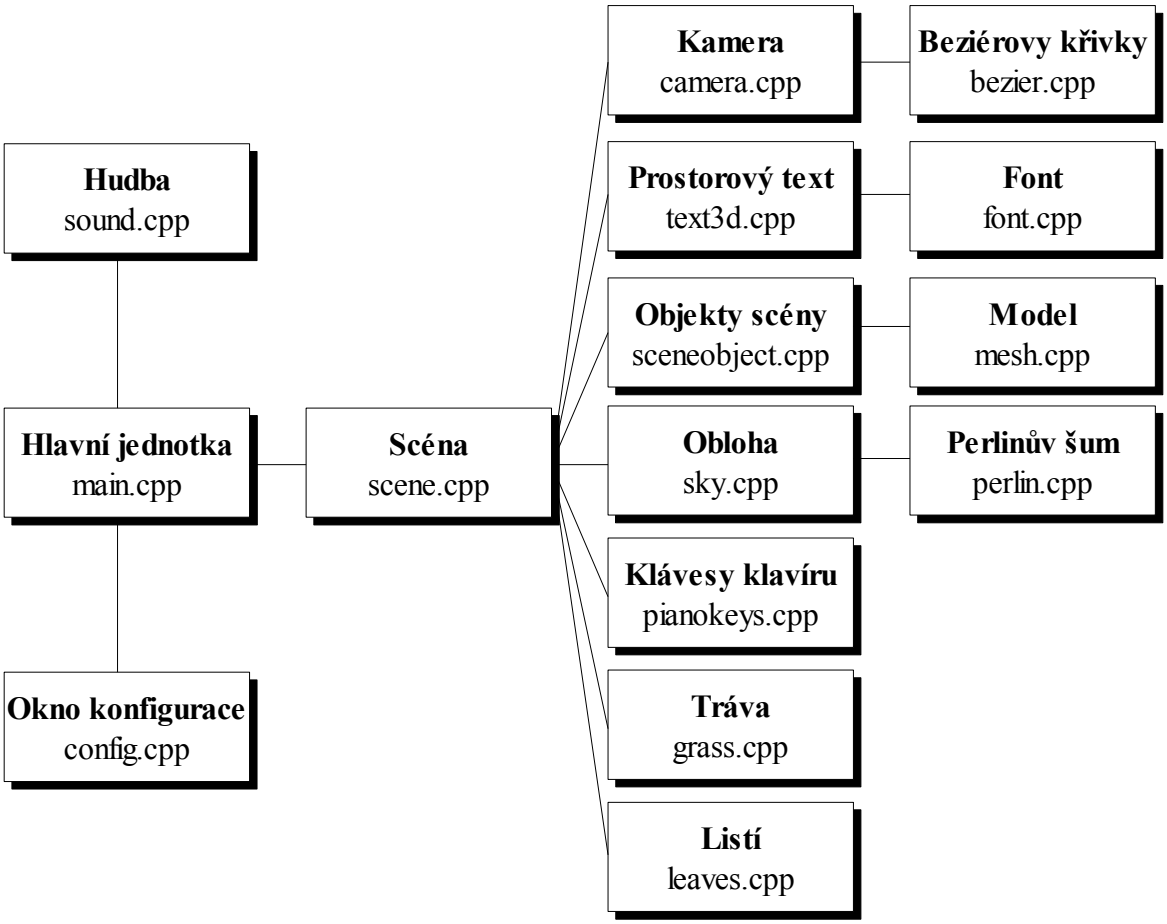
Další kapitoly popisují základní způsoby renderování zajímavých přírodních entit (tráva, listí, obloha) a jejich rozpohybování. V závěru je proveden drobný test nejčastěji používaných komprimačních utilit pro tento druh aplikací.

2 Architektura

Ačkoliv se na první pohled může zdát, že relativně malý prostor 64 kB neposkytuje dostatek možností k realizaci komplexnější programové struktury, opak je pravdou. V případě, že chceme vytvořit dostatečně rozmanité scény, může nám rozdělení programu na dílčí jednotky usnadnit spoustu práce.

Celá vnitřní architektura intra je proto rozdělena na několik částí, které spolu úzce spolupracují a dohromady vytvářejí kompletní dynamickou scénu. Na následujícím obrázku 2.1 je stručně znázorněno propojení jednotlivých částí v rámci celého systému. Některé části úmyslně chybí, jelikož jejich funkcí je pouze zastřešení určitých implementačních detailů.

V následujících kapitolách budou jednotlivé důležité sekce probrány podrobněji, tato kapitola poskytuje pouze pohled na základní schéma programu za účelem zvýšení přehlednosti a snažšího pochopení vazeb popsaných na dalších stranách.



Obrázek 2.1: Vazby jednotlivých modulů

3 Nastavení kompilátoru

Prvním a velmi důležitým krokem při návrhu jakéhokoliv programu s omezenou velikostí by mělo být vhodné nastavení parametrů kompilátoru. Vhodnou volbou přepínačů můžeme radikálně snížit výchozí velikost celého programu ještě před jeho samotnou komprimací.

Následující popis přepínačů se vztahuje pouze pro prostředí Microsoft Visual C++ 2005. Není zaručeno, že u jiných vývojových prostředí nebo kompilátorů bude totožný. Lze ovšem předpokládat, že vhodné alternativy budou existovat pod jiným označením.

3.1 Standardní běhová knihovna jazyka C

3.1.1 Vstupní programová funkce

Ke všem programům v jazyce C/C++ je implicitně připojena standardní běhová knihovna („C Runtime“, dále jen CRT). Její použití automaticky zvyšuje velikost výsledného souboru zhruba o 40 kB. Úkolem CRT knihovny je poskytnout základní funkce pro výstup textu, zpracování řetězců, obsluhu paměti apod.

Jelikož celé intro běží v jednoduchém okně s OpenGL kontextem a většinu výše zmíněných funkcí vůbec nevyužívá, je lepší CRT knihovnu vůbec nepoužívat (už jen kvůli její abnormální velikosti vůči celému projektu). Přepínačem „/NODEFAULTLIB“ u nastavení linkeru tedy zakážeme používání veškerých implicitních knihoven. Naneštěstí s sebou toto řešení přináší určité komplikace, které znemožňují úspěšnou kompilaci v případě, že nejsou náležitě ošetřeny.

Prvním problémem, který se při odstranění CRT knihovny vyskytne, je nepřítomnost vstupní spouštěcí procedury. Není totiž úplnou pravdou, že činnost programu začíná funkcí **`main()`** (popř. **`WinMain()`**). Ta je totiž volána až po úspěšné inicializaci běhové knihovny. V našem případě, kdy jsme běhovou knihovnu odstranili, jsme zároveň ztratili i onen prvotní vstupní programový bod. Řešení je snadné, stačí dodatečně definovat chybějící vstupní funkci v následujícím tvaru:

```
int WinMainCRTStartup(HINSTANCE hInstance)
{ … }
```

Nyní můžeme bez problémů vynechat implementace jinak povinných funkcí **`main()`** nebo **`WinMain()`** a pro tyto účely využít námi vytvořenou inicializační funkci CRT knihovny. Ve své původní implementaci totiž sama tyto povinné funkce volá. V našem případě, kdy není potřeba CRT knihovnu inicializovat, ji můžeme využít jako regulérní funkci hlavní.

3.1.2 Chybějící matematické funkce

Další komplikací vzniklou odstraněním CRT knihovny jsou chybějící matematické funkce a funkce pro číselnou typovou koverzi. Je zřejmé, že při práci s 3D grafikou a s různými efekty se jejich použití nevyhneme.

Nezbývá než postupovat podobně jako u předchozího problému a pro všechny používáné funkce vytvořit vlastní implementace. Jako příklad je možno uvést jednoduchý kód funkce **`sin()`**:

```
float sin(float v)
{
    __asm fld v;
    __asm fsin;
    __asm fstp v;
    return v;
}
```

Analogicky můžeme vytvořit i funkce zbývající (nahrazením instrukce `fsin` jinou). Jmenovitě např. kosinus nebo druhá odmocnina čísla.

Poslední chybějící funkcí pro pohodlnou práci s čísly je funkce zajišťující typovou konverzi mezi desetinným a celým číslem. Za normálních okolností není „viditelná“ a její volání probíhá automaticky při každém uvedeném převodu. Implementace je podobná jako u matematických funkcí:

```
int _ftol2(float f)
{
    volatile int result;
    __asm fistp result;
    return result;
}
```

3.1.3 Správa paměti

Poslední výraznou ztrátou oproti CRT knihovně je nepřítomnost standardních alokačních a dealokačních paměťových funkcí malloc a free. V tomto případě je nutno použít alternativy poskytované rozhraním WinAPI, tedy funkce **`GlobalAlloc()`** a **`GlobalFree()`** (více na [1]).

Pro příklad můžeme uvést způsob implementace náhrady za funkci **`malloc()`**:

```
void *malloc(unsigned int size)
{
    return (void *)GlobalAlloc(GMEM_FIXED, size);
}
```

3.2 Speciální přepínače

3.2.1 Přepínač „/QIfist“

V běžných případech, kdy přepínač „/QIfist“ nepoužíváme, překladač vkládá volání na konverzní funkci **`_ftol()`** vždy, když náš kód vyžaduje konverzi čísla s plovoucí řádovou čárkou na číslo celé (viz 3.1.1.2). Funkce je zodpovědná za korektní konverzi podle standardu IEEE (Institute of Electrical and Electronics Engineers). Toto je nejenže pomalý proces, ale zároveň činí náš kód závislým na CRT knihovně, tudíž je v našem zájmu se mu vyhnout.

Tím, že poskytneme překladači vlastní verzi funkce **`_ftol()`** můžeme získat vyšší rychlost a menší velikost. V případě, že se rozhodneme použít alternativu zmíněnou v kapitole 3.1.1.2, musíme dbát na to, aby konverze probíhala s korektním nastavením režimu zaokrouhlování jednotky FPU. Původní funkce obsažená v knihovně CRT provádí toto nastavení u každého volání. Pokud máme jistotu, že zaokrouhlovací režim zůstane po celý běh programu stejný, postačí nám jednorázové nastavení při startu následujícím krátkým kódem:

```
static unsigned short ctrl = 0x177F;
__asm fldcw ctrl;
```

Instrukce `fldcw` zajistí správné nastavení řídicího registru FPU. Pro podrobnější popis jejich jednotlivých významů viz [2].

3.2.2 Přepínač „/Gr“

Další použitou optimalizací je změna běžné konvence předávání parametrů. Při standardním nastavení kompilátoru jsou parametry předávány skrze zásobník. V případě použití přepínače „/Gr“ jsou parametry předávány přes registry procesoru, kdykoliv je to možné. Toto nastavení opět činí kód nejen menší, ale i rychlejší.

3.2.3 Přepínač „/GX-“

Třetím přepínačem vypínáme obsluhu vyjímek. V případě tohoto projektu, kdy jsou v kódu použity pouze konstrukce jazyka C a nevyskytuje se objektově orientovaného programování, není třeba vyjímek využívat. V rámci radikálně omezeného limitu velikosti bohatě postačí jednoduchá obsluha chyb na bázi návratových hodnot funkcí.

3.2.4 Přepínač „/Gs“

Podobně jako u předchozího přepínače zajistíme tímto vypnutí kontroly „nedbalosti“ při práci se zásobníkem. S dobrým programovým konceptem a kódem se není třeba obávat následných komplikací.

3.2.5 Sloučení a zarovnání sekcí

Typický spustitelný soubor pro platformu Windows je rozdělen na tzv. sekce. Těchto sekcí je několik typů, každá s jiným účelem. Prvním typem je sekce .data, která obsahuje např. řetězce, konstanty, tabulky atd. Dalšími typy jsou sekce .text (kód programu), .rdata (data pouze pro čtení) a .reloc (tabulka externích symbolů). Při standardním nastavení jsou všechny sekce zarovnány na hranici 4 kB, což je v našem případě velké plýtvání. Přepínačem „/opt:nowin98“ zajistíme zarovnání na hranici 512 bytů. Použitím následujících direktiv preprocesoru můžeme navíc některé sekce sloučit:

```
#pragma comment(linker, "/MERGE:.rdata=.data")
#pragma comment(linker, "/MERGE:.text=.data")
```

Více informací lze nálezt na [5].

3.2.6 Standardní optimalizační volby a doporučení

Samozřejmostí je použití přepínačů pro optimalizaci velikosti („/Os“) a optimalizaci rychlosti („/Og“, „/Oa“). Zvláštním doporučením je snaha o používání pouze datového typu `float` v případě práce s čísly s pohyblivou řádovou čárkou ve zdrojových kódech. Vždy bychom měli uvádět postfixovou značku „f“ za desetinným číslem (např. `1.5f`), takže překladač dané číslo uloží pouze na 4 byty, na rozdíl od 8 bytů v případě typu `double`, který je implicitně použit v případě neuvedení postfixové značky.

4 Trojrozměrné modely

4.1 Vnitřní reprezentace

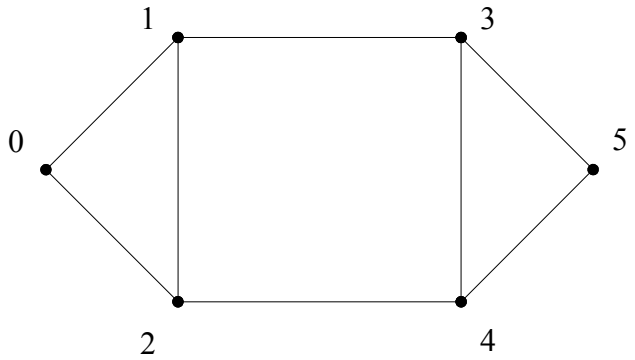
Pokud ve svém programu chceme v rámci statických dat uchovat větší počet trojrozměrných modelů (což je i případ mého dema), musíme zvolit co možná nejvhodnější datovou reprezentaci.

Hlavní nevýhodou trojrozměrných dat je nepřítomnost „přebytečných“ údajů. Vždy musíme obsáhnout kompletní polygonovou strukturu, ztrátová komprese nepřichází v úvahu. Z této skutečnosti logicky vyplývá, že hlavním faktorem ovlivňujícím celkovou datovou velikost modelu je jeho komplexnost. Tuto nevýhodu řeší různé metody dělení ploch (surface subdivision), v našem případě konkrétně metoda Catmull-Clark podrobněji rozebraná v kapitole 4.2.

Protože výsledná binární podoba programu je nadále zpracována speciální komprimační utilitou, je vhodné „ušetřit“ komprimačnímu algoritmu práci a veškerá statická modelová data uložit v podobě, která pro něj bude snadněji zpracovatelná. Nejsnadněji komprimovatelným typem dat jsou samozřejmě dlouhé bloky stejných hodnot, popř. bloky několika málo neustále se opakujících hodnot. Této skutečnosti se budeme snažit využít při návrhu vhodného formátu uložení dat.

4.1.1 Základní reprezentace

V následujících příkladech a ukázkách budeme vycházet z jednoduchého modelu na obázku 4.1. Obsahuje dva trojúhelníky, jeden čtyřúhelník (quad), a šest vrcholů (vertexů) z nichž některé jsou mezi jednotlivými polygony navzájem sdílené.



Obrázek 4.1: Jednoduchý model

Tabulka 4.2 znázorňuje souhrn informací potřebných ke kompletní rekonstrukci celého modelu. Ačkoliv by bylo možné veškeré plochy definovat pouze pomocí trojúhelníkové sítě (čtyřúhelníky nahradit dvojicemi trojúhelníků), použitím čtyřúhelníků výrazně snížíme výskyt zbytečně se opakujících dat (dva trojúhelníky potřebují k popisu 6 indexů, čtyřúhelník pouze 4). Tato reprezentace navíc zajistí lepší výsledky při použití metody Catmull-Clark.

Při pohledu na jednotlivé údaje v tabulce 4.2 zjistíme, že není jasné, jaké datové typy jsou pro jednotlivé položky použity. Můžeme předpokládat, že počet vertexů pravděpodobně nepřesáhne maximum typu `unsigned short` (65535). Totéž platí o počtu polygonů a čtyřúhelníků. Vertexové souřadnice v prostoru používají jako svůj výchozí formát typ `float` (4 byty), což dává celkovou velikost 12 bytů pro jeden vertex. Na indexy polygonů nám postačí typ shodný s typem pro celkový

počet vertexů, což je `unsigned short` (2 byty). Výsledná datová struktura bude v paměti uložena způsoben naznačeným v tabulce 4.3.

Počet vertexů (maxV)	`6`
Počet polygonů (maxP)	`3`
Počet čtyřúhelníků (maxQ)	`1`
Vertex 0	`[x0, y0, z0]`
Vertex 1	`[x1, y1, z1]`
Vertex 2	`[x2, y2, z2]`
Vertex 3	`[x3, y3, z3]`
Vertex 4	`[x4, y4, z4]`
Vertex 5	`[x5, y5, z5]`
Trojúhelník 0	`[0, 1, 2]`
Trojúhelník 1	`[3, 5, 4]`
Čtyřúhelník 0	`[1, 3, 4, 2]`

Tabulka 4.2: Datový popis jednoduchého modelu

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	maxV		maxP		maxQ		x0				y0				z0	
20	z0		x1				y1				z1				x2	
30	x2		y2				z2				x3				y3	
40	y3		x4				y4				z4				x5	
50	x5		y5				z5				0		1		2	
60	3		5		4		1		3		4		2			

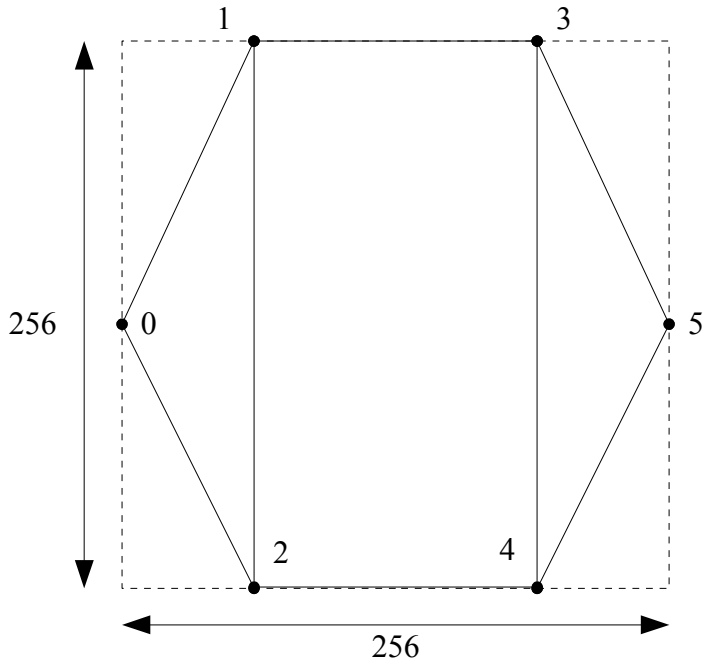
Tabulka 4.3: Výchozí formát modelových dat

Velikost kompletní datové struktury je 110 bytů, což se může jevit jako zanedbatelné. Problém s touto reprezentací nastane ve chvíli, kdy se pokusíme uložit model s velkým počtem polygonů. Příkladem budiž v projektu obsažený model stromu, který sestává ze zhruba 1600 vertexů a téměř stejného počtu čtyřúhelníků. Takový objekt má v tomto formátu velikost zhruba 40 kB, po použití komprese necelých 30 kB, což je naprosto nepřijatelné.

4.1.2 Změna formátu vertexových souřadnic

Z tabulky 4.3 vyplývá, že největší prostor v datech celého modelu zaujímají vertexové souřadnice. Prostor čtyř bytů na jednu souřadnicovou složku je zbytečně velký. Pro jednoduché účely dema postačí mnohem menší velikosti, konkrétně pouhý jeden byte (tzn. 3 byty na vertex).

Transformaci na jednobytové souřadnicové složky provedeme tak, že celý model vložíme a roztáhneme do krychle o hraně 256 jednotek. Následně všechny vertexové souřadnice zaokrouhlíme na celá čísla a převedeme na byty, viz obrázek 4.4.



Obrázek 4.4: Zarovnání modelu do „jednotkové“ krychle

Viditelný problém se změnou proporcí celého modelu se snadno vyřeší uložením limitů v jeho původním rozměru. Dalším méně viditelným problémem jsou chyby vzniklé při zaokrouhlování na celočíselné hodnoty. Naštěstí na většinu použitých modelů přesnost jednoho bytu bohatě stačí a změna oproti přesnějšímu originálu je prakticky nerozeznatelná.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	maxV		maxP		maxQ		minX				minY				minZ	
20	minZ		maxX				maxY				maxZ				x0	y0
30	z0	x1	y1	z1	x2	y2	z2	x3	y3	z3	x4	y4	z4	x5	y5	z5
40	0		1		2		3		5		4		1		3	
50	4		2													

Tabulka 4.5: Formát s vertexy omezenými na bytové hodnoty souřadnic

Nově navrhnutý způsob reprezentace dat zmenšil celkovou velikost modelu na 84 bytů. Oproti předchozí verzi byly přidány zmíněné limity původních rozměrů, aby bylo možno model zvětšit (zmenšit) zpět na korektní velikost. Dříve zmíněný model stromu se tímto způsobem zmenší na 18 kB, po kompresi na 12 kB, což je stále nedostatečné.

4.1.3 Delta modulace a uspořádání vertexových souřadnic

Delta modulací se rozumí jednoduchá úprava, kdy nejsou ukládány konkrétní hodnoty jednotlivých souřadnicových složek, ale jejich změny oproti předchozí hodnotě.

V souvislosti s delta modulací je vhodné sjednotit jednotlivé složky souřadnic do společných bloků. To znamená, že vertexové souřadnice jednotlivých os budou uspořádány vedle sebe. Více napoví tabulka 4.6.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	maxV		maxP		maxQ		minX				minY				minZ	
20	minZ		maxX				maxY				maxZ				x0	dx1
30	dx2	dx3	dx4	dx5	y0	dy1	dy2	dy3	dy4	dy5	z0	dz1	dz2	dz3	dz4	dz5
40	0		1		2		3		5		4		1		3	
50	4		2													

Tabulka 4.6: Vertexové souřadnice s delta modulací

Z tabulky 4.6 můžeme vyčíst, že počáteční hodnoty jednotlivých bloků delta modulaci nemají, což je způsobeno tím, že nemají své předchůdce, od kterých by byl vypočítán rozdíl (popř. je jejich předchůdce roven nule).

Může se zdát, že tento krok je naprosto zbytečný. Model má sice stále stejnou velikost, ale v případě komprese dosáhneme o něco lepšího poměru. U modelu stromu se jedná o úsporu 1 kB po komprimaci, což je vzhledem k jednoduchosti tohoto vylepšení zajímavý výsledek.

4.1.4 Delta modulace vertexových indexů polygonů

Posledním krokem při úpravě datového formátu je delta modulace zbývající části, tedy jednotlivých vertexových indexů u výčtu polygonů.

Řešení vypadá stejně jednoduše, jako předchozí krok. Ovšem při detailnějším pohledu zjistíme, že po delta modulaci indexů se v datech vytvoří dlouhé úseky, kde za nulovým bytem následuje byte s malou hodnotou (malou změnou) a úseky stejného charakteru, ale v doplňkovém kódu. Názornější informace poskytne tabulka 4.7 (jednotlivé dvojice bytů jsou sjednoceny výraznějším ohraničením).

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	maxV		maxP		maxQ		minX				minY				minZ	
20	minZ		maxX				maxY				maxZ				x0	dx1
30	dx2	dx3	dx4	dx5	y0	dy1	dy2	dy3	dy4	dy5	z0	dz1	dz2	dz3	dz4	dz5
40	00	00	00	01	00	01	00	01	00	02	FF	FF	FF	FD	00	02
50	00	01	FF	FE												

Tabulka 4.7: Vertexové indexy polygonů s delta modulací

Nevhodné uspořádání dat v tabulce 4.7 můžeme změnit třemi jednoduchými kroky. Prvním krokem je změna způsobu zobrazení záporných čísel z doplňkového kódu na kód přímý. V tomto případě budou záporná čísla označena logickou jedničkou na nejvýznamnějším bitu (15.), zbývající část zůstane nezměněna (nedojde k „podtečení“ jako u doplňkového kódu).

Použití přímého kódu bez dodatečných úprav ovšem situaci nijak nezlepší. Tím bychom jen zařidili, že na nejvyšším bytu jednotlivých hodnot se nám střídá nulová hodnota s hodnotou 128. Nabízí se tedy jednoduché přeuspořádání bitů.

Protože jsme tímto způsobem vytvořili souvislý blok dat, který je prakticky celý prokládaný nulovými hodnotami (znaménkový bit je přesunut na nejvyšší pozici nižšího bytu), nabízí se rozdělení všech dvoubajtových složek do dvou samostatných bloků. První blok bude obsahovat část

dat, která je při delta modulaci „nejaktivnější“, tzn. prvních 7 bitů se znaménkovým bitem navíc. Tato oblast tedy pojme veškeré indexové změny v rozsahu -127 až +127. V dalším bloku následují data významnějších bitů.

Toto řešení radikálně snižuje výslednou velikost komprimovaného souboru. Z obrázku 4.1 je patrné, že se vertexové indexy u většiny polygonů liší jen o malé hodnoty. U velké části trojrozměrných modelů na většině místech nedojde k překročení rozsahu -127 až +127, takže ve výsledku získáme dvě lehce komprimovatelné pole. První, které obsahuje převážně neustále se opakující sekvence čísel a druhé, které obsahuje převážně dlouhé nulové úseky. Zmiňovaný model stromu lze po této optimalizaci zkomprimovat na 7 kB, což se dá oproti původní komprimované velikosti 30 kB považovat za velký úspěch.

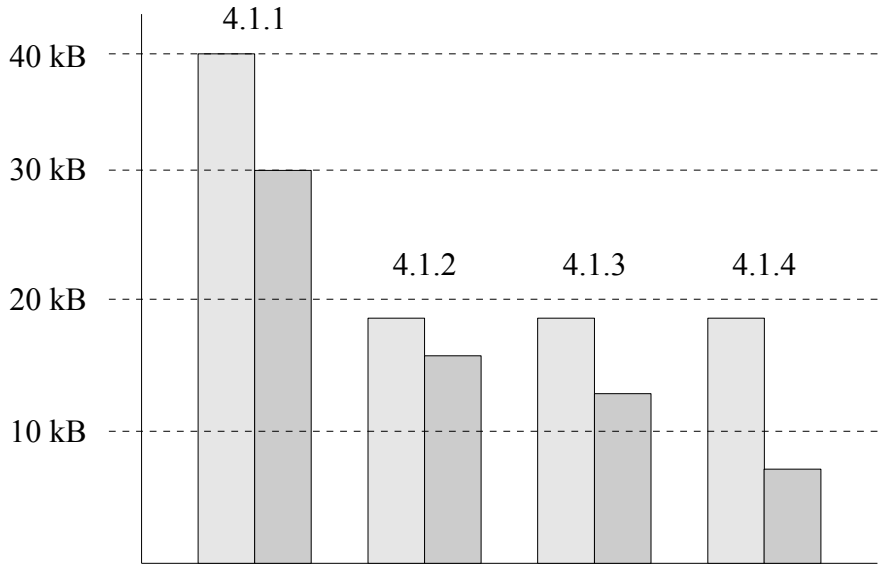
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**

Obrázek 4.8: Hodnota (-5) v klasickém doplňkovém kódu (nahoře) a v upraveném přímém kódu

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	maxV		maxP		maxQ		minX				minY				minZ	
20	minZ		maxX				maxY				maxZ				x0	dx1
30	dx2	dx3	dx4	dx5	y0	dy1	dy2	dy3	dy4	dy5	z0	dz1	dz2	dz3	dz4	dz5
40	00	01	01	01	02	71	73	02	01	72	00	00	00	00	00	00
50	00	00	00	00												

Tabulka 4.9: Náhled na finální reprezentaci modelových dat



Obrázek 4.10: Velikost modelu stromu v nekomprimovaně (světle) a komprimovaně (tmavě)

4.2 Algoritmus Catmull-Clark

V kapitole 4.1 bylo zmíněno, že z důvodu úspory velikosti výsledného programu je nutno drtivou většinu trojrozměrných modelů reprezentovat pokud možno s co nejnižším počtem polygonů. Problémem je v tomto případě výskyt jakýchkoliv zaoblených ploch, které jsou v konečné podobě tvořeny vysokým počtem polygonů.

Výborným řešením je použití algoritmu Catmull-Clark, který model rozdělí na čtyřúhelníkovou síť, která je oproti originálu „hladší“. Následující popis nezahrnuje vysvětlení kompletního matematického modelu, pro podrobnější informace viz [4].

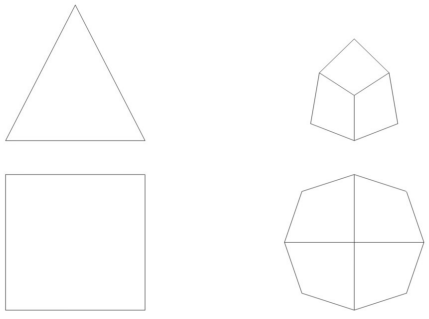
4.2.1 Postup dělení

Následující popis čerpá z [4]. Začínáme s polygonovým modelem, u kterého budeme všechny jeho vertexy nazývat **originálními vertexy**.

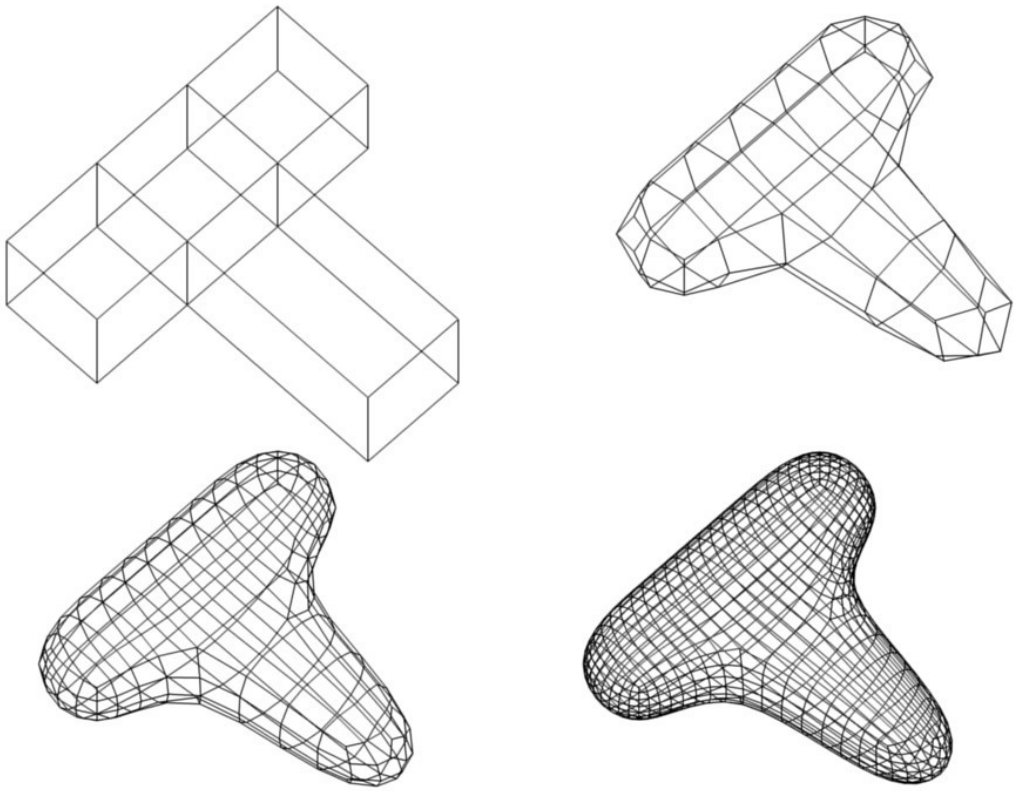
- Pro každý polygon vytvoříme **středový vertex**
  - Nastavíme pozici **středového vertexu** na aritmetický průměr všech přilehlých originálních vertexů daného polygonu
  - U každého **středového vertexu** vytvoříme hrany, které spojí středový vertex se všemi **hranovými vertexy** daného polygonu
- Pro každou hranu vytvoříme **hranový vertex**
  - Nastavíme pozici **hranového vertexu** na aritmetický průměr všech přilehlých **středových vertexů** a **originálních vertexů**
- Pro každý **originální vertex** P zjistíme aritmetický průměr F všech přilehlých **středových vertexů**, jejichž polygony se dotýkají P. Dále vypočítáme aritmetický průměr R všech středů hran, které se dotýkají P, kde zaždý střed hrany je vypočítán jako aritmetický průměr dvou spojujících vertexů. Každý originální vertex pak posuneme na pozici vypočítanou podle následujícího vzorce, kde n je počet polygonů přilehlých k P:

$$\frac{F + 2\mathrm{R} + (n-3)\,P}{n}$$

Pro lepší názornost je na obrázku 4.11 zobrazen výsledek dělení pro jednotlivé základní polygonové typy (jiné v demu nejsou použity). Na obrázku 4.12 jsou pak zobrazeny tři kroky dělení pro jednoduchý model.



Obrázek 4.11: Jeden krok algoritmu Catmull-Clark (vpravo) pro základní typy polygonů



Obrázek 4.12: Postupné aplikování algoritmu Catmull-Clark na jednoduchý model

Z předchozích dvou obrázků a ze samotného algoritmu vyplývá, že produktem dělení je model obsahující pouze čtyřúhelníky. Nevýhodou je, že v případě modelu, který obsahuje velké množství trojúhelníků, mohou vznikat různé vizuální artefakty.

Další nevýhodou je postupná rostoucí paměťová náročnost. Každý krok dělení totiž zvyšuje celkový počet polygonů až čtyřnásobně. Z toho důvodu má uživatel při startu programu možnost volby nízkých detailů, které zamezí použití Catmull-Clark algoritmu na většinu modelů.

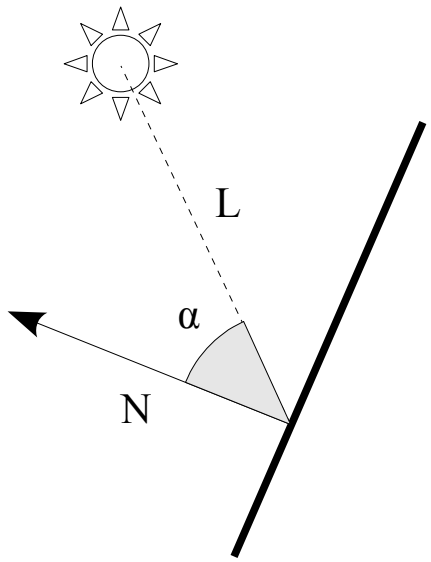
4.3 Optimalizace vykreslování

4.3.1 Předpočítané osvětlení

Aby scéna vypadala dostatečně realisticky, je vždy potřeba její správné nasvícení. Náročnost tohoto procesu je bohužel přímo úměrná celkovému počtu osvícených polygonů. V našem případě se v demu nevyskytují žádné dynamické světla, které by v čase měnily svou polohu, barvu nebo intenzitu, proto se nabízí možnost veškeré osvětlení při startu programu předpočítat.

Nárůst výkonu je v případě této optimalizace velmi vysoký. Hlavní výhoda spočívá ve faktu, že není potřeba v každém snímku u každého modelu přeposílat kompletní soubor normálových vektorů, díky kterému je následně počítána intenzita osvětlení u jednotlivých vertexů. Namísto toho je veškeré osvětlení spočítáno předem a s každým vertexem je pak zároveň předána i jeho vlastní barva.

V celé scéně je definován jediný zdroj světla, podle kterého je každý model osvětlen ihned po svém načtení. Matematický model je založen na jednoduchém určení intenzity světla dopadajícího na vertex nebo polygon (podle zvoleného typu stínování) dle skalárního součinu vlastního normálového vektoru a vektoru směřujícího ke zdroji světla.



Obrázek 4.13: Grafické znázornění výpočtu intenzity dopadajícího světla

Podle obrázku 4.13 je zřejmé, že čím bude znázorněný úhel mezi normálou plochy a vektorem směřujícím ke zdroji světla menší (tzn. plocha bude více nakloněna světlu), tím bude plocha osvětlena intenzivněji. Totéž platí analogicky pro samostatné vertexy. Výpočet tohoto vztahu je založen na jednoduché rovnici:

$$arc\cos a = \frac{\vec{N} \times \vec{L}}{|\vec{N}| \cdot |\vec{L}|}$$

V případě, že všechny vektory před použitím normalizujeme (přepočítáme na jednotkovou délku), můžeme arcus kosinus úhlu spočítat prostým skalárním součinem a z výsledku přímo odvodit intenzitu.

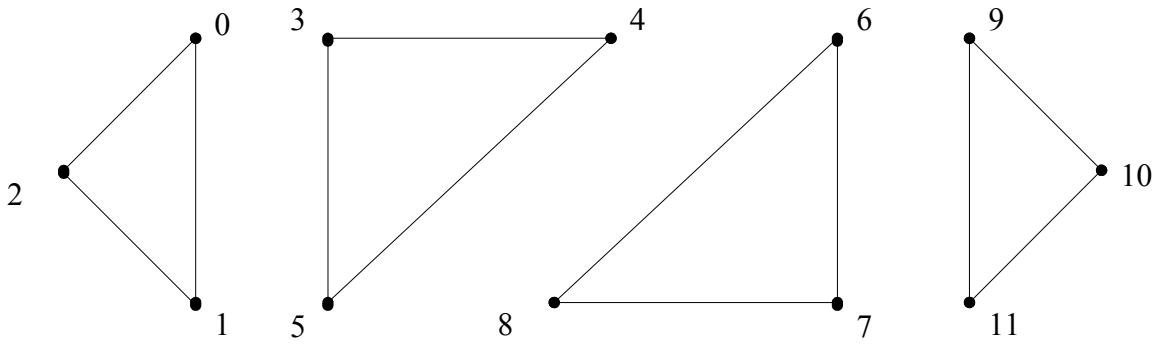
Tato optimalizace má ovšem jednu nevýhodu. Výpočet osvětlení totiž většinou není ta poslední úprava, která je na model aplikovaná. Problém nastává tehdy, když model začneme jakkoliv transformovat (posunovat, rotovat, měnit velikost). Jelikož je osvětlení staticky předpočítané, transformacemi se nijak nezmění, ačkoliv by mělo. Ve většině případech je však tato chyba přijatelná a běžný pozorovatel ji prakticky nezaznamená.

4.3.2 Vertexové pole (vertex array)

Obrázek 4.1 naznačil, jakým způsobem jsou v programu uchovány modelová data. Všechny polygony jsou popsány pouze vertexovými indexy, skrze které lze získat podrobnější informace o daném vrcholu (např. úmístění, normálový vektor, barva). Bohužel je tento způsob uchování dat z hlediska rychlosti přístupu nevýhodný. Při vykreslování polygonů se totiž svým způsobem neustále chaoticky pohybujeme v poli uložených vertexů a často se i několikrát vracíme k dříve již získaným

informacím. Tento způsob je sice relativně nenáročný na velikost obsazené paměti, ale naopak velmi neefektivní v rámci efektivního pohybu v paměťovém prostoru a velmi neefektivní v rámci samotného rozhraní OpenGL.

Mnohem výhodnějším, avšak paměťově náročnějším řešením je každý model „rozbalit“ do proudu vertexů z jednotlivých trojúhelníků a následně všechny vertexy v proudu hromadně zpracovávat prostředky OpenGL (obrázek 4.14).



Obrázek 4.14: Výsledek rozložení původního modelu 4.1

Celkový počet vertexů oproti původnímu modelu stoupnul dvojnásobně, což skutečně dokazuje velmi zvýšenou paměťovou náročnost oproti předchozímu způsobu uspořádání. Ovšem díky této datové reprezentaci nyní můžeme velmi jednoduše vykreslit celý model bez nutnosti neustálého volání funkcí **`glVertex3f()`**, **`glBegin()`**, **`glEnd()`** apod.

Následující kód znázorňuje, jakým způsobem jsou modely v demu vykreslovány:

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glInterleavedArrays(GL_C3F_V3F, 0, vertexPointer);

glDrawArrays(GL_TRIANGLES, 0, vertexCount);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Ve funkci **`glInterleavedArrays()`** je použit parametr `GL_C3F_V3F`, kterým oznamujeme způsob uložení dat v paměti. Jak již bylo řečeno v kapitole 4.3.1, veškeré osvětlení je předpočítáno a s každým vertexem je zároveň v proudu uložena i jeho barva (což z obrázku 4.14 nevyplývá). Funkci **`glDrawArrays()`** nakonec předáme způsob uspořádání (v našem případě trojúhelníky) a celkový počet vertexů.

5 Scéna

Modul scény zajišťuje základní rozhraní pro renderování výsledného obrazu. Zajišťuje načtení všech modelů včetně jejich dělení algoritmem Catmull-Clark a zároveň inicializuje kompletní časování sekvencí kamer, textu a některých objektů (duch, lekníny).

5.1 Vodní hladina

Efekt odrazu scény na vodní hladině je implementačně velice jednoduchý a razantně zvyšuje celkový vizuální dojem. Bohužel za cenu dvojnásobně zvýšené zátěže, protože prakticky celá scéna se touto metodou musí kreslit dvakrát. Aby byl výsledek korektní, musíme provést následující kroky:

- Nastavíme pozici a rotaci kamery inverzně vůči směrnici odrazové plochy
- Zapneme ořezávací test
- Vykreslíme scénu
- Vypneme ořezávací test, vymažeme hloubkový buffer
- Vrátíme kameru do původní polohy
- Vykreslíme průhlednou vodní hladinu
- Vykreslíme scénu

První krok se může jevit jako výpočetně komplikovaný, ale není tomu tak. Jelikož je ve scéně vodní hladina zobrazena jako nekonečná plocha bez náklonu (rovnoběžná se dvěmi osami souřadného systému), postačí nám k inverzi kamery následující kód:

```
sceneSetCamera();
glScalef(1.0f, 1.0f, -1.0f);
glTranslatef(0.0f, 0.0f, -waterLevel);
```

První příkaz nastaví modelovou matici, aby odpovídala pohledu kamery nad hladinou. Příkazem **`glscalef()`** nastavíme modelovou matici tak, aby všechny vertexy byly překlopeny podél osy Z. Dodatečně ještě celou scénu musíme posunout opačně vůči stanovené úrovni vodní hladiny (proměnná `waterLevel`). Takto získáme kameru, která sleduje celou scénu z pohledu pod hladinou.

Dalším důležitým krokem je povolení ořezávacích testů, které zajistí, aby objekty „pod hladinou“ nezasáhly do scény nad hladinou. K ořezání potřebujeme znát pouze parametry rovnice ořezávací plochy, vše ostatní již zajistí OpenGL. K parametrickému vyjádření musíme znát posun v souřadném systému a normálový vektor roviny, který je v našem případě kolmý k rovině podél os X a Y. Viz následující kód:

```
double eqr[4];
eqr[0] = 0.0f; eqr[1] = 0.0f; eqr[2] = 1.0f;
eqr[3] = -waterLevel;

glEnable(GL_CLIP_PLANE0);
glClipPlane(GL_CLIP_PLANE0, eqr);
```

První tři prvky pole `eqr` obsahují složky normálového vektoru a čtvrtý prvek posun, který je v tomto případě roven výšce vodní hladiny. Po zapnutí ořezávací roviny funkcí **`glEnable()`** a nastavením parametrů funkcí **`glClipPlane()`** budou veškeré části polygonů nad vodní hladinou ořezány.

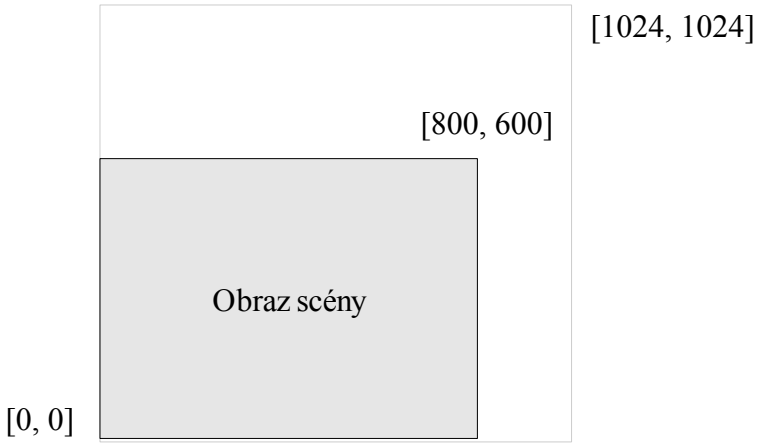
Teď nám již nic nebrání k vykreslení scény z pohledu pod vodní hladinou. Ihned poté vypneme ořezávací rovinu příkazem **`glDisable()`** a vymažeme hloubkový buffer, aby odražený obraz neovlivňoval nově renderované polygony. Nastavíme kameru do původní polohy a vykreslíme zbývající části scény.

5.2 Radiální rozmazání

Efekt radiálního rozmazání (tzv. radial blur) je zde úmyslně použit proto, aby více zdůraznil celkový náladový charakter dema. Radiálním rozmazáním se rozumí rozmazání pixelů obrazu oproti směru, než se nachází střed rozmazání.

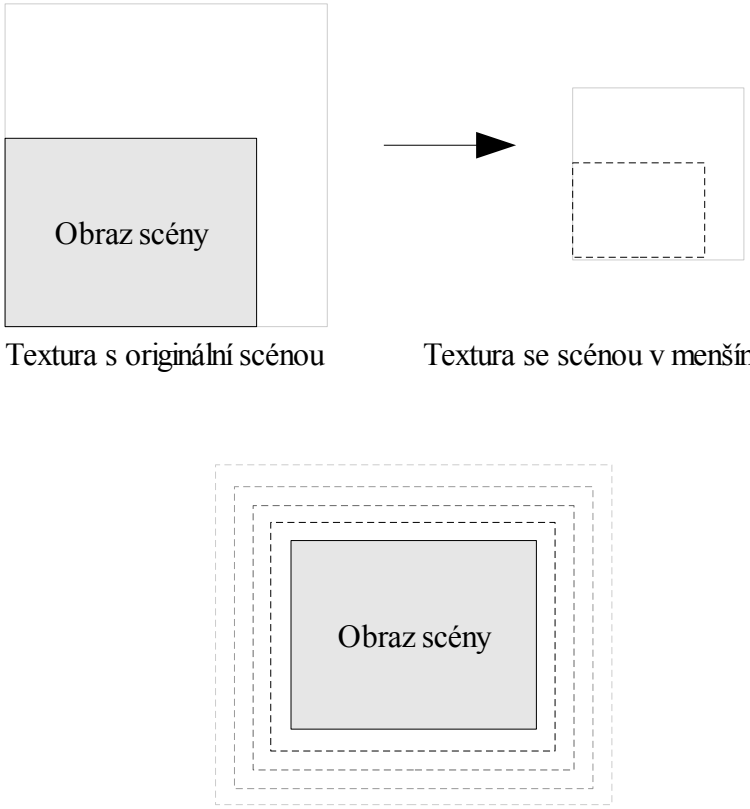
V případě softwarového renderingu postačí aplikace několika jednoduchých konvolučních filtrů a následný alpha-blending zvětšujících se výsledných obrazů přes sebe. U hardwarové realizace na běžných grafických kartách si tento „luxus“ v podobě přímého přístupu k obrazovým pixelům nemůžeme dovolit. Musíme se proto uchýlit k alternativnímu řešení za pomoci renderování do textury.

Základním a prvním krokem celého efektu je vykreslení scény v původním rozlišení do textury. Jelikož hardware grafických karet z optimalizačních důvodů nepřijímá textury s rozměry jinými, než jsou násobky mocniny čísla dva, musíme vytvořit dostatečně velkou texturu, abychom scénu vyrenderovali bez jakéhokoliv ořezání či změny rozměrů. Pro rozlišení 800x600 pixelů bude mít tato textura velikost 1024x1024 pixelů, což je s ohledem na předchozí podmínky nejmenší možná velikost (obrázek 5.1). O nevyplněné místa na okrajích textury se nemusíme starat. Na obrazovce bude zobrazen pouze výřez plochy z původního rozlišení.



Obrázek 5.1: Využití pomocné textury s obrazem scény

V dalším kroku potřebujeme vytvořit texturu, kterou několikrát naneseme na výsledný obraz scény. K tomu úmyslně použijeme texturu menších rozměrů a degradujeme kvalitu uloženého obrazu. Tímto způsobem můžeme částečně nahradit efekt rozmazání podobně, jako bychom použili konvoluční filtr. Postačí celý obraz dvakrát nebo čtyřikrát zmenšit (vícenásobné zvětšení již nevypadá nejlépe) a po jeho roztažení nad původním obrazem se hardwarové bilineární filtrování postará o rozmazání (obrázek 5.2).



Obrázek 5.2: Zmenšení pomocné textury se scénou a její několikanásobné nanesení přes obraz

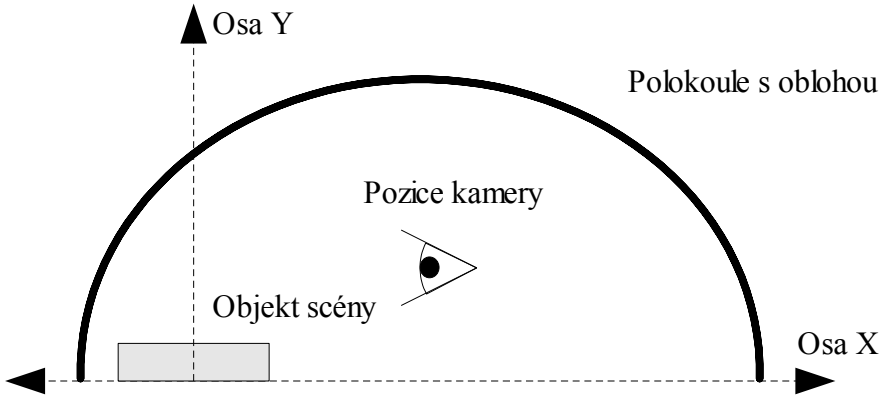
Obrázek 5.3: Výsledek aplikace efektu na jednoduchý obrázek s textem

6 Obloha

6.1 Polokoule

Předtím než začneme jakékoliv vykreslování oblohy a mraků, musíme se rozhodnout, na jaký útvar nebe namapujeme. V podstatě máme na výběr ze dvou možností. První možností je mapování na krychli (tzv. skybox), druhou možností je mapování na polokouli (skydome).

Ačkoliv krychle je z hlediska vykreslování jednodušší (6 stěn), mapování textury je oproti polokouli komplikovanější. U krychle je totiž potřeba deformovat mapované textury tak, aby v dálce uživatel vlivem perspektivy nepoznal ostré hrany krychle. U procedurálně generované textury mraků by tento proces znamenal zbytečné výpočty navíc. Proto je jako mapovací objekt zvolena polokoule.



Obrázek 6.1: Polokoule s oblohou ve scéně

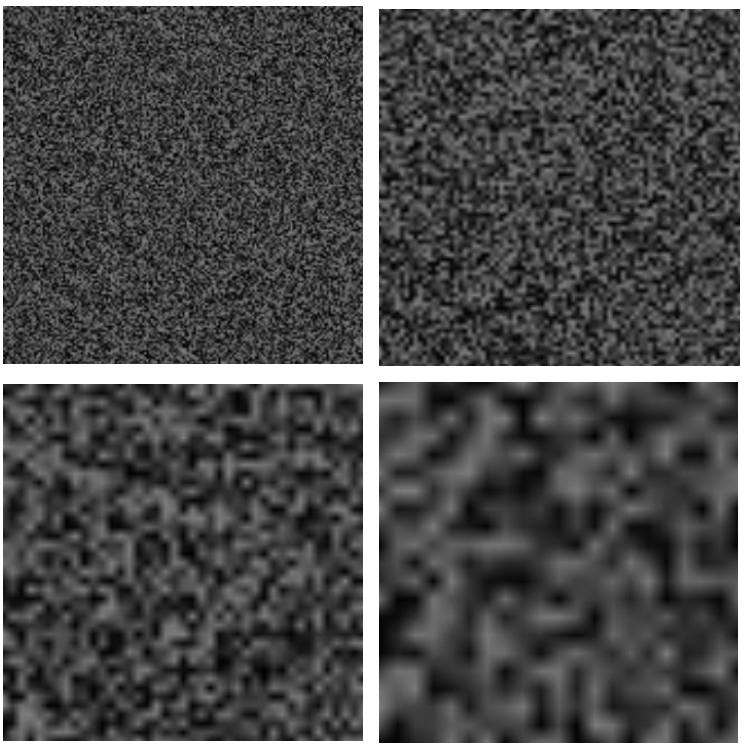
Na obrázku 6.1 si můžeme všimnou důležité skutečnosti. Poloha oblohy je vždy určena relativně vůči pozici kamery (platí i pro krychli). Jedná se o prostou snahu zabránit kameře dosažení horizontu. Při pohybu kamery se tedy bude nebe jevit nehybně, zatímco se ostatní objekty budou pohybovat. Stejně jako v reálném světě.

6.2 Mraky a Perlinův šum

Textura mraků je jedinou texturou, která se v demu vyskytuje. I přesto, že je jediná, musí být při startu programu generovaná. Kvůli svým rozměrům (512x512 pixelů) je její statické uchování v programu nemožné. Otázkou zůstává, jakým způsobem tuto texturu „vyrobit“?

Odpověď nabízí využití jednoduchého algoritmu Perlinova šumu (autorem je Ken Perlin). Myšlenka algoritmu vychází ze sady vygenerovaných náhodných čísel, sloužící jako základ šumových funkcí o různých frekvencích (oktávách), které jsou následně kombinovány dohromady.

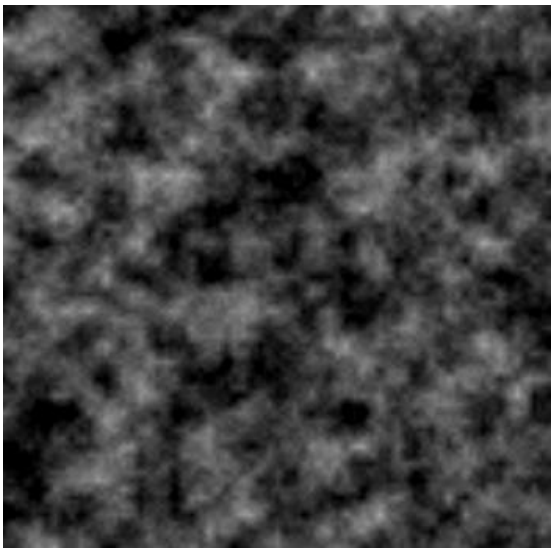
Protože textura mraků je dvourozměrná, budeme potřebovat dvourozměrnou šumovou funkci. Na obrázku 6.2 jsou znázorněny 4 oktávy z náhodně vygenerované sady čísel. Aby se předešlo tzv. pixelizaci je prostor mezi jednotlivými body interpolován lineární funkcí, která vytvoří hladké přechody mezi jinak barevnými skoky. Pro lepší kvalitu výsledného obrazu se mohou použít i jiné druhy interpolací (kosinová, spline křivka), ale náš případ si vystačí s interpolací lineární.



Obrázek 6.2: 4 oktávy dvourozměrné šumové funkce

Jakmile dokážeme vypočítat jednotlivé oktávy, zbývá nám provést jejich sloučení. Abychom získali očekávaný výsledek, musíme při jejich sčítání zároveň upravovat jejich amplitudy. Čím vyšší má oktáva frekvenci (jemnější zrnitost), tím menší musí mít amplitudu. Na obrázku 6.3 je znázorněn výsledek při zachování těchto pravidel.

Můžeme si všimnout, že tvarový základ obrazu leží v oktávách s nižšími frekvencemi (4. část obrázku 6.2). Oktávy s vysokými frekvencemi (1. část obrázku 6.2) ovlivňují výsledek minimálně. Výsledný obraz poslouží jako ideální základ pro texturu mraků.



Obrázek 6.3: Dvourozměrný Perlinův šum se čtyřmi oktávami

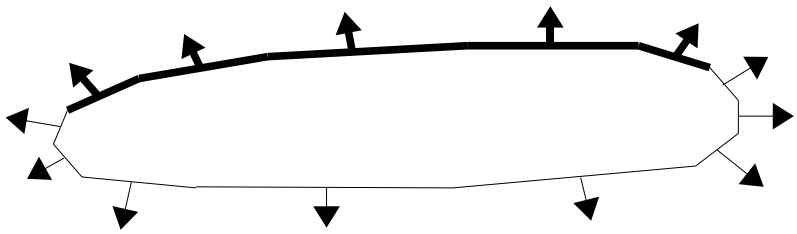
7 Tráva

Tráva je v tomto demu příkladem zvláštní formy statického částicového systému. Každý trs je unikátní částice, která nemění svou polohu, ale která se pouze ohýbá ve směru fiktivně vanoucího větru.

7.1 Obklopení objektu částicemi

Aby celý systém pro vizualizaci trávy vypadal dostatečně realisticky, je potřeba použít co možná největší počet částic. V případě tohoto dema se jedná o počet téměř deseti tisíc částic.

Při tak vysokých číslech vzniká zásadní problém, jakým způsobem všechny částice správně rozmístit na zvoleném objektu. Způsobů je několik. První možností je vlastní definice jednotlivých pozic ve zdrojovém kódu, což je při použitém objemu spíše nesmysl, než alternativa. Další možností je ohlehčení první možnosti a vytvoření menšího počtu několika travnatých „ostrovů“. Tato alternativa je jednodušší, ale v případě použití zakřiveného povrchu (náš případ) stále zbytečně komplikovaná. Poslední možností je automatické rozmístění částic podle tvaru vybraného modelu. U zakřivených povrchů prakticky jediný možný způsob.



Obrázek 7.1: Profil modelu s jeho normálovými vektory

Než začneme pokrývat model časticemi trávy, musíme vědět, které jeho části jsou k pokrytí vhodné. Pokrýt objekt ve všech směrech by bylo nejen neefektivní, ale i nelogické a odporující realitě. Proto vybereme pouze ty části (polygony), jejichž normálový vektor má dostatečně vysoký sklon (zvýrazněné segmenty obrázku 7.1).

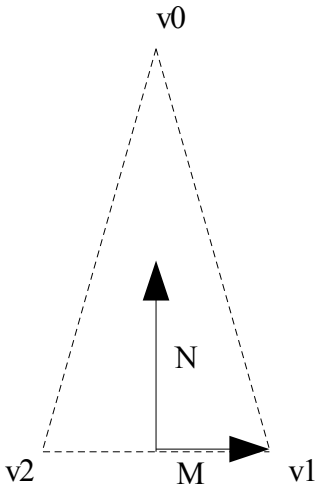
Jakmile víme, které polygony modelu mohou posloužit jako základna pro částice trávy, zbývá nám zjistit, jakým způsobem je na dané polygony náhodně rozmístit. Pro tento účel jsou velmi užitečné tzv. barycentrické koordináty [6]. Dle nich pro jakýkoliv bod P v trojúhelníku ABC platí následující vztah:

$$P = t \cdot A + u \cdot B + v \cdot C$$
$$1 = t + u + v$$

Ve vzorci pak proměnné A, B a C zastupují body trojúhelníku a proměnné t, u, v barycentrické souřadnice. V případě, že chceme najít libovolný náhodný bod na takovém trojúhelníku, postačí nám náhodně zvolit např. proměnné t a u v intervalu <0, 1> tak, aby jejich součet nebyl větší než 1. Zbývající barycentrickou souřadnici v dopočítáme podle výše uvedeného vzorce. Dosazením získáme náhodný bod P, který leží na trojúhelníku ABC.

7.2 Vykreslení a pohyb částic

V kapitole 7.1 již bylo zmíněno, že počet částic trávy dosahuje hodnoty téměř deseti tisíc. S tak velkým množstvím polygonů je nutno zacházet obdobně, jako v případě modelových dat s jejich optimalizací vertexovými poli.



Obrázek 7.2: Polygon částice trávy

Každá částice trávy je reprezentována jediným trojúhelníkovým polygonem (obrázek 7.2). Orientaci částice určují dva směrové vektory. Vektor *N* na obrázku je shodný s normálovým vektorem polygonu, na kterém částice leží (kvůli zachování zakřivení povrchu). Vektor *M* je doplňující a slouží k nastavení pootočení celého polygonu ve směru osy vektoru *N*.

Simulace pohybu ve větru je realizována modifikací polohy vertexu v0. Vrchol polygonu je posunován do stran ve směru vektoru *M* jednoduchou sinusovou funkcí. Úhel pohybu je navíc u každé částice individuálně fázově posunutý. Takto se vytvoří dojem, že vítr nefouká jednotně na všechny částice najednou, ale působí ve vlnách, které plynule putují po celém povrchu.

7.3 Dodatečné estetické úpravy

I přes ryze automatický proces obklopení modelu částicemi trávy je v některých případech nutno zasáhnout a vytvořený virtuální porost individuálně poupravit.

Na některých místech může být výška trávy zbytečně velká (ve scéně dema např. u klavírní stoličky nebo lavičky), proto rozhraní poskytuje možnost dodatečně poupravit „škody“ napáchané automatickým generátorem.

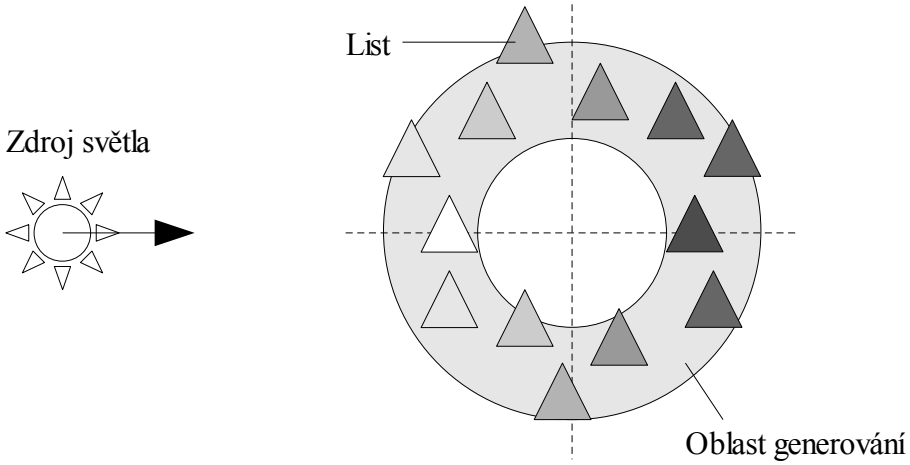
Dalším méně viditelným detailem je změna odstínu některých míst k navození dojmu stínu. Jedná se pouze o změnu barvy částic v určeném okruhu. Tento efekt byl využit v místech okolo kmene objektu stromu, pod objektem klavíru a za objektem náhrobního kamene. Díky tomu tyto místa působí dojmem, že skutečně vrhají stín, ačkoliv v jejich okolí jsou pouze jinak zbarvené částice trávy.

8 Listí

Částicový systém listí je obdobou předchozího částicového systému trávy. Liší se v několika detailech, principem i způsobem vykreslování je však téměř totožný.

8.1 Generování shluků

Pokud bylo v případě trávy nevýhodné „ručně“ určovat souřadnice růstu a namísto toho bylo výhodné nechat veškeré rozložení částic vygenerovat, zde musíme stanovisko změnit. Model stromu, na kterém má lístí vyrůst, je příliš komplikovaný a univerzální algoritmus pro automatické rozmístění částic listů by se programoval obtížně. Výhodnější proto bude manuálně rozmístit větší shluky listí.



Obrázek 8.1: Jednoduchý nákres listů ve shluku a jejich osvětlení

V programu je definováno 40 shluků v celé koruně stromu. Každý shluk vygeneruje při plných detailech 250 náhodných částic listí v prostoru duté koule určitého poloměru (viz obrázek 8.1). Prostor celé koule není zaplněn z důvodu, aby se částice listí spíše nakoncentrovaly v okrajích a vytvořily tak dojem plného středu. V opačném případě by muselo být ke stejnému efektu zapotřebí mnohem více částic.

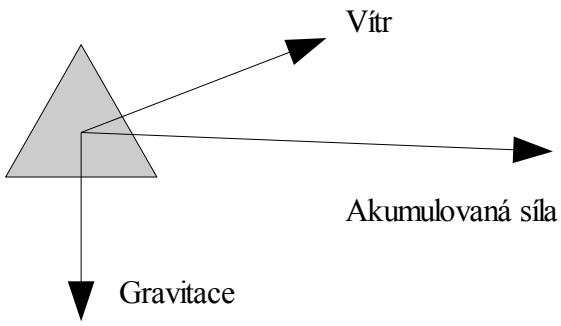
Stejně jako v kapitole 4.3.1 je i zde využito techniky předpočítaného osvětlení. Shluk je v tomto případě skutečně považován za kouli, kde jsou částice listí na světlu odvrácené straně tmavší.

8.2 Vykreslení a pohyb listí

V předchozí kapitole 8.1 bylo uvedeno, že v programu je definováno 40 shluků, každý o 250 listech. Celkový počet opět dosahuje deseti tisíců, proto se ani částicový systém listí nevyhne optimalizaci shodné s optimalizací systému trávy.

Tvar částice listu je opět trojúhelníkového charakteru. Rotace částice je taktéž tvořena dvěma směrovými vektory, ale s tím rozdílem, že oba jsou generovány náhodně (listy nemají referenční

povrch pro normálu). Hlavní změnou oproti částicím trávy je propracovanější systém pohybu. Každý list se může pohybovat podle jednoduchého fyzikálního modelu.



Obrázek 8.2: Síly působící na částici listu

V případě, že nastane událost, kdy mají částice listů opustit korunu stromu, je jejich pohyb ovlivňován třemi silovými vektory (obrázek 8.2). Vektor gravitace je konstantní pro všechny částice. Vektor větru je taktéž konstantní s tím rozdílem, že jsou u každé částice přidány drobné odchylky, aby pohyb vypadal více chaoticky a tudíž i přírozeněji. Akumulovaná síla slouží k simulaci setrvačnosti a akcelerace. Pohybové rovnice můžeme vyjádřit takto:

$$\vec{A} = \vec{A} + (\vec{V} + \vec{G}) \cdot dt$$
$$\vec{P} = \vec{P} + \vec{A}$$

Při výpočtu každého snímku je akumulovaná síla A navýšena o součet vektorů větru a gravitace v míře přímo úměrné prodlevě oproti předchozímu snímku (dt). Akumulovaná síla je následně přičtena k pozici částice P.

Samozřejmě, že tento fyzikální model není nikterak přesný (alespon v rámci časové integrace). Ale naopak je dostatečně jednoduchý, aby s jeho použitím bylo možno přepočítat velké množství částic při každém renderovaném snímku.

9 Doplňující moduly

Následující kapitola popisuje zbývající programové jednotky, jejichž rozsah a struktura je, oproti modulům popsaných v kapitolách předchozích, podstatně jednodušší.

9.1 Kamera

Implementace pohybu kamery je velmi jednoduše založena na kombinaci Bézierových křivek a funkce **`gluLookAt()`**. Systém je koncipován tak, aby bylo možno nejen řídit křivkou pohyb kamery, ale zároveň řídit křivkou směr pohledu.

Modul uchovává seznam kamerových střihů, které jsou spouštěné v jednotlivých časových úsecích a automaticky nastavuje aktuální kameru a interpoluje její pozici a rotaci v čase.

9.2 Prostorový text

Prostorový text využívá předrenderovaných fontů získaných ze specifického rozšíření OpenGL ve Windows (funkce **`wglUseFontBitmaps()`**, více na [1]). Podobně jako systém kamery uchovává seznam textů s jejich časy objevení, zmizení, pozice a pootočení a automaticky je zobrazuje v závislosti na aktuálním času.

9.3 Manažer objektů scény

Tento modul byl vytvořen převážně kvůli jedinému účelu – umožnit hromadné zpracování a vykreslování modelů a u speciálních případů vytvořit jednoduché efekty (kvetení leknínu, průhledný stín ducha). Ve své podstatě se jedná pouze o základní implementaci kontejneru trojrozměrných modelů ve scéně.

9.4 Klávesy klavíru

Modul, který pomocí základních funkcí rozhraní OpenGL vykresluje klaviaturu. Klávesy tedy nejsou tvořeny uloženým modelem, ale jsou generovány procedurálně. Zároveň je implementováno jednoduché rozhraní pro simulaci stisku vybraných kláves řízené globálním časovačem pro vytvoření dojmu synchronizace hry s hudbou.

9.5 Hudba

Velmi zajímavým problémem při tvorbě dema s omezenou velikostí je použití hudebního doprovodu. V případě, že chceme zvýšit výsledný dojem z celé prezentace, nevyhneme se jeho použití.

Je zřejmé, že běžné zvukové formáty typu WAVE, MP3 nebo OGG jsou v tomto případě naprosto nepoužitelné (i při drastickém snížení kvality). Alternativ existuje několik, každá má své specifické výhody a nevýhody.

Bezpochyby první a nejstarší zvukové řešení pro dema s omezenou velikostí byla (a v některých případech stále je) syntéza hudby pomocí MIDI (Musical Instrument Digital Interface). Hudba v MIDI formátu se obejde bez složitého popisu vzorků (samplů) jednotlivých použitých nástrojů. K produkci stačí pouhé časové značky s informacemi o výšce, dynamice a případném rozšiřujícím efektu k dané notě. Výhodou a zároveň nevýhodou takového řešení je přenechání veškerých výpočtů zvukovému hardwaru. Právě ten rozhoduje o výsledné kvalitě, kterou svým způsobem může ovlivnit pouze výrobce. Dnes se s tímto druhem reprodukce hudby v demech prakticky nesetkáme. Vyjímku tvoří ojedinělé případy z kategorií omezených na 4 kB a méně.

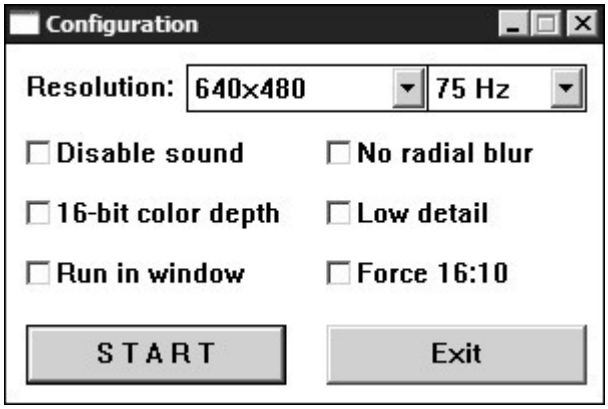
Druhou a dnes stále hojně využívanou metodou je použití tzv. hudebních modulů. Hudební moduly se svým konceptem snaží odstranit hlavní nevýhodu MIDI syntézy, tj. minimální možnost ovlivnit znění jednotlivých nástrojů. Stále využívají (mírně upravené) techniky časových notových značek, ale navíc obsahují kompletní soubor zvukových samplů jednotlivých nástrojů. Výhodou je v mnoha případech lepší kvalita výsledného zvuku, ovšem za ceny nárůstu celkového datového objemu (běžně několikanásobek oproti MIDI). Pro přehrávání modulů je vhodná knihovna FMOD, pro programy s omezenou velikostí pak speciálně její varianta miniFMOD. Více na [3].

Poslední variantou, která kombinuje předchozí dvě, je tzv. syntetizátor. Hudební syntetizátor je program, který je schopen pomocí různých matematických algoritmů tvořit zvukové vlny rozličných nástrojů a efektů. Toto demo používá jeden z nejznámějších volně dostupných [7] syntetizátorů jménem V2, který vytvořil Tammo Hinrichs, člen známé německé demoskupiny Farbrausch. V2 je primárně určen k použití v programech s omezenou velikostí. Jeho samostatná velikost je zhruba 10 kB, navíc poskytuje výbornou kvalitu výsledného zvuku. Použitá píseň je z volně dostupného programu Synthezze [11].

9.6 Konfigurační dialog

Většina dem dává uživateli před samotným startem možnost ovlivnit některá programová nastavení. Ani tento projekt není vyjímkou. Konfigurační okno ocení především majitelé hardwarově méně výkonných systémů.

Primárním účelem konfiguračního okna je poskytnout příležitost ovlivnit detaily výsledné animace (v převážně degradujícím směru).



Obrázek 9.1: Vzhled konfiguračního okna

V první položce „Resolution“ je možno vybrat celou škálu grafických rozlišení. Rozlišení označené písmenem „W“ jsou určeny pro širokoúhlé monitory. Zároveň s velikostí rozlišení je možné zvolit obnovovací frekvenci.

Volbou „Disable sound“ můžeme vypnout zvuk syntetizátoru. Volba „16-bit color depth“ spustí demo v režimu 16-bit barevné hloubky. Je zde pouze z důvodů kompatibility. Běh v okně zajistí volba „Run in window“.

V případě potřeby je možno vypnout efekt radiálního rozmázání volbou „No radial blur“, což může zajistit o něco lepší výkon. Stejně tak volba „Low detail“ zajistí snížení kvality radikální degradací komplexnosti scény (zákaz aplikace algoritmu Catmull-Clark u některých modelů, menší počet částic trávy a listí).

Poslední volba „Force 16:10“ vynutí použití širokoúhlého zobrazovacího poměru v případě nastavení neširokoúhlého rozlišení.

10 Finální komprimace

10.1 Velikost před komprimací

Poté, co jsme úspěšně dokončili naše demo nás čeká poslední krok. I přes všechny optimalizace jsme při tvorbě pravděpodobně (i několikanásobně) překročili stanovený limit, což je v pořádku. Finálním krokem u dema s omezenou velikostí je totiž jeho komprimace vybranou externí utilitou.

Komprimační utility pro spustitelné soubory fungují tak, že zvolený spustitelný soubor zkomprimují a výsledek uloží spolu s dekomprimačním algoritmem do nového spustitelného souboru. Při spuštění daného programu je pak jeho obsah na pozadí automaticky dekomprimován a spuštěn, takže uživatel většinou nic netuší.

Náš spustitelný soubor měl výslednou nekomprimovanou velikost skoro 3,7 MB, což se může zdát extrémně mnoho, ale po nahlédnutí do binárního obsahu zjistíme, že konec souboru je prakticky celý vyplněn nulovými hodnotami. Tuto obrovskou velikost způsobilo použití externí knihovny rozhraní DirectSound, kterou vyžaduje syntetizátor V2. Jak naštěstí uvidíme, i s těmito případy si komprimační utility poradí.

10.2 Komprimace

K testování nejlepší efektivity použijeme tři nejčastěji používané utility využívané při tvorbě programů s omezenou velikostí. Následující tabulka 10.1 ukazuje výsledky programů UPX [8], Upack [9] a kkrunchy [10].

Program	Výsledná velikost (byty)
UPX	66 048
Upack	56 324
kkrunchy	54 784

Tabulka 10.1: Výsledky komprese

Výsledek je relativně překvapující. Utilita UPX navzdory faktu, že je ze všech tří použitých nejdéle ve vývoji, nebyla schopna zkomprimovat program pod danou hranici. Utilitám Upack a kkrunchy se podařilo získat i rezervu. V případě utility kkrunchy se jedná o potvrzení faktu, proč je používána v drtivé většině světových demoprodukcí. K této skutečnosti přispívá i fakt, že za jejím vývojem stojí již výše zmiňovaná (kapitola 9.5) demoskupina Farbrausch.

11 Závěr

Výsledek práce relativně přesvědčivě dokazuje, že programová hranice 64 kB ještě stále poskytuje dostatek prostoru pro uchování velkého množství informací a pro realizaci zajímavých audiovizuálních prezentací.

Vývoj projektu přinesl mnoho zajímavých poznatků především z oblasti algoritmizace (Catmull-Clark, efektivní uložení modelových dat) a z oblasti renderovacích technik. Výsledný program zdaleka nedosáhl limitu velikosti a stále je k dispozici 10 kB volného prostoru. Využití zbývajícího místa se přímo nabízí – v programu se prakticky nevyskytují textury (krom oblohy). Většina moderních dem využívá vlastní procedurální texturové generátory, což by mohlo být zajímavým námětem na rozšiřující práci. Další možností pokračování v projektu je implementace vlastního hudebního syntetizátoru a vlastní hudby. Ten měl být původně taktéž součástí tohoto řešení, ale vzhledem k vysoké komplexnosti a náročnosti by toto téma vydalo pravděpodobně na samostatnou práci.

Z hlediska estetiky byl po celou dobu vývoje kladen vysoký důraz na detaily a celkovou souhru jednotlivých částí (převážně hudební synchronizace). Myslím, že výsledek je v tomto směru uspokojivý.

Literatura

[1] WWW stránky – Microsoft Developers Network, *http://msdn.microsoft.com*
dostupná v květnu 2007

[2] WWW stránky - Raymond Filiatreault, Simply FPU
http://www.website.masmforum.com/tutorials/fptute/fpuchap1.htm#cword
dostupná v květnu 2007

[3] WWW stránky – FMOD sound system, *http://www.fmod.org*
dostupná v květnu 2007

[4] WWW stránky – Wikipedia, Catmull-Clark subdivision surface
http://en.wikipedia.org/wiki/Catmull-Clark_subdivision_surface
dostupná v květnu 2007

[5] WWW stránky – Codeproject, Aggressive optimizations
http://www.codeproject.com/tips/aggressiveoptimize.asp
dostupná v květnu 2007

[6] WWW stránky – Mathworld, Barycentric coordinates
http://mathworld.wolfram.com/BarycentricCoordinates.html
dostupná v květnu 2007

[7] WWW stránky – Pouët, V2 Synthesizer system
http://www.pouet.net/prod.php?which=15073
dostupná v květnu 2007

[8] WWW stránky – Ultimate packer for eXecutables, *http://upx.sourceforge.net/*
dostupná v květnu 2007

[9] WWW stránky – The Upack program package
http://www.crystal.chem.uu.nl/~vaneyck/upack.html
dostupná v květnu 2007

[10] WWW stránky – Farbrausch, kkrunchy, *http://www.farbrausch.de/~fg/kkrunchy/*
dostupná v květnu 2007

[11] WWW stránky – Pouët, Synthezze
http://www.pouet.net/prod.php?which=25889
dostupná v květnu 2007

Seznam příloh

Příloha 1. CD s programem, zdrojovými soubory, videovými záznamy, pomocnými utilitami a modelovými daty