VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

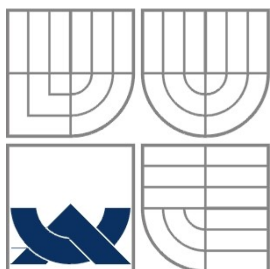
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MINIMALISTICKÁ REPREZENTACE MODELU AREÁLU BOŽETĚCHOVA

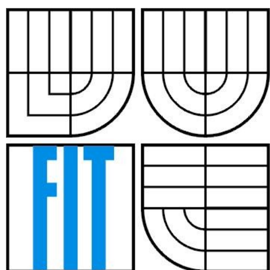
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ KRÁL
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MINIMALISTICKÁ REPREZENTACE MODELU AREÁLU BOŽETĚCHOVA

MINIMAL REPRESENTATION OF THE BOŽETĚCHOVA COMPLEX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ KRÁL
AUTHOR

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

BRNO 2009

Abstrakt

Práce se zabývá tvorbou grafických aplikací s omezenou velikostí. Popisuje techniky vhodné pro kompresi polygonálních modelů. V druhé části potom postup jejich praktického využití při vytvoření scény v 3D modelovacím prostředí a postupném exportu a převodu těchto dat do spustitelného souboru. V závěrečných kapitolách se věnuje optimalizaci kompilace zdrojových kódů v jazyce C a možnostem komprese spustitelných souborů.

Abstract

The document describes developing graphical application with limited size. It describes suitable techniques for a polygonal mesh's compression. The second part is focused on practical usage of this techniques for developing scene in 3D modeling environment and also describes how to transfer this model to the executable file. The work attends to optimalizations of source code compilation and executables compression at the final chapters.

Klíčová slova

Minimalistická reprezentace, aplikace s omezenou velikostí, intro, demo, OpenGL, Božetěchova, komprese modelů, LOD, dělení ploch, triangle strip, procedurální textury, model, scéna, Rhinoceros 3D, Rhinoscript, PERL, optimalizace kódu, komprese exe souborů, UPX.

Keywords

Minimal Representation, application with limited size, intro, demo, OpenGL, Božetěchova, mesh compression, LOD, surface subdivision, triangle strip, procedure textures, model, scene, Rhinoceros 3D, Rhinoscript, PERL, source code optimalization, executables compression, UPX.

Citace

Tomáš Král: Minimalistická reprezentace modelu areálu Božetěchova, diplomová práce, Brno, FIT VUT v Brně, 2009

Minimalistická reprezentace modelu areálu Božetěchova

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. A. Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Tomáš Král
Datum 25.5.2009

Poděkování

Rád bych zde poděkoval Ing. A. Heroutovi, Ph.D. jako vedoucímu projektu za jeho čas věnovaný konzultacím, rady ve věcech odborných a neméně důležitý přátelský přístup. Poděkování také patří Lukáši Duránikovi za ochotnou pomoc při získání technických podkladů pro vytvoření modelu areálu Božetěchova.

© Bc. Tomáš Král, 2009
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah.....1
1 Úvod.....2
1.1 Grafické aplikace s omezenou velikostí.....2
1.2 Motivace.....3
2 Techniky komprese modelů.....5
2.1 Reprezentace modelu.....5
2.2 Uložení světelných zdrojů.....14
2.3 Komprese materiálů a textur.....14
3 Vytvoření modelu.....17
3.1 Použité nástroje.....17
3.2 Ukázky komprimovaných struktur.....22
4 Zobrazovací program.....24
4.1 OpenGL.....24
4.2 Pohyb kamery.....24
4.3 Teselace.....26
4.4 Optimalizace kódu.....28
4.5 Kompresní programy.....29
5 Závěr.....31
Literatura.....33

1 Úvod

Minimalistické aplikace nejsou jistě doménou dnešní doby neomezených možností. Nezasvěcený čtenář by se tak mohl podivit nad dokumentem, který se mu dostal do ruky. Proč v době, kdy je nejmenším používaným médiem CD s kapacitou ve stovkách megabajtů bychom měli bazírovat nad rozdíly velikosti programů v řádu kilobajtů. Na tyto otázky se čtenáři pokusí odpovědět kapitola 1.2, která diskutuje možnosti využití grafických aplikací s omezenou velikostí. Nesmíme ovšem opomenout se dotknout historie. Proto hned v následující kapitole 1.1 se věnuji *demoscéně*. Čtenář se v ní dozví co to demoscéna je, možná bude i překvapen bohatou historií. Aplikace s omezenou velikostí tu totiž byly již v době, kdy jejich velikost byla spíše běžnou než minimalistickou a kdy byly spíše nutností než soutěží ve schopnostech programátorů.

V další části se tento text věnuje programátorským technikám, jejichž využití je vhodné pro vytvoření minimalistického modelu. Ke každé z nich potom diskutuji výhody a nevýhody jejího použití na konkrétním případu – minimalistické reprezentaci školní budovy areálu FIT VUT v Brně – Božetěchova. Třetí kapitola potom popisuje postup při vytvoření modelu, použité nástroje od 3D modelovacího programu Rhinoceros přes skriptovací jazyky až po samotnou knihovnu – API – v jazyce C pro vývojáře zobrazovacích programů. Poslední kapitola se věnuje implementaci zobrazovacího programu a možnostem komprese samotného kompilovaného kódu.

1.1 Grafické aplikace s omezenou velikostí

V úvodu se sluší si položit otázku, co to vlastně jsou grafické aplikace s omezenou velikostí? Proč v době, kdy 1 GB datového média má stejnou hodnotu jako 30 vteřin práce programátora a kdy konečně i do českých domácností pronikl skutečně vysokorychlostní internet, máme plýtvat drahocenným časem na vytváření minimalistických modelů?

Na dotaz „grafické aplikace s omezenou velikostí" nám přední český i světový vyhledávač neodpoví žádným relevantním výsledkem hledání, což naznačuje velmi nízkou rozšířenost této problematiky v běžném životě. Pokud dnešní hry nebo programy přerostou velikost jednoho DVD, jednoduše přidáme další. Cena je zanedbatelná v porovnání s časem vývojáře, který by byl nutný na optimalizaci programu/datových struktur, a uživatel má alespoň pocit, že za své peníze toho víc získal.

Obrázek 1: ukázky pokročilé grafické aplikace s omezenou velikostí [181 KB]

Tvorba grafických aplikací s omezenou velikostí je tedy spíše doménou profesionálů, nadšenců. Umělecká hodnota zde převyšuje možnosti komerčního využití. Technická stránka takové aplikace je důležitější než její obsah. Kvalita převyšuje nad kvantitou. Autoři takovýchto aplikací tvoří tzv. *demoscénu*, jejíž vznik je datován již do 80. let dvacátého století. Od té doby se podoba vytvářených děl měnila stejně dynamicky jako schopnosti hardwarových prostředků, které byly aktuálně k dispozici. Co se však nezměnilo je podstata jejich tvorby. Grafické aplikace s omezenou velikostí, zkráceně řečeno *intra* či *dema*, mají prezentovat autorovy schopnosti a technické znalosti

z této oblasti. Tvůrci inter se tak snaží podobně jako sportovci překonávat rekordy. Posunout hranice zase o kousek dál a zhustit do směšného datového objemu vizuelně velmi působivé scény a animace.

V této činnosti se jako snad ve všech ostatních oborech lidské činnosti soutěží, a to v nejrůznějších kategoriích. Nejznámější je kategorie do 64 kB, na kterou jsou vypsány i další témata diplomových prací. Často se setkáme také s kategorií do 128 kB. Opravdu kvalitní *intra*, nebo chcete-li *dema*, které se dostanou na čelní pozice v hodnocení svých kategorií, pak nejsou prací jednotlivců, ale celých týmů profesionálů z různých oborů. Mezi nimi potom nechybí režisér, zvukař či animátor/grafik. V neposlední řadě také programátor, resp. ve většině případů tým programátorů, kteří mají za úkol vyždímat z dostupného hardware a softwareových nástrojů co nejvíce.

1.2 Motivace

Tato práce nemá ve svém zadání žádné velikostní omezení, jak bylo uvedeno v předchozí kapitole. Neklade si ani za cíl soutěžit s týmy autorů grafických inter. Jejím cílem je prozkoumání technik používaných při tvorbě aplikací s omezenou velikostí, navržení obecného postupu pro tvorbu takové aplikace z připraveného modelu a navržení takového rozhraní, které by umožnilo nejen další pokračování a zdokonalování zde popsaných postupů, ale i možnost spolupráce celého týmu vývojářů. Výstupem projektu pak bude model areálu Božetěchova v optimálně zvoleném poměru *grafická kvalita x velikost aplikace x rychlost spuštění*.

Neméně důležité je pak položit si otázku, zda má vůbec smysl investovat čas do projektu, který jde opačným směrem než aktuální trendy vývoje. Přeci jen dnes přepočítáváme vše na peníze a čas programátora je dražší než datové médium. Jistě, je to pravda. Tento projekt si také neklade za cíl měnit způsob vytváření složitých 3D programů či her. Takové programy jsou složitější a vytvářeny na stále vyšší úrovni abstrakce, kde je velmi obtížné sledovat efektivní využití každého kilobajtu výsledného souboru. Využití závěrů této práce vidím především v prezentační oblasti. Například vytvoření 3D průvodce budovou, který by byl k dispozici ke stažení na internetových stránkách instituce – v našem případě vysokoškolské fakulty. Případný zájemce by pak nemusel být omezen dny otevřených dveří a mohl by kdykoli nahlédnout do míst, kam by se za běžného provozu instituce nedostal. V tomto případě již má smysl optimalizovat velikost takového souboru – prezentace. V první řadě proto, že lidé nejsou ochotni čekat. Chtějí co nejvíce informací v co nejkratším čase. V této fázi můžete namítnout, že rozdíl několika kilobajtů kódu ovlivní rychlost stažení jen minimálně, přece jen máme dnes již k dispozici domácí internetové přípojky s rychlostmi downloadu v řádu megabitů za sekundu. Takovéto rychlosti jsou ovšem běžnější spíše ve větších městech, nesmíme tedy zapomínat ani na ostatní uživatele. Když pomineme ADSL připojení, které je již dnes velmi kvalitní a svým pokrytím k dispozici na téměř 2 milionech telefonních linek (>95%), přijdou na řadu bezdrátové sítě. Ty jsou v dnešní době velmi populární. Nikdo nechce mít doma roztahané kilometry nevzhledných šedých kabelů, aby se mohl připojit k internetu v každém koutě svého domu pomocí notebooku. Někdo si může chtít absolvovat virtuální prohlídku se svými přáteli v kavárně nebo využít volného času na chatě, kde má k dispozici pouze připojení přes mobilní telefon.

V prvním případě domácího wifi připojení není s velikostí až takový problém, zvláště s přicházející normou 802.11n, která přináší propustnost minimálně 100 Mbit/s. I přesto je ale zbytečné uživatele zatěžovat prezentacemi řádově v desítkách megabajtů. V druhém případě veřejného přístupového bodu v kavárně je rychlost ještě více omezena počtem připojených uživatelů, kteří linku současně využívají. Nejdůležitější ze jmenovaných případů je ovšem mobilní připojení.

Mobilní připojení se stává stále oblíbenější a je dostupné stále širší skupině obyvatel. Jeho používáním se vracíme o mnoho let zpět (samozřejmě pouze z hlediska propustnosti), kdy bylo běžné modemové připojení o rychlosti řádově v kilobajtech (desítkách kilobajtů) za vteřinu. Jsou sice k dispozici i rychlejší varianty v podobě sítí třetí generace, ale s jejich zavedením někteří naši operátoři ani příliš nespěchají, ti ostatní teprve postupně rozšiřují pokrytí od větších měst. Situaci také nejlépe vystihují trendy vývoje webových stránek, jejichž průměrná doba potřebná ke stažení v průběhu let vývoje neklesla ani přes vzrůstající rychlosti internetových přípojek. Příčina je v tom,

že ruku v ruce se zvyšováním rychlosti internetových přípojek se navyšovala spíše grafická složitost a rost textový obsah (reklamy nevyjímaje) a tím i velikost webových prezentací. Až v posledních letech s rozvojem mobilního internetu se postupně zavádějí mobilní a textové verze stránek. Tento trend se poukouší sledovat i projekt minimalistické reprezentace školní budovy, který máte nyní v rukou.

V neposlední řadě se také podívejme na problematiku z trochu jiného pohledu. Uvažme vytvoření 3D virtuálního průvodce a jeho umístění na serveru ke stažení. Uživatel je umístěn na rychlé lince, takže rozdíl 100 KB je pro něj nepozorovatelný. Vynásobme si ovšem toto číslo (které by klidně mohlo být i ve většině případů větší) počtem uživatelů, kteří by tuto službu mohli využít, a dostáváme se do mnohem zajímavějších čísel, které by mnohému správci serveru či sítě mohly přidělat alespoň trochu práce. Další zajímavé srovnání nabízí tabulka 1, která srovnává vyhledávání na serverech googlu s činnostmi produkujícími stejné množství emisí CO_2 (což úzce souvisí se spotřebou energie). Pojďme ještě o něco dále a uvažujme daleko za rámec tohoto projektu, ba i internetu v podobě jak jej známe nyní. S dalším rozvojem internetu jistě přijde doba, kdy 3D prostředí bude naprosto běžné. Můžeme uvést příklady mobilní navigace uvnitř rozsáhlých budov, institucí nebo třeba na letišti, online nakupování ve virtuálních obchodech z pohodlí domova a mnoho dalšího. V takové budoucnosti budeme každou úsporu násobit miliony či miliardami online uživatelů na celém světě a i přes stále úspornější hardware může právě kvůli četnosti využití mít každá (byť nepatrná) minimalizace velký vliv nejen na pohodlí uživatelů, ale i na finanční náklady provozovatele, což přímo úměrně souvisí s ekologií. Ekologii zmiňuji z důvodu, že bude postupem času více a více skloňována ve společnosti a s rostoucím počtem elektronických zařízení objemu přenesených dat to budou právě tyto malé úspory, které v globálním měřítku přinesou nejlepší výsledky.

Activity	**Google Searches**
CO_2 emissions of an average daily newspaper (PDF) (100% recycled paper)	850
A glass of orange juice	1,050
One load of dishes in an EnergyStar dishwasher (PDF)	5,100
A five mile trip in the average U.S. automobile	10,000
A cheeseburger	15,000
Electricity consumed by the average U.S. household in one month	3,100,000

Tabulka 1: srovnání počtu hledání na googlu s podobnou činností z hlediska emisí CO_2, zdroj [12]

Tyto úvahy jsou však opravdu nad rámec této práce, která si neklade za cíl snižovat emise skleníkových plynů, mají však přispět k myšlence, že neustálé zvyšování výkonu počítače nemusí být zneužíváno neefektivností aplikací a že v budoucnu reálně může být tlak na vytváření minimalistických a efektivních aplikací nejen ze strany fyzikálních limitů hardware, ale i např. z hlediska ekologie. Má tedy smysl se jimi již nyní zabývat. Čtenář se v dalších kapitolách dozví, jaké jsou v dnešní době možnosti úspory a limity grafických aplikací s omezenou velikostí a v závěru potom proběhne diskuze, zda takovéto úspory mají smysl jak dnes, tak z dlouhodobého hlediska. Praktickým výstupem této práce bude, jak bylo zmíněno v úvodu této kapitoly, aplikace podobná *demu* či *intru*, včetně nástrojů vytvořených při jejím vývoji, které budou moci být dále využívány buď při vytváření dalších modelů podobného charakteru nebo jako základ pro vytvoření komplexnějších a obecnějších nástrojů.

2 Techniky komprese modelů

Tato kapitola pojednává o technikách komprese modelových dat. Je rozdělena do tří částí. V první se věnuji uložení geometrických dat, tedy souřadnic vrcholů polygonů a texturovacích souřadnic. Druhá část se krátce věnuje světelným zdrojům. Největšímu úspornému potenciálu – materiálům a texturám – se věnuje třetí a poslední část této kapitoly. Popisuje možnosti procedurálního generování textur.

Veškeré techniky popsané v této kapitole nebyly sice použity v současné verzi programu pro minimalistické uložení, byly však zvažovány minimálně jako alternativa, nebo jejich implementace byla odložena výhledově na příští verze programu. Ke každému zde uvedenému postupu uvádím jeho výhody a nevýhody, jak z obecného hlediska, tak i s přihlédnutím ke konkrétnímu zadání projektu, tedy vytvoření modelu školního areálu, který má svá specifika.

2.1 Reprezentace modelu

Každý objekt reálného světa je těleso zaujímající určitý objem. Jako takové by bylo logické jej i reprezentovat při modelování. To je přístup objemové reprezentace dat. Spojité reálné těleso segmentuje na diskrétní voxely, které uspořádá do pravidelné trojrozměrné mřížky (podobně jako např. fotoaparátem pořídíme 2D obraz). Podle druhu dat a požadavků na jejich uspořádání nemusíme využívat pouze pravidelné mřížky. Počítačová grafika zná i nepravidelné pravoúhlé, strukturované či zcela nestrukturované mřížky anebo naopak striktnější kartézské, jak je uvedeno v literatuře [1]. Základní charakteristikou objemových dat je, že reprezentujeme jak hraniční body tělesa, tak i vnitřní body.

Přirozenější ovšem je reprezentovat těleso popisem jeho povrchu. Z toho vychází i častěji v počítačové grafice využívaná hraniční reprezentace těles (*boundary representaiton, B-rep*). Výhody spočívají v podpoře rychlého zobrazení grafickými procesory, snadnější zpracování takových dat a pro naše účely především nižší datový objem.

Povrch tělesa ovšem můžeme reprezentovat dále několika způsoby, lišícími se v množství informací, které nám o tělese poskytují a tím i velikosti uložených dat, dále přesností zobrazení (popisu) a v neposlední řadě i podpoře přímého zobrazení grafickým hardware.

2.1.1 CSG strom

V první fázi tohoto projektu budeme školní budovu modelovat pomocí programu Rhinoceros 3D, což je jeden z mnoha dostupných CAD programů (*Computer Aided Design*). Obecně v těchto programech využíváme většinou popis tělesa pomocí tzv. konstruktivní geometrie (*CSG, Constructive Solid Geometry*), která reprezentuje těleso stromovou strukturou základních geometrických primitiv. Tato metoda spočívá v tom, že máme v programu definované funkce pro vytváření základních geometrických útvarů (bod, přímka, NURBS plocha, koule, kvádr, atd...) a funkce pro jejich kombinaci – tedy množinové operace sjednocení a rozdílu – jak nad jednotlivými primitivy, tak nad celými CSG stromy. Kombinováním primitiv pak modelujeme matematicky přesný povrch tělesa. Tato reprezentace je výhodná především pro konstruktéry, protože jednak tento postup práce je blízký běžnému postupu konstrukce těles a jednak pro vyrobitelné těleso je vhodné mít přesný popis tělesa nezávisle na měřítku. Pro minimalistický program a zobrazení tato reprezentace již vhodná není, protože ji nelze přímo zobrazit pomocí grafického hardware a museli bychom tedy do programu implementovat složitější algoritmy, které by sami o sobě zabraly větší množství dat.

2.1.2 Hraniční reprezentace

Pro potřeby minimalistického exportu zvolíme některý vhodný typ hraniční reprezentace. Zcela jistě to nebude bodová reprezentace, protože její základní charakteristikou je velký datový objem. Tato se používá hlavně při digitálním snímaní reálných objektů nebo může být výstupem některých algoritmů.

Další možností je hranová reprezentace, která je v dnešní době spíše historií. Používala se dříve kvůli nižším nárokům na hardware. Dnes ji známe pouze z CAD programů pod pojmem *drátový model*, kde slouží pro možnost rychlého náhledu nebo nahlédnutí dovnitř tělesa. Je také velmi úsporná, bohužel na úkor topologických informací, což ji činí nejednoznačnou. Z drátového modelu nemůžeme přesně určit reálnou podobu tělesa.

Použitelným typem je strukturovaná plošková reprezentace, u které vytváříme seznam vrcholů přestavující geometrická data modelu, následně pak seznam hran a ploch definujících topologii. Nejsložitější strukturou této reprezentace je seznam hran, díky němuž je tato reprezentace také často nazývána *okřídlená hrana*. Výhodou tohoto přístupu jsou velmi komplexní informace, které nám o modelovaném objektu poskytuje. Pro naše účely však nevýhoda spočívá v množství topologických dat, které zabírají dle literatury až 75 % paměti. Mnohé z nich nejsou pro náš projekt podstatné, takže je lepší přejít k jednoduché ploškové reprezentaci.

Jednoduchá plošková reprezentace uchovává stejně jako předchozí seznam vrcholů, tedy geometrii, rozdíl spočívá v uložení topologických dat. Z nich ukládáme pouze seznam ploch, kde každá plocha obsahuje pouze indexy do pole vrcholů. Chybí tedy informace o sousedních plochách nebo společných hranách dvou ploch. Výhoda je, jak již bylo zmíněno, datová úspora. Orientace jednotlivých plošek, tedy směr jednotkového normálového vektoru, je určena při zobrazení plošky na základě pořadí uložených vrcholů polygonu. Obvyklé je uložení proti směru hodinových ručiček. Jednoduchá plošková reprezentace také umožňuje pracovat jak se sítí trojúhelníků, tak i obecných polygonů. Budeme ji tedy v projektu používat a v dalším textu a popisu algoritmů budu předpokládat použití právě této datové reprezentace. Podrobný popis zde uvedených reprezentací naleznete také v literatuře [1].

2.1.3 Level of detail

Úroveň detailu modelu chápeme jako bezrozměrnou subjektivní veličinu. Při minimalistické reprezentaci modelu se tedy snažíme o vypuštění co největšího počtu detailů, resp. snížení počtu polygonů modelu tak, aby výsledný vytvořený model si zachoval vizuelně přijatelnou kvalitu.

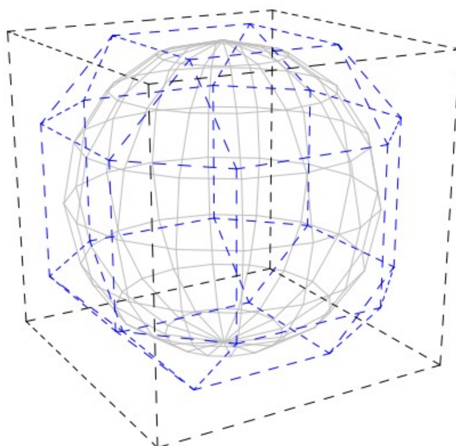
Při minimalistické reprezentaci areálu Božetěchova se na LOD můžeme dívat z několika úhlů. Základní nejběžnejší pohled je snížení počtu polygonů viditelných částí modelu. Vzhledem k subjektivitě této techniky a tedy její obtížné automatizaci, stanovujeme úroveň detailu ručně již při vytváření modelu v 3D modelovacím prostředí – v našem případě programu Rhinoceros. Z válcových podpěrných sloupů potom budou tělesa s osmiúhelníkovou podstavou, atd...

Dalším pohledem na úroveň detailu je v našem případě zcela vynechání některých částí modelu. Předpokládejme využití výsledného programu k prezentačním účelům fakulty – virtuální „den otevřených dveří“. V takovém případě budeme chtít podrobně zpracovat ty části areálu, které budou cílového uživatele zajímat, jako jsou přednáškové místnosti, studovny či laboratoře a ostatní lze do modelu zakomponovat buď s nízkou úrovní detailu nebo vůbec – výtahové šachty, kotelny, ale i rovné střechy a technické místnosti v budovách. V neposlední řadě pro prezentační účely nebudeme trvat ani řádově na centimetrové přesnosti vytvářeného modelu, takže můžeme jednotlivé detaily upravit tak, abychom mohli dosáhnout vyššího stupně komprese – jinými slovy optimalizovat pro námi zvolené kompresní algoritmy bez ohledu na přesné reflektování reality.

Výhody a nevýhody této techniky jsou zřejmé. Výhodou je snížení množství dat, nevýhodou vizuelní kvalita. U této metody si tedy neklademe otázku zda ji použít či ne, ale v jaké míře.

2.1.4 Dělení ploch

Úzce související technikou s LOD pro ušetření dalších bajtů výsledného modelu je algoritmus dělení ploch. Jeho kouzlo spočívá v tom, že ve statických datech si uložíme jen velmi hrubou kostru modelu, tedy podobně jako bychom si model uložili ve velmi nízké LOD (platí pouze pro interpolační schémata). Výhoda této techniky ovšem spočívá v zobrazovací části, kde můžeme aplikací algoritmu dělení ploch velmi hrubou kostru rozgenerovat do podrobného modelu a tedy i přes velmi nízkou úroveň detailu uloženého modelu získat vizuelně přijatelný výsledek. Tento algoritmus je rekurzivní, aplikací většího počtu kroků se budeme limitně blížit matematickému popisu povrchu objektu. V praxi je však dostačující jen několik kroků k získání uspokojivých výsledků.



Obrázek 2: černá - kostra, modrá - první krok,

šedá - výsledná koule; inspirace z [1]

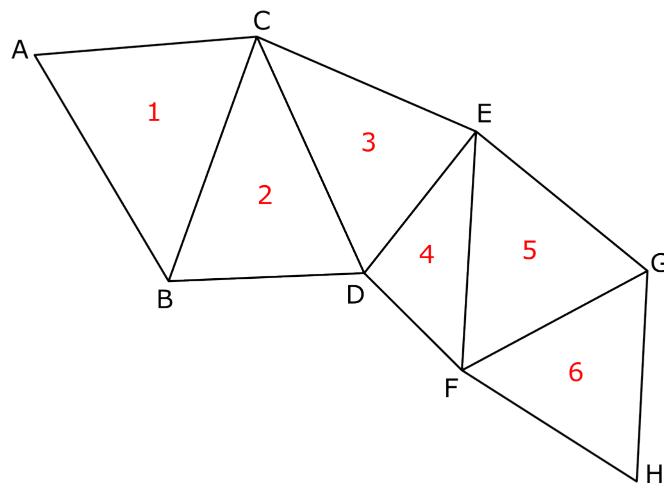
Pro subdivision existuje několik algoritmů. Mezi nejznámější patří *Catmull-Clark*, *Doo-Sabin*, ale také třeba *Loop* nebo *Butterfly*. Poslední dva uvedené algoritmy jsou odvozením předchozích pro specifický typ sítě. Hlavní rozdíly v těchto algoritmech představují tzv. dělící schémata, která lze kategorizovat podle toho, zda dělení probíhá nad vrcholy nebo nad ploškami, zda vstupem může být libovolná topologie nebo schéma pracuje pouze se sítí trojúhelníků a v neposlední řadě zda se jedná o aproximační nebo interpolační schéma. Na obrázku 2 vidíte příklad aproximačního dělení nad vrcholy. Pro interpolační schéma by body kostry ležely na matematicky přesně definovaném povrchu výsledného objektu. Pozorný čtenář si také položí otázku, zda je nějakým způsobem vyřešeno jen částečné vyhlazování povrchů (rozumějte vyhlazování pouze v určitých oblastech modelu). V architektuře se nabízí příklad sloupů, které bychom chtěli vyhladit – zakulatit – po obvodu, ale jejichž hlavám musí zůstat zachované ostré hrany. Toto je v dělících algoritmech vyřešeno různými stupni vrcholů, které lze definovat. Bližší popis algoritmů i dalších možností najdete ve studijní literatuře [14].

A abychom nepěli samou chválu, jaké jsou nevýhody této techniky? Za nevýhodu můžeme označit složitější vytvoření modelu. Nelze pouze snížit počet jeho polygonů, ale je třeba najít vhodnou kostru, ze které bude možné vygenerovat věrnou podobu původního modelu. Pro využití v architektuře se neobejdeme ani bez definování stupně vrcholů, abychom některé hrany mohli zachovat ostré a některé naopak vyhladit. To ovšem přináší nutnost ukládat další data, resp. definovat buď komplexnější datové struktury nebo více typů struktur. S tím souvisí i potřeba při dělení znát topologické informace o sousedních polygonech. Tyto bychom také museli buď uložit do datového formátu nebo je složitěji dopočítávat v klientském programu. Nevýhodou je také nutnost implementace patřičného algoritmu dělení ploch ve výsledném programu a vytvoření struktury pro takto vytvořené části modelu, což obojí znamená další datové nároky. Nesmíme ani zapomínat

na otexturované objekty. Pokud v 3D modelovacím programu vytvoříme objekt tvaru krychle a potáhneme jej texturou, exportují se nám s ním i texturovací souřadnice pro vymodelovanou krychli, které jsou zřejmě velmi rozdílné od souřadnic na výsledné kouli. Správné namapování těchto souřadnic by bylo tématem pro samostatnou diplomovou práci. Když toto vše dáme do souvislosti s charakterem areálu, tedy z velké části moderních budov s minimem zaoblených ploch, které by bylo potřeba vyhladit, zjistíme, že využitím tohoto algoritmu spíše zvětšíme velikost výsledného souboru. Budeme se tedy raději věnovat dalším technikám, které mohou být pro náš konkrétní případ užitečnější.

2.1.5 Triangle strip

Triangle strip je posloupnost trojúhelníků spojených hranou, tedy takových trojúhelníků, kde každé dva sousední trojúhelníky sdílí dva ze tří vrcholů. V počítačové grafice se využívá především pro urychlení vykreslování a snížení počtu uložených vrcholů modelu (odstraní se duplicity). Stejně tak je tato technika vhodná i pro minimalistické uložení modelu, kdy můžeme efektivně snížit počet uložených vrcholů modelu až téměř na třetinu.

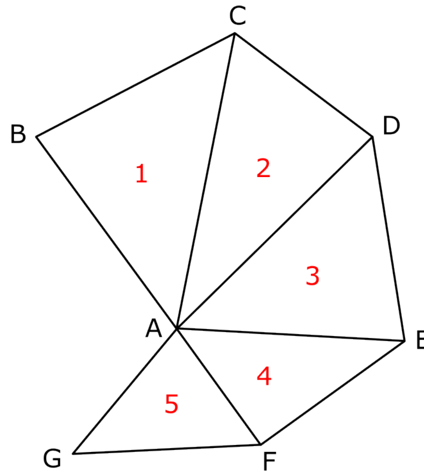


Obrázek 3: ukázka triangle strip, uložené vrcholy jsou ABCDEFGH

V praxi s takovouto úsporou ovšem počítat nemůžeme. Důvodem je nemožnost převést obecný model na jediný triangle strip. Řešením je buď vytvoření více proužků pro jeden model nebo začlenění trojúhelníků s nulovou plochou, které budou při vykreslení vyřazeny. Začlenění více triangle strips k jednomu modelu znamená vytvoření složitější datové struktury a s tím spojené datové nároky. Oproti tomu druhý přístup znamená „pouze“ uložení jednoho vrcholu navíc pro každý přechod mezi různými proužky. Pro naše potřeby je tedy vhodnější druhý přístup s trojúhelníky s nulovou plochou, který sice znamená vyšší zátěž při vykreslování modelu, což ovšem není v tomto případě tak důležité jako datová velikost. Tento základní popis naleznete v encyklopedii [13], o použití triangle strips v OpenGL se můžete dočíst např. zde [7].

Na druhou stranu jak jsme popsali v kapitole 2.1.3 se budeme snažit sestavit model tak, aby obsahoval co nejméně polygonů. Ve výsledku tak budeme mít stěnu, kterou budeme exportovat samostatně složenou např. jen ze dvou trojúhelníků nebo jednoho čtyřúhelníku. Protože jak je popsáno v kapitole 3.1.2, budeme v exportovaných modelech také využívat čtyřúhelníky, což je další komplikace pro využití tohoto postupu. Když uvážíme i fakt, že pro uložení topologických informací o vrcholu (počet vrcholů se nezmění, tedy ani objem geometrických dat) využíváme indexu do pole vrcholů v rozsahu 0 až 255 a tedy pro každý vrchol navíc potřebujeme pouze 1 byte, není tato metoda pro naše účely vůbec zajímavá.

V počítačové grafice ještě existuje alternativa k této metodě, nazvaná triangle fan. Tato je velmi podobná předchozí popsané, s tím rozdílem, že první uložený vrchol znamená střed pomyslného vějíře a každý další vrchol potom jeho kraj.



Obrázek 4: ukázka triangle fan

Z hlediska úspory dat není tato technika tak atraktivní jako předchozí, protože jsme prostorově omezeni právě jedním společným bodem – středem „vějíře“. S přihlédnutím k charakteru ukládaných dat je tato metoda nevhodná pro kompresi prostorových dat, resp. je plně nahraditelná použitím metody *trinagle strips*.

2.1.6 Způsob uložení v paměti

Další oblastí s velmi vysokým potenciálem úspory dat je způsob uložení jednotlivých vrcholů. Běžně jsou souřadnice vrcholů polygonů uloženy jako reálná čísla s plovoucí řádovou čárkou. Vhodným postupem můžeme reálná čísla převést na čísla s menším počtem bitů. Provedeme to tak, že vybraný model uzavřeme do jednotkové krychle. Tuto jednotkovou krychli budeme považovat za nový souřadný systém, v rámci kterého budeme ukládat souřadnice jednotlivých vrcholů modelu. Jako reálná čísla s plovoucí řádovou čárkou si uložíme pouze reálné souřadnice ve scéně bodu [0,0,0] a [1,1,1] jednotkové krychle, tzv. *bounding boxu*. Hrany krychle potom rozdělíme na tolik dílků, jaká je požadovaná přesnost uložení vrcholů modelu. Nejvhodnější je uložit každou souřadnici jako jeden byte, tedy hrany jednotkové krychle rozdělit na 255 dílků. Označíme-li *max* jako bod s největší hodnotou všech souřadnic a *min* jako naopak bod nejmenší, výpočet nové celočíselné souřadnice libovolného bodu uvnitř bounding boxu vypočítáme jako:

$$new_x = floor(255*(old_x-min_x)/(max_x-min_x));$$
$$new_y = floor(255*(old_y-min_y)/(max_y-min_y));$$
$$new_z = floor(255*(old_z-min_z)/(max_z-min_z));$$

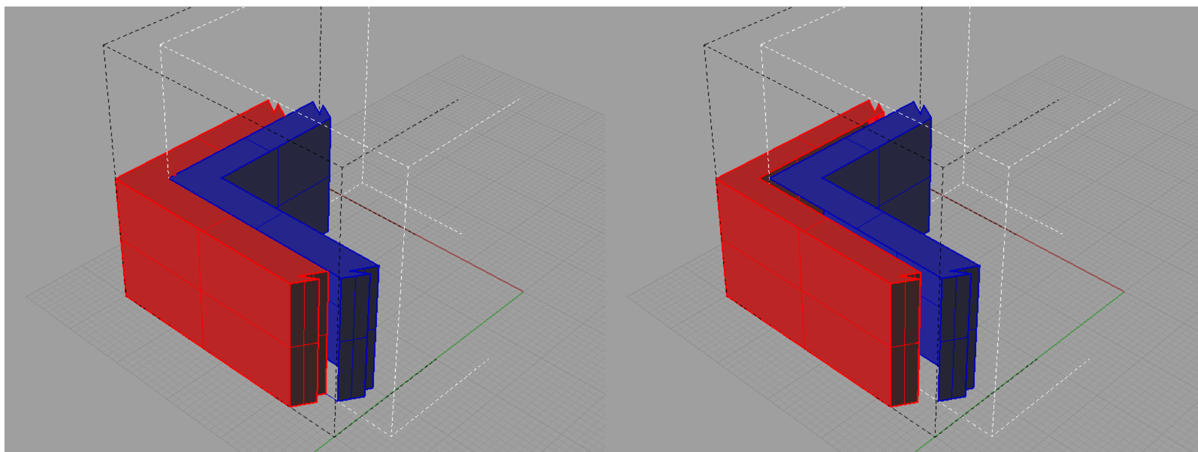
Při dekompresi postupujeme obráceně. Jak je vidět, zaokrouhlením přicházíme o část informace, jedná se tedy o ztrátovou kompresi. Tímto způsobem musíme v komprimované struktuře uložit krajní souřadnice bounding boxu bez komprese, což nám zabere 2*3*4 = 24 B, potom místo běžného uložení vrcholu na 3*4 = 12 B jej můžeme uložit na 3*1 = 3 B. Pro N uložených vrcholů tedy získáme úsporu 9*N – 24 B. Vzhledem k tomu, že N ≥ 3 (musíme uložit alespoň jeden trojúhelník) vidíme, že za každých podmínek získáme úsporu dat. Procentuelně lze vyjádřit nová velikost dat jako:

$$100 * (N + 8) / 4N \quad [\%]$$

kde N je počet uložených vrcholů.

Tato metoda má však i své negativní stránky. Především se jedná o ztrátovou kompesi. Po rozgenerování dat zpět z celočíselných souřadnic do reálných čísel s plovoucí řádovou čárkou získáme nepřesné údaje ovlivněné chybou po zaokrouhlení. Vzhledem k tomu, že nevytváříme matematicky přesný model, ale intro určené pouze pro vizuelní prezentaci, tak nám ztráta přesnosti nevadí. Můžeme však narazit na dva problémy.

Jednak nám ve výsledném modelu mohou vznikat „díry“ a to v případě, kdy hrany dvou ploch v původním modelu na sebe navazovaly, ale pro každou plochu byla zvolena jiná jednotková krychle, jako je to vidět na obrázku 5. V takovém případě mohou být souřadnice původně shodných vrcholů zaokrouhleny jiným způsobem a mezi původně navazujícími plochami vznikne mezera.



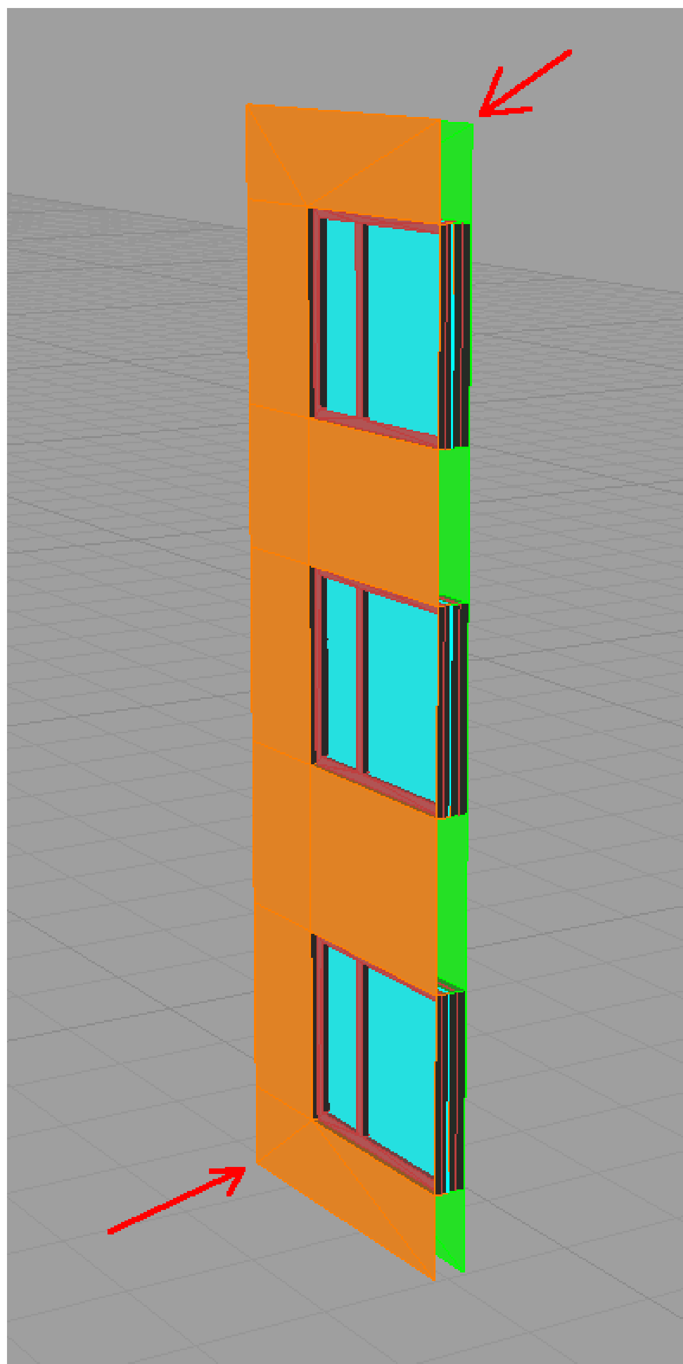
Obrázek 5: vlevo předloha s jednotkovými krychlemi, vpravo vzniklá mezera po zaokrouhlení

Tomuto lze předejít vhodným zvolením jednotkové krychle. Pokud jedna krychle bude podmnožinou druhé, lze tyto krychle sjednotit, čímž zajistíme stejné zaokrouhlení obou souřadnic a tedy i plynulé navázání ploch i v komprimovaném modelu. V druhém případě, kdy se obě jednotkové krychle dotýkají stěnou, k tomuto jevu nedochází, protože krajní souřadnice nejsou postiženy ztrátou zaokrouhlením. Nejlépe je to vidět po dosazení do rovnice pro přepočet za původní souřadnici old_x postupně jak maxima, tak minima:

```
newx = floor(255*(oldx-minx)/(maxx-minx));
newx = floor(255*(maxx-minx)/(maxx-minx)) = floor(255) = 255
newx = floor(255*(minx-minx)/(maxx-minx)) = floor(0) = 0
```

Další problém také souvisí s vhodným zvolením jednotkové krychle. Pokud zvolíme jednotkovou krychli tak, že její krok je větší než v ní obsažené plochy, pak tyto plochy zmizí, resp. se zmenší do jediného bodu. Pokud tedy chceme komprimovat objekty s přesností na 10 cm, pak bychom neměli volit hranu krychle větší než 25 metrů. Nejlépe tento problém ilustruje obrázek 7.

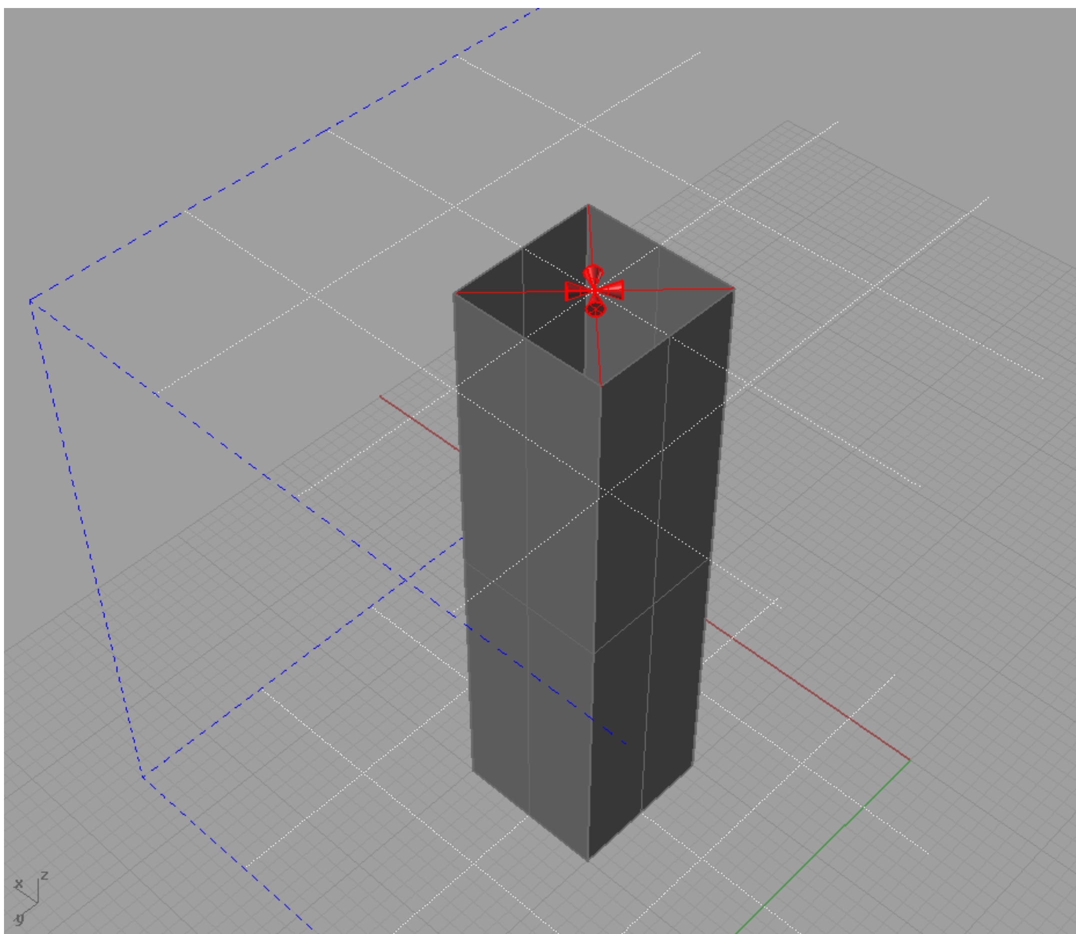
Z hlediska ztrátovosti komprese by bylo ideální až po celkovém dokončení modelu jej pokrýt pravidelnou sítí bounding boxů o pevné délce hrany. Pomocí této sítě celý model rozřezat a jednotlivé bounding boxy (resp. modely a části modelů v nich obsažené) exportovat zvlášť. V takovém případě by ztrátovost komprese nebyla okem patrná. Takové řešení má však i svá úskalí. Jednak je to obtížně prakticky realizovatelné. Potom by nám takový přístup zbytečně rozdělil některé části modelu a přistupoval by tak k nim zvlášť (zvlášť uložení bounding boxu, odkazu na materiál, návaznost textury, atd...). V neposlední řadě by se takováto technika hůře kombinovala s dalšími kompresními technikami, jako je např. opakování stejných částí popsané v kapitole 2.1.7. Zbývá tedy určovat bounding boxy pro každou část modelu zvlášť a testováním dospět k ideálnímu výsledku.



Obrázek 6: Ukázka určení společného bounding boxu

Na obrázku 6 je vidět příklad z praxe, kdy vyřízneme část stěny a pro všechny viditelné části (oranžová omítka, zelená vnitřní stěna, modrá sklo okna a tmavě červená rám) zvolíme jeden společný bounding box tak, jak naznačují šipky. Veškeré vrcholy uvnitř bloku budou zaokrouhleny stejným způsobem a nestane se tak, že by například mezi rámem okna a stěnou vznikla viditelná mezera. Metodou kopírování pak vytvoří tento blok celou stěnu budovy.

Je nutné taky podotknout, že přidáním možnosti volby bounding boxu uživatelem může teoreticky dojít k tomu, že zvolený bounding box je menší než reálný bounding box. V takovém případě by se pak některé vrcholy ocitly mimo krajní souřadnice a jejich přepočítáním by vznikla čísla mimo povolený rozsah. Tedy v případě vzorkování 256ti úrovněmi čísla záporná nebo větší než 255. Na toto je ovšem autor upozorněn nejpozději při překladu.



Obrázek 7: bounding box (modře), krok (bíle), vrcholy do jediného bodu (červené šipky)

Ač se v této kapitole zmiňuji o jednotkové krychli, v praxi tato odpovídá většinou obecnému kvádru v původním souřadném systému. V takovém případě algoritmus funguje zcela stejně, jen je potřeba dávat pozor na možnost zkreslení komprimovaného modelu z důvodu rozdílné přesnosti v různých osách souřadného systému.

Tato technika je tedy velmi účiná z hlediska snížení datového objemu geometrických dat modelu. Jedná se však o ztrátovou kompresi, což může mít velký vliv na výslednou kvalitu aplikace. Problémům však lze předejít správnou volbou parametrů – jednotkových krychlí – při kompresi modelu a je tedy pro naše účely více než vhodnou metodou.

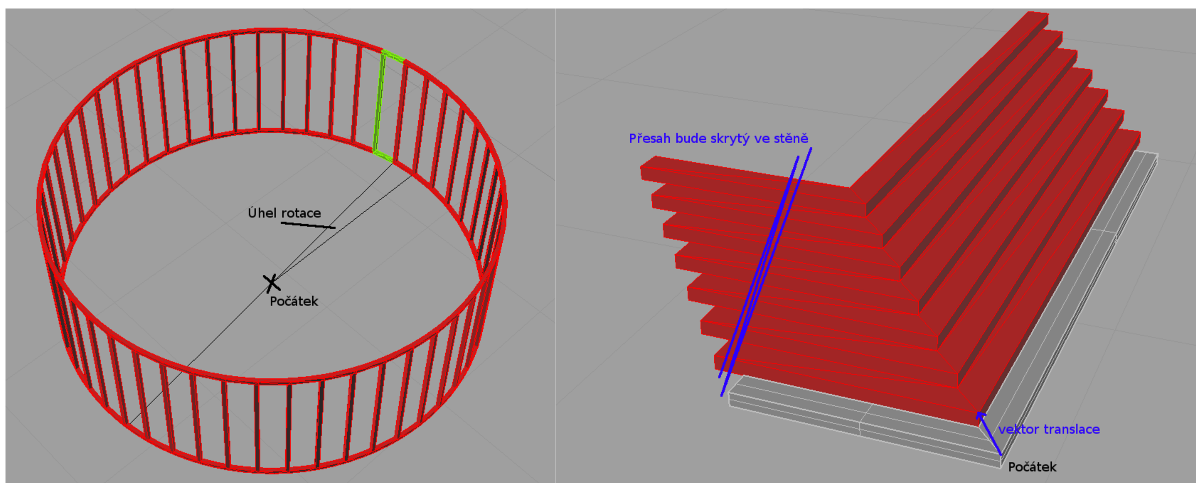
2.1.7 Opakování stejných částí

Z pohledu využití v moderní architektuře je vhodné také zmínit možnost instancování částí modelu, tedy jejich sdružování do celků, které pak mohou být opakovaně použity. Inspirací pro tuto techniku můžeme najít v RLE (*run length encoding*), které je běžně používané jak u kompresních programů, tak i v algoritmech používaných pro snížení velikosti obrazových dat. Pokud se v souboru vyskytuje velké množství stejných po sobě jdoucích shluků dat, například bílá plocha v obrázku, je do souboru místo „bílá, bílá, bílá, … bílá“ zapsáno „125x bílá“ (Můžeme-li to pro názornost takto zjednodušit). Lépe tedy každá sekvence vstupních hodnot je kódována dvojcí *(počet opakování, hodnota)*. Efektivita takové komprese je silně závislá na charakteru vstupních dat. Pro moderní architekturu, jak se ukáže později, je stupeň komprese modifikace této metody velmi dobrý.

Pro naše účely ovšem nebudeme v datech hledat stejné shluky na úrovni jednotlivých bytů, ale použijeme tuto metodu na vyšší úrovni abstrakce. Budeme tedy ve scéně hledat stejné části (nebo

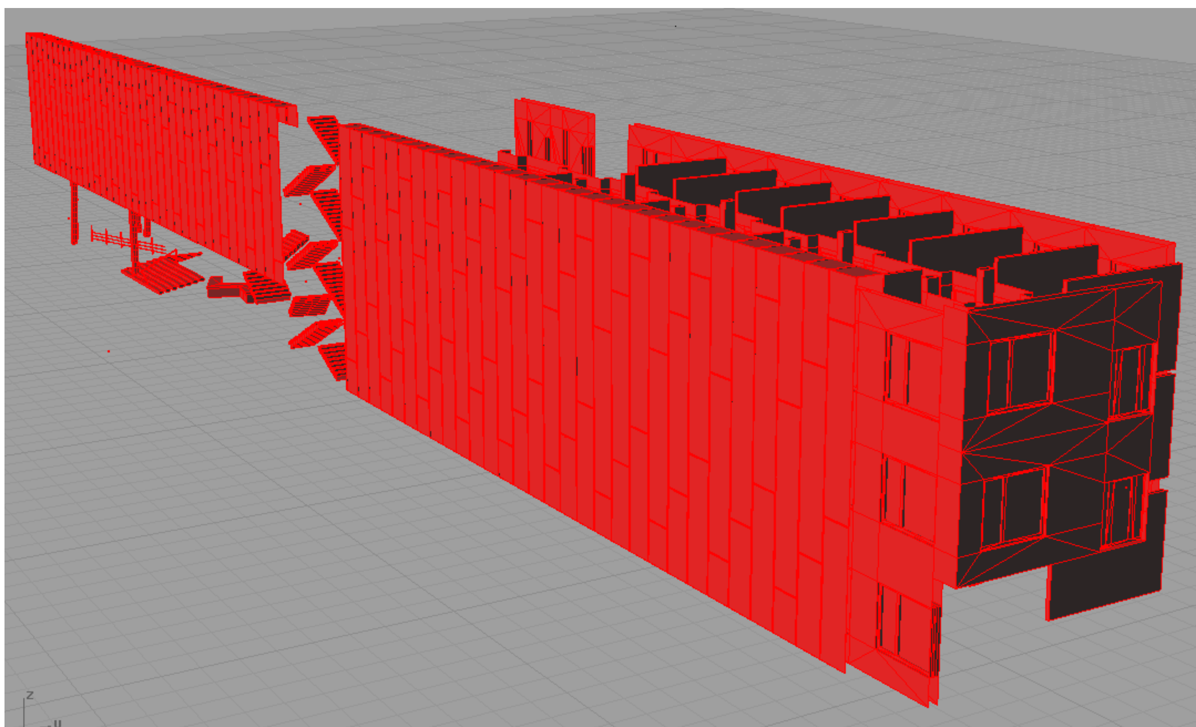
alespoň podobné, jejichž drobné rozdíly budeme v rámci LOD ignorovat), které můžeme detailně vymodelovat (například okno z několika částí – skleněná tabule, rám, žaluzie) a následně pomocí kumulativních rotací a posunů jej rozkopírovat do celé scény. Tuto techniku lze s výhodou využít při vytváření schodů, zábradlí, opěrných sloupů, apod... Veškeré pravidelně se opakující prvky tak můžeme buď vytvořit s vyšší úrovní detailu při zachování stejné velikosti výsledného modelu, což je velmi důležité pro vizuelní dojem ze scény, nebo naopak uspořit další data minimalistické reprezentace areálu.

Obrázek 8 ukazuje konkrétní použití této techniky pro vytvoření zábradlí a schodů. Na obrázku 9 je zase vidět, že se nám podařilo kopírováním vytvořit téměř celou budovu. Obrázek totiž zobrazuje pouze ty polygony, které se vytvoří až po spuštění programu kopírováním. Jejich předlohy jsou pro názornost skryté. I přesto je torzo budovy k poznání. Ještě více nám řeknou prostá čísla. Počet polygonů před rozgenerováním skupin (tedy počet uložených polygonů) je **6 297** a po něm **24 508**. Pomocí seskupování objektů do skupin a jejich následných kumulativní transformací získáme tedy úsporu dat přes **75 %**.



Obrázek 8: ukázka konkrétní praktické implementace, rozgenerované objekty jsou červeně

Na obrázku 8 vpravo jsou vidět vstupní schody do hlavní budovy, které jsou vytvořeny kumulativní translací spodního schodu. S každou další translací by se ovšem měl schod i zmenšovat. Vyřešení takovýchto konkrétních problémů by příliš zesložitilo kopírovací mechanismus a proto zde raději využíváme toho, že přesahující schody se skryjí ve stěně, u které jsou postaveny. Není to zcela čisté řešení, ale vizuelně to scénu nedegraduje. Jediné na co je třeba dávat pozor je návaznost horního schodu na podlahu. To se vyřeší mírným přetažením kraje podlahy přes plochu schodu. Tímto nám ale mohou vzniknout polygony z různých materiálů ležící přes sebe ve stejné rovině, což se projevuje při zobrazení problikáváním. Řešení je opět vnesení úmyslné malé nepřesnosti do modelu a posunout podlahu o milimetr nahoru (nebo lépe schod dolů, aby nevznikla nepřesnost i na dalších hranách podlahy). Tento drobný zásah není okem pozorovatelný a je tím pádem asi nejlepším kompromisem mezi vizuelní kvalitou a velikostí. K jiným problémům než výše zmíněným při praktivkém využití této metody nedochází.



Obrázek 9: z opakovaných částí je složen téměř celý model

2.2 Uložení světelných zdrojů

V komprimovaném modelu je nutné uložit i světelné zdroje umístěné ve scéně. Údajů pro světla však není mnoho – pozice, barva či směr – a tak bychom jejich kompresí mnoho nezískali. Světla tedy nekomprimujeme žádnou zvláštní metodou, pouze mohou být jako každá jiná část modelu sdružena do skupin, jak je popsáno v kapitole 2.1.7, a s nimi kopírována.

2.3 Komprese materiálů a textur

Materiály stejně jako světelné zdroje z předchozí kapitoly nám nenabízejí příliš velký manipulační prostor pro významnou kompresi. Je nutné dbát na šetrné uložení v paměti, tedy např. jednotlivé složky barvy ukládat jako celé číslo v rozsahu 0 až 255 a nikoli reálné číslo z rozsahu 0 až 1, jak jsme tomu zvyklí z OpenGL. Samozřejmostí je také uložení materiálů odděleně od geometrie modelu a odkazování se na ně adresou, protože bude jistě použit opakovaně v různých částech modelu.

Největší datovou velikost a tedy i nutnost komprese představují textury. Pojďme si tedy v následujících kapitolách popsat možnosti jejich generování – tzv. procedurální textury – a s tím související využití funkcí pro generování šumu.

2.3.1 Procedurální textury

Procedurální textura je tedy textura popsaná nikoli statickými hodnotami barvy každého pixelu obrázku, ale funkcí, která určuje hodnotu textury v každém bodě. Výhodou takových textur je možnost snadno měnit měřítko textury pouze upravením několika parametrů. My však využijeme přednosti procedurálních textur z hlediska množství uložených dat. V uložených datech bude tak zapsán pouze předpis pro vygenerování textury a textura samotná bude vytvořena až při spuštění programu dynamicky v operační paměti. Snadno tak lze generovat pravidelné vzory, jako např.

textura cihlové zdi, tyčkový plot, kachličky na podlaze či tašky na střeše. Procedurální textury založené na šumových funkcích zase snadno vygenerují materiály jako beton nebo omítka. Kombinace pravidelné textury s šumovou funkcí zajistí realističtější dojem výsledné textury nebo textury složitějších materiálů – mramor, dřevo, apod. [1]

2.3.2 Šum

V počítačové grafice existují nejrůznější metody pro generování šumu. Optimální šum, tzv. bílý šum, je vytvářen generátorem náhodných čísel. Textura vytvořená pomocí takové funkce pak obsahuje detaily na všech možných úrovních a může být tedy jakkoli zvětšována. Takový šum má ale pro praktické využití hned několik nevýhod. Jednak jeho výstup nelze parametrizovat, tedy pro stejné vstupní parametry bychom nedostali vždy stejné hodnoty, což je při zobrazování reality problém. Neméně podstatný je také vysoký početní výkon, který je třeba pro výpočet textury takovouto šumovou funkcí.

Pro potřeby počítačové grafiky tedy klademe na šumové funkce následující požadavky:

- statistickou invariantnost vzhledem k otáčení a posunu
- spojitost
- omezené frekvenční spektrum
- závislost na vstupu, parametrizovatelnost

Všechny výše zmíněné požadavky splňuje nejznámější z používaných šumů v počítačové grafice – Perlinův šum. Kromě výše zmíněného je taky velmi příznivá nízká náročnost výpočtu Perlinovy funkce. Základní popis Perlinovy šumové funkce naleznete v další kapitole nebo také v literatuře [1], podrobnější popis a implementace algoritmu od samotného autora potom v [15].

2.3.3 Perlinova šumová funkce

Perlinova šumová funkce generuje spojitý šum, který je vypočítáván v diskrétní mřížce. Lze ji definovat v libovolné dimenzi. Pro účely texturování objektů je pro nás vhodná 2D implementace, jejíž postup tedy popíši i dále v textu. Pro trojrozměrnou definici může čtenář nahlédnout do literatury [1], ze které vychází i tento text.

Základem bude šumová funkce *noise(float x, float y)*, která pro každé dvě hodnoty x a y bude vracet vždy stejné číslo z intervalu `<-1,1>`. Tato funkce bude splňovat podmínku parametrizovatelnosti, protože pro stejný vstup vrátí vždy stejné výsledky. To nám zajistí, že zobrazovaná textura bude sice náhodná, ale v čase konstantní. Při každém spuštění programu ovšem konstantní nebude, protože jak uvidíme níže, při inicializaci programu vytvoříme pseudonáhodné hodnoty, na kterých bude výsledek funkce záviset a tyto budou zřejmě při každém spuštění jiné. To ovšem není pro účely tohoto projektu na škodu, rozhodně méně než kdybychom zbytečně zvětšili program o tyto předem nadefinované hodnoty.

Podstatou Perlinovy funkce je rozdělení prostoru do diskrétní mřížky. Pro každý vrchol této mřížky definuje tzv. vlnkovou funkci (*wavelet*), která je pseudonáhodná, v našem případě dvourozměrná a má určitý poloměr určující její rozsah. Tato funkce je nulová ve vrcholu, ke kterému přísluší. Není tedy třeba ji definovat úplně, postačí pouze gradient. Výpočet Perlinovy šumové funkce potom spočívá ve třech krocích.

V prvním kroku určíme buňku, do které bod [x,y] náleží a to tak, že obě jeho souřadnice zaokrouhlíme dolů na celé čísla, čímž získáme „levý dolní“ vrchol. Označíme jej souřadnicemi [i,j]. Ostatní 3 vrcholy určíme postupným přičtením jedničky, tedy

```
[i+1,j],[i,j+1],[i+1,j+1]
```

K výpočtu tvaru vlnky v druhém kroku doporučuje Perlin využít jednak tabulku 256ti pseudonáhodných vektorů **G**, které rovnoměrně vzorkují jednotkovou kružnici a potom permutované

pole 256 indexů, pomocí kterého budeme k předchozímu přistupovat (označíme **P**). Pole **G** vygenerujeme podle algoritmu z [1]:

```
Proveď pro 256 vzorků i:
   1. Vygeneruj 2 pseudonáhodná čísla -1 < x,y < 1
   2. Pokud jsou vně jednotkového kruhu, zamítni je a opakuj předchozí krok
   3. Normalizuj vektor daný souřadnicemi (x,y)
   4. Ulož vektor (x,y) do pole G[i]
```

Pro vytvoření permutovaného pole indexů využijeme také algoritmus z literatury [1]:

```
Zaplň pole P hodnotami od nuly do 255
Pro i od nuly do 255:
   1. vygeneruj pseudonáhodné číslo 0 ≤ k < 256
   2. zaměň P[i] a P[k]
```

V této fázi máme jednorozměrné pole gradientů vlnky **G** a dvourozměrné hodnoty [i,j], pro které tyto náhodné směry – gradienty – hledáme. Z literatury opět víme, že k indexaci se použije tzv. přeložení souřadnic – v anglickém text *fold*. Pro souřadnice [i,j] získáme hodnotu gradientu podle následujícího vztahu:

```
fold(i, j) = P[(P[i mod 256] + j) mod 256]
```

Gradient získáme potom voláním **G**[*fold(i, j)*]. Abychom získali hodnotu vlnky v bodě [x, y], spočteme nejdříve relativní vzdálenost bodu od vrcholu, aplikujeme na obě složky relativního úbytku kubickou funkci *drop(t)* a znásobením výsledků pro obě složky získáme celkový relativní úbytek **Ω**. Vynásobením s náhodnou funkcí G získáme výsledek pro jeden vrchol.

```
drop(t) = 1 – 3 |t|2 + |t|3
[u, v]  = [x, y] – [i, j], -1 ≤ u, v < 1
Ω(u, v) = drop(u) * drop(v)
Ω(u,v) * G(i, j)              (= výsledek pro vrchol [i,j])
```

Nakonec součtem výsledků pro všechny vrcholy získáme hodnotu šumu v bodě [x,y]. Jak vidíme, celý výpočet je velmi jednoduchý. Pseudonáhodná pole si vygenerujeme při inicializaci programu a výpočet konkrétní hodnoty je proveden podle triviálních vzorců uvedených výše. Algoritmus přímo od autora pro implementaci je uveden v příloze knihy [15].

Výše jsem uvedl postup implementace pro 2D texturu. Obvykle se v literatuře uvádí trojrozměrná varianta, to proto, že je univerzálnější. Pokud pomocí trojrozměrné varianty potřebujeme vygenerovat dvourozměrnou texturu, stačí místo jedné souřadnice dosadit konstantní hodnotu, čímž získáme rovinný řez trojrozměrnou texturou. Pokud ovšem dopředu víme, že 3. rozměr v programu nikdy nevyužijeme a záleží nám na celkové velikosti výsledného programu, můžeme jej vynechat zcela a ušetřit několik instrukcí programu.

2.3.4 Skládání šumových funkcí

Perlinova šumová funkce nám generuje šum. To je sice pěkné, ale samo o sobě ne až tak užitečné. Praktické využití nám přináší teprve možnost skládání šumových funkcí. Nyní je třeba zmínit, že funkce *noise(2x, 2y)* generuje šum s dvojnásobnou frekvencí. Skládání funkcí potom spočívá v prostém součtu Perlinových funkcí s různou amplitudou a frekvencí. Eventuelně je pro generování složitějších struktur pomocí šumu modifikována nějaká pravidelné funkce. Například posunutím funkce sinus do kladných hodnot, její „normalizací“ do rozsahu 0 až 1 a následným zašuměním získáme texturu mramoru.

3 Vytvoření modelu

Z teoretické roviny se nyní přesuneme k čistě praktické části projektu. V dalších kapitolách si popíšeme samotné vytvoření modelu, které bude ilustrované na konkrétním případu areálu Božetěchova – resp. vzhledem k vyšší časové náročnosti vytvoření modelu oproti předpokladům a tedy nedostatku času pouze na případu hlavní budovy školního areálu. Výstupem této části práce potom bude statická knihovna s uloženou scénou, kterou posléze využijeme v další části práce – zobrazovacím programu.

3.1 Použité nástroje

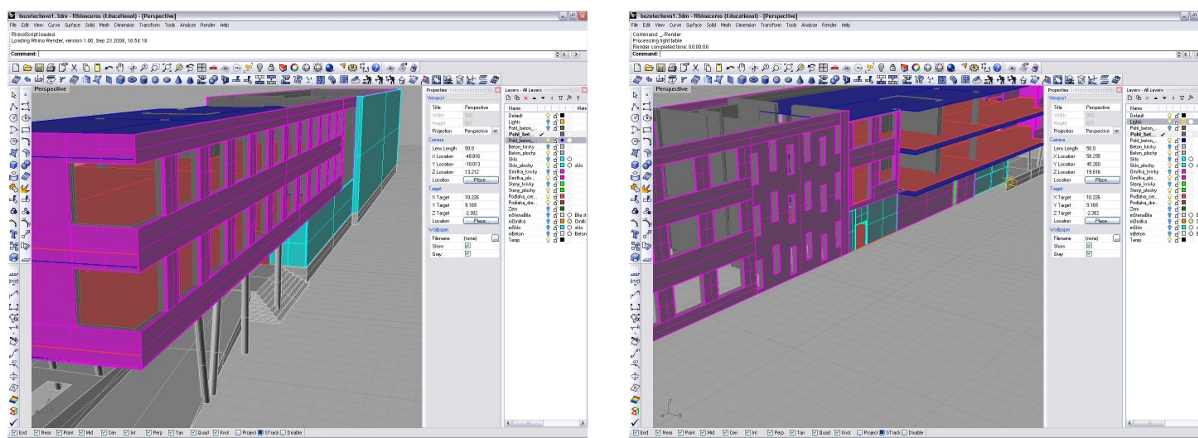
Při návrhu aplikace byl, kromě požadavků na minimalističnost programu a kvalitu modelu, brán zřetel také na možnosti budoucího vývoje. Cílem tedy nebylo pouze vytvořit konkrétní scénu s konkrétním modelem, byť to byla hlavní motivace tohoto projektu, ale také vytvoření určitého pracovního postupu, který by v budoucnu mohl být rozšířen, snadno upraven pro jinou konkrétní aplikaci či zobecněn pro snadnější vytváření minimalistických aplikací začátečníky. S přihlédnutím k možnosti práce na projektu celého týmu (ne-programátory nevyjímaje) byl celý proces rozdělen do následujících menších samostatných částí:

1. Vytvoření scény v programu Rhinoceros 3D. Výstupem této části je model ve formátu *.3dm, což je standardní soubor pro uložení dat z tohoto programu. V této fázi vytvoříme polygonální model, ručně musíme určit LOD. Jednotlivé části modelu pojmenujeme po skupinách, po kterých jej budeme následně exportovat. Rozdělíme polygony také do vrstev, kterým nadefinujeme a pojmenujeme materiály. Stejně tak vytvoříme ve scéně světelné zdroje a trajektorii pro pohyb kamery. Připravíme si skupiny, které budou v komprimovaném modelu instancovány a rozkopírovány.
2. Pro export dat z 3D programu použijeme skript napsaný v jazyce Rhinoscript. Autor má předpřipravené 3 soubory se skripty, jejichž načtením (jednorázově pomocí funkce *loadScript*) se v programu nadefinují funkce *minExport*, *grpExport* a *camExport*. Spuštěním funkce *minExport* je programátor vyzván k výběru objektů typu *mesh* a *light*. Pro objekty typu *mesh* se ručně zadává bounding box. Je možné zadání přeskočit, čímž se použijí nejbližší možné body bounding boxu. Ostatní se exportuje automaticky, včetně materiálů a textur patřících k těmto objektům. Při volání funkce *grpExport* je programátor vyzván k výběru skupin a ke každé musí ručně nadefinovat počátek, vektor translace, úhel rotace ve všech osách a počet opakování těchto transformací (tedy počet kopií). U *camExport* vybere pouze *pointCloud* určující body trasy kamery. Výstupem této části jsou textové soubory s daty scény bez komprimačních modifikací ve formátu jak je uvedeno v kapitole 3.1.2.
3. V této fázi je nutné hrubá exportovaná data v textových souborech převést na zdrojové soubory v jazyce C. K tomu máme připravený skript v PERLu *conv.pl.* Tato fáze je zcela automatická, programátor pouze skript spustí. V této fázi proběhne i přepočet souřadnic do celých čísel, tedy hlavní část komprese, jak je uvedeno v kapitole 2.1.6.
4. Následuje samotná kompilace zdrojových souborů, k čemuž je v předchozí fázi vygenerován *makefile*. Výstupem této fáze je knihovna *scene.o* obsahující veškerá data modelu, které se společně se zobrazovacím programem linkuje do spustitelného souboru *main.exe*.
5. Se spustitelným souborem zbývá ořezání zbytečných symbolů programem strip.exe a následná komprese samotného spustitelného souboru pomocí specializovaných programů jak je popsáno v kapitole 4.5. Pro náš program dosáhl nejlepších výsledků kompresor *Kkrunchy*.

Jednotlivé části jsou nezávislé na ostatních, což umožňuje samostatný vývoj každé z nich. Pokud např. nebude u rozsáhlejších modelů výkonnostně dostačovat export dat z 3D prostředí pomocí interpretu skriptovacího jazyka Rhinoscript, lze tento krok nahradit vytvořením kompilovaného plug-inu pro Rhinoceros, případně vytvořit exportní skripty i pro jiná 3D modelovací prostředí. To vše bez nutnosti zásahu do ostatních částí programu. V dalších kapitolách se potom budu věnovat jednotlivým krokům vytvoření modelu až po vytvoření statické knihovny s modelem v jazyce C. Poslední části – vytvoření zobrazovacího programu – je věnována samostatná 4. kapitola.

3.1.1 Rhinoceros

Pro vytvoření modelu areálu byl pro účely této práce zvolen modelovací program Rhinoceros 3D ve verzi 4. Jedná se o propracovaný komerční NURBS modelář pro operační systém Windows od americké firmy Robert McNeel & Associates s širokými možnostmi importu a exportu dat. Tento program nachází uplatnění především v architektuře, strojírenství či jakémkoli jiném oboru využívajícím CAD aplikace. Styl práce s tímto programem je velmi podobný práci ve známějším programu AutoCAD od firmy AutoDesk, naopak uživatelům zvyklým na modelovací nástroje typu 3ds Max nebo Maya nemusí práce s tímto nástrojem příliš vyhovovat. Tedy oproti zmíněným modelovacím nástrojům, ve kterých je důraz kladen na intuitivnost při vytváření nepravidelných objektů a výslednou vizuelní podobu, je program Rhinoceros (v dalším textu jej budu zkráceně nazývat Rhino) spíše „rýsovací nástroj“, kde výsledný objekt je vytvářen skládáním matematicky přesných křivek a ploch a kde je velmi důležitá přesnost výsledku. Oproti AutoCADu je více orientovaný na vytváření 3D modelů.



Obrázek 10: ukázka z programu Rhinoceros 3D 4.0

Tento program jsem vybral pro vytvoření modelu jednak pro osobní sympatie k tomuto typu modelování, jednak také pro příznivou licenční politiku pro studenty a školy – s čímž také souvisí jeho dostupnost ve fakultních laboratořích. Neméně důležitým faktorem je také široká komunita uživatelů ochotných se podělit o své zkušenosti – především tedy z řad studentů. Podmínkou nutnou pak byla schopnost zpracovat dodané materiály pro tvorbu modelu areálu – *.dwg soubory vytvořené v konkurenčním AutoCADu – a snadná rozšiřitelnost. Z hlediska rozšiřitelnosti má Rhino široké možnosti od tvorby jednoduchých maker, přes tvorbu skriptů v jazyce Rhinoscript – což je rozšíření VBScriptu, které si popíšeme v následující kapitole – až po možnost tvorby vlastních výkonných plug-inů v jazyce C/C++ pomocí zdarma dodávaného SDK.

Nevýhodou volby tohoto programu, jak se při tvorbě modelu ukázalo, je celkem špatná podpora práce s polygonální reprezentací modelu. Rhinoceros sice nabízí několik základních funkcí pro vytváření polygonálního modelu, ale nakonec se ukázalo, že efektivnější (nikoli bohužel efektivní) metodou je vytvoření modelu pomocí NURBS ploch a jeho následná konverze do polygonální reprezentace.

Nejvíce informací z uživatelského pohledu získá čtenář na českých stránkách věnovaných tomuto programu [10], kde lze nalézt i odkazy na specializované zdroje k různým tématům. Pro vývojáře je dostupný dostatek informací na samotných stránkách výrobce [11].

3.1.2 Export dat

Pro export dat bylo na výběr hned několik možností. Jednou z alternativ bylo využití standardních datových formátů pro uložení 3D dat, např. rozšířený a otevřený (textový) formát VRML. Výhodou tohoto přístupu by byla možnost vytvářet modely v různých modelovacích prostředích, která zpravidla podporují export do tohoto obecně známého datového formátu. Za nevýhodu můžeme označit zbytečnou složitost VRML formátu oproti zvolenému vlastnímu formátu, ale především nemožnost pohodlné interakce s uživatelem v již hotovém 3D prostředí.

Především proto byl pro export dat ze Rhina zvolen skriptovací jazyk Rhinoscript. Je to ve své podstatě jazyk VBScript od společnosti Microsoft, který je rozšířen o API pro práci se scénou a funkcemi Rhina. Místo popisu nezajímavé syntaxe tohoto jazyka odkáži čtenáře na příslušnou literaturu [2], stejně tak mi byla dobrým pomocníkem reference funkcí Rhinoscriptu [3].

Z vlastností jazyka zde zmíním pouze závislost funkcí pro tisk reálných čísel na lokálním nastavení Windows – desetinný oddělovač jako „tečka“ nebo „čárka“ – a nemožnost toto ovlivnit při běhu skriptu, což mi přijde velmi nešťastné. Spokojený nemohu být ani s rychlostí interpretu tohoto jazyka, která je nesrovnatelně horší než např. rozšířený PERL. Také díky tomu jsem Rhinoscript využil pouze k „obyčejnému“ exportu vrcholů a dalších dat do textového souboru. Jejich další zpracování – přepočítání do celých čísel, apod. – jsem potom nechal na známějším, výkonnějším a především na funkce bohatším jazyku PERL.

Výstupem této fáze je tedy několik textových souborů – pro každý objekt, světlo, materiál či texturu samostatný soubor. Stejně tak lze exportovat i skupiny, pro které je vytvořen také samostaný soubor s odkazy na seskopené objekty. Každý soubor má na prvním řádku datum a čas exportu, druhý řádek obsahuje potom řetězec s typem objektu – možné hodnoty jsou LIGHT, MATERIAL, MESH, TEXTURE a pro skupinu předchozích objektů také GROUP. Další řádky souboru se liší dle typu objektu. Nejjednodušší je soubor s texturou, který obsahuje cestu k obrázku na disku. Soubor se světlem obsahuje další dva řádky. Na prvním je hodnota barvy světla a na dalším jeho pozice. Složitější je soubor s materiálem, který postupně obsahuje řádky s barvou materiálu, barvou odlesků, hodnotou odrazivosti (lesklosti) a hodnotou průhlednosti materiálu. Na posledním řádku je potom název textury, pokud je nastavená. Soubor s geometrií modelu potom na dvou řádcích obsahuje krajní souřadnice bounding boxu, na dalším název materiálu nebo prázdný řádek (pokud není nastaven) a potom geometrická data modelu, tedy souřadnice jednotlivých vrcholů – co řádek to vrchol – včetně texturovacích souřadnic. Pro otexturovaný objekt je tedy na každém řádku 5 čísel, pro ostatní pouze 3 souřadnice kartézského souřadného systému. Následuje oddělovač v podobě dvou rovnítek na samostatném řádku, který říká, že dále v souboru jsou na jednotlivých řádcích uloženy topologická data. Každý řádek obsahuje 3 nebo 4 celočíselné indexy do pole vrcholů určující jednotlivé trojúhelníky modelu (resp. čtyřúhelníky). Posledním typem souboru je skupina – GROUP. Tento soubor obsahuje na řádcích popořadě za identifikací řetězcem GROUP: počet seskupených skupin, objektů (geometrických dat) a světel, zvolený počáteční bod skupiny, vzhledem k němuž budou aplikovány rotace, následně vektor translace skupiny, potom na jednom řádku úhly rotace kolem jednotlivých os (ve stupních) a nakonec již pouze celé číslo odpovídající počtu opakování transformací se skupinou a seznam jmen jednotlivých seskupených částí.

Níže definice exportovaných souborů pomocí gramatiky v BNF:

```
<file>   ::= <c> <group> | <c> <materi> | <c> <mesh> | <c> <textur> | <c> <light>
<c>      ::= '// Export date and time ' <actual-date> <EOL>
<group>  ::= 'GROUP' <grp-count> <grp-geom> <grp-repeat> <obj-list>
<light>  ::= 'LIGHT' <EOL> <color> <EOL> <point> <EOL>
<mesh>   ::= 'MESH' <EOL> <bounding-box> <material-name> <mesh-data>
<materi> ::= 'MATERIAL' <EOL> <color> <EOL> <color> <EOL> <INT> <EOL> <INT> <EOL>
             <texture-name>
```

```
<textur> ::= 'TEXTURE' <texture-file-path>
<color>  ::= <INT> ' ' <INT> ' ' <INT>
<point>  ::= <FLOAT> ',' <FLOAT> ',' <FLOAT>
<bounding-box>  ::= <point> <EOL> <point> <EOL>
<mesh-data> ::= <tex-vertices> '==' <EOL> <indexes> |
                <vertices>     '==' <EOL> <indexes>
<vertices>  ::= <vertex> <vertices> | <vertex>
<vertex> ::= <point> <EOL>
<tex-vertices>  ::= <tex-vertex> <tex-vertices> | <tex-vertex>
<tex-vertex> ::= <point> ',' <FLOAT> ',' <FLOAT> <EOL>
<indexes>    ::= <polygon> <indexes> | <polygon>
<polygon>    ::= <INT> ',' <INT> ',' <INT> ',' <INT> <EOL> |
                 <INT> ',' <INT> ',' <INT> <EOL>
<texture-name>  ::= <STRING> <EOL>
<material-name> ::= <STRING> <EOL>
<texture-file-path> ::= <STRING> <EOL>
<grp-count>  ::= <INT> <EOL> <INT> <EOL> <INT> <EOL>
<grp-geom>   ::= <point> <EOL> <point> <EOL> <point> <EOL>
<grp-repeat> ::= <INT> <EOL>
<obj-list>   ::= <STRING> <EOL> | <STRING> <EOL> <obj-list>
```

V poslední verzi programu také přibyl skript pro export souřadnic pro pohyb kamery. Ten se oproti předchozích popsaných chová trochu odlišně. Jednak kameru máme ve scéně jen jednu, takže vznikne právě jeden soubor s pevným názvem *camera.dat*. Tento soubor je textový jako předchozí uvedené a je uložen ve stejné složce. Rozdíl je v koncovce, protože návrh s tímto souborem nepočítal a koncovka souborů byla použita k určení zda se jedná o exportovaný soubor s modelem, materiálem, atd... K datům kamery však potřebujeme přistupovat jinak, proto jiná koncovka. Obsahem souboru jsou potom jednotlivé body kudy bude kamera prolétávat. Co řádek to bod. Jednoduchou strukturu popisuje nádledující gramatika v BNF:

```
<file>   ::= <points>
<points> ::= <points> <point> | <point>
<point>  ::= <FLOAT> ',' <FLOAT> ',' <FLOAT> <EOL>
```

(Non-terminál <file> je zřejmě zbytečný. Byl použit pouze proto, aby bylo na první pohled zřejmé, že se jedná opravdu celý soubor, nikoli jen o část.)

3.1.3 PERLový konvertor

Představovat jeden z nejrozšířenějších a nejvýkonnějších interpretovacích jazyků by bylo nošením dříví do lesa. Tento jazyk je populární především pro svou schopnost snadné a rychlé práce s textovými soubory a tak byl logickou volbou i v této fázi vytváření modelu, kdy potřebujeme z exportovaných dat vytvořit zdrojový kód jazyka C.

Vstupem v této fázi jsou exportované textové soubory ve formátu uvedeném v předchozí kapitole, výstupem potom stejný počet souborů se stejným názvem, ale odlišnou koncovkou (*.c) a samozřejmě obsahem. U souborů se světly a materiálem převedeme data pouze do podoby kódu v jazyce C se statickou strukturou, u modelů (typ *mesh*) to samé plus převedení souřadnic do celočíselné reprezentace tak, jak je popsáno v kapitole 2.1.6.

Konverze textury potom vytvoří soubor s funkcí stejného jména jako je jméno souboru. Načte data souboru s texturou a staticky (bez komprese) je zapíše do zdrojového kódu funkce. Programátor může následně takový soubor vzít a funkci předefinovat tak, aby vrácená textura byla vygenerována až po spuštění programu – v momentě volání funkce – a tedy tímto způsobem ušetřeno místo. Více o této problematice najdete v kapitole 2.3.1. Funkce je nadefinována podle této šablony:

```
unsigned char * nazev_textury(unsigned int *w, unsigned int *h);
```

3.1.4 Knihovna v jazyce C

Poslední fází vytvoření minimalistického modelu je zkompilování vygenerovaných zdrojových souborů, společně s předpřipravenými funkcemi pro práci s ním, do statické knihovny. Získáme tak binární reprezentaci scény včetně API pro získání dekomprimovaných dat. Níže přikládám ukázku dekomprimovaných struktur a deklaraci funkcí pro jejich získání.

```
/**
 * struktury, které jsou vráceny po dekompresi scény
 */
typedef struct { float x; float y; float z; } TVertex;
typedef struct { float x; float y; } TTextCoord;
typedef struct {
  int transparent;         // priznak, zda je pruhledny
  float diffuse[4];        // barva materiálu - diffuse
  float ambient[4];        // barva materiálu - ambient
  float specular[4];       // barva materiálu - specular
  float shininess;         // hodnota odlesku
  int textured;            // příznak, zda je texturovaný
  unsigned int tex_w;      // šířka textury
  unsigned int tex_h;      // výška textury
  unsigned char * texture; // texturovací data
  unsigned texid;          // texid (OpenGL), je prázdné
  unsigned int c;          // pocet polygonu
  TVertex * triangles;     // vrcholy trojúhelníků; počet = c
  TTextCoord * textcoords; // texturovací souřadnice; počet = c
  TVertex * normals;       // normály; počet = c/3
  float midpoint[3];       // těžiště objektu
} Object;


typedef struct {
  float position[4];       // pozice světla
  unsigned char col[3];    // barva světla
} Light;

/**
 * funkce pro získání rozgenerovaných struktur scény (pro dekompresi)
 * @param int c - počet vrácených objektů/světel
 */
Light ** getLights(int * c);
Object ** getObjects(int * c);
```

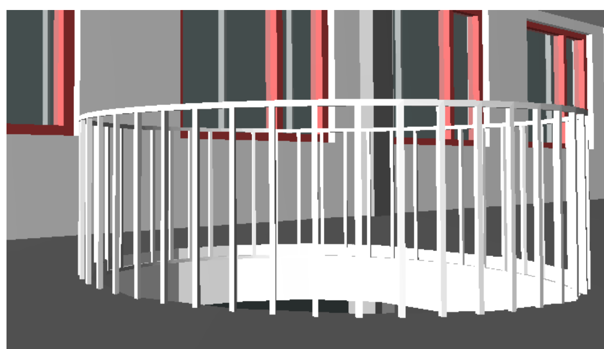
Další užitečnou složkou knihovny je také API pro práci s kamerou. K dispozici jsou základní funkce pro inicializaci kamery, funkce realizující pohyb kamery (timetick) a dvě funkce pro získání pozice a směru kamery. Jako bonus jsou tu dvě funkce pro ovládání rychlosti pohybu kamery. Funkci pro inicializaci kamery lze volat opakovaně, uvedeme jí tím do původního stavu.

```
void camera_init();
void camera_timetick(unsigned int fps);
float * camera_position();
float * camera_direction();
void camera_speedup();
void camera_speeddown();
```

3.2 Ukázky komprimovaných struktur

Výsledky celkové komprese na papíře zachytit bohužel nelze, ale mohu zde předvést několik příkladů struktur obsahujících částečně zkomprimovaná data. Jedná se o výstup PERLového konvertoru jak je popsán v kapitole 3.1.3. První ukázkou jsou 2 soubory představující kruhové zábradlí kolem otvoru ve stropě nad vrátnicí. Jeden je modelem výseče tohoto zábradlí a druhý potom skupina, která pomocí transformací dotvoří zábradlí jako celek. Zdrojový soubor je pro lepší čitelnost naformátován (přidány konce řádků).

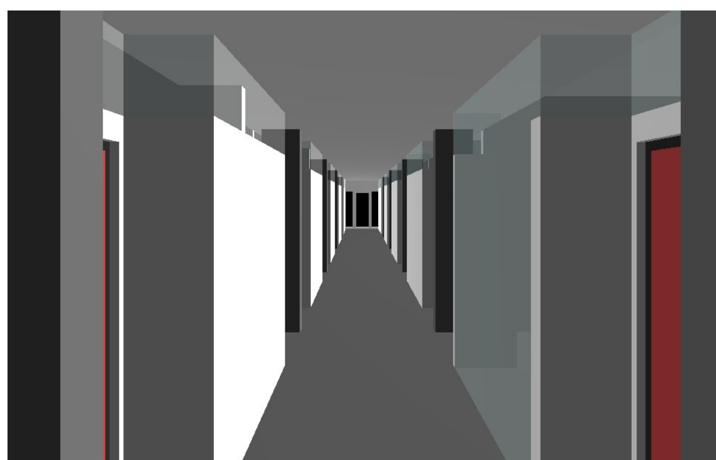
```
#ifndef grp_zabradli_kruhove_c
#define grp_zabradli_kruhove_c

#include "../scene.h"
#include "mod_zabradli_kruhove.c"

static Group grp_zabradli_kruhove =
       {  4,
         {4.55190, 3.10427,  3.80000},
         {0.00000, -0.00000, 0.00000},
         {0.00000, 0.00000,  7.20000},
         49,
         {&mod_zabradli_kruhove}
       };

#endif

#ifndef mod_zabradli_kruhove_c
#define mod_zabradli_kruhove_c

#include "../scene.h"
#include "mat_hlinik.c"

static Model mod_zabradli_kruhove =
       { {4.74194, 1.66031, 3.80000},
         {4.92470, 1.72010, 4.82500},
         &mat_hlinik,
         72, 16, 84
          {255, 151, 0, 130, 63, 0, 4, 0, 0, 0, 105, 0, 246, 255, 0, 221, 125, 6, 4,
0, 6, 255, 151, 6, 212, 229, 6, 246, 255, 6, 0, 105, 6, 107, 158, 6, 255, 151, 255,
246, 255, 255, 0, 105, 255, 4, 0, 255, 124, 168, 255, 4, 0, 248, 255, 151, 248,
221, 125, 248, 0, 105, 248, 212, 229, 248, 107, 158, 248, 246, 255, 248, 5, 19, 21,
8, 23, 18, 7, 9, 8, 21, 23, 9, 18, 19, 5, 7, 0, 1, 4, 1, 2, 3, 3, 4, 1, 5, 6, 1, 6,
2, 1, 0, 7, 5, 1, 0, 5, 8, 9, 4, 3, 10, 11, 4, 3, 11, 11, 8, 4, 10, 6, 11, 6, 5,
11, 5, 8, 11, 12, 13, 16, 14, 15, 16, 15, 12, 16, 12, 15, 19, 18, 12, 19, 15, 17,
19, 14, 16, 22, 16, 13, 21, 13, 23, 21, 20, 14, 22, 21, 22, 16, 19, 17, 22, 20, 22,
17, 22, 21, 19}};

#endif
```



Obrázek 11: výsledek v aplikaci

Jako druhou ukázku jsem zvolil část členité stěny chodby opět na obrázku dole.

```
#ifndef mod_stenachodba_4_c
#define mod_stenachodba_4_c

#include "../scene.h"
#include "mat_bila_stena.c"

static Model mod_stenachodba_4 =
       { {25.82500, 7.10000, -0.00000},
         {43.90000, 7.75000, 2.25000},
         &mat_bila_stena,
         375, 240, 264,
         {2, 58, 0, 2, 58, 255, 2, 255, 0, 2, 255, 255, 0, 0, 0, 0, 0, 255, 25, 0, 0, 25, 0, 255, 0,
255, 255, 255, 197, 255, 62, 196, 255, 76, 196, 255, 77, 196, 255, 25, 196, 255, 27, 196, 255, 41, 196,
255, 77, 0, 255, 23, 58, 255, 23, 255, 255, 226, 58, 255, 181, 58, 255, 179, 196, 255, 179, 0, 255, 226,
255, 255, 181, 255, 255, 127, 0, 255, 127, 196, 255, 125, 255, 255, 125, 58, 255, 80, 58, 255, 80, 255,
255, 228, 0, 255, 228, 196, 255, 230, 196, 255, 244, 196, 255, 255, 255, 255, 244, 255, 255, 76, 255, 255,
62, 255, 255, 230, 255, 255, 41, 255, 255, 27, 255, 255, 0, 196, 255, 80, 0, 255, 125, 0, 255, 178, 255,
255, 128, 255, 255, 142, 196, 255, 164, 196, 255, 164, 255, 255, 142, 255, 255, 128, 196, 255, 178, 196,
255, 0, 255, 0, 23, 58, 0, 76, 255, 238, 62, 255, 238, 62, 196, 0, 62, 196, 238, 62, 255, 0, 62, 235, 238,
62, 215, 238, 76, 196, 0, 76, 255, 0, 76, 196, 238, 77, 196, 0, 27, 255, 238, 41, 255, 238, 41, 196, 0, 41,
255, 0, 41, 196, 238, 41, 235, 238, 41, 215, 238, 27, 196, 0, 27, 196, 238, 27, 255, 0, 25, 196, 0, 23,
255, 0, 77, 0, 0, 80, 58, 0, 80, 255, 0, 127, 0, 0, 127, 196, 0, 181, 255, 0, 178, 255, 0, 178, 255, 238,
181, 58, 0, 179, 0, 0, 179, 196, 0, 125, 58, 0, 125, 255, 0, 128, 255, 0, 128, 255, 238, 164, 196, 0, 142,
196, 0, 164, 196, 238, 142, 196, 238, 142, 255, 0, 164, 255, 0, 164, 255, 238, 142, 255, 238, 128, 196, 0,
128, 196, 238, 142, 235, 238, 142, 215, 238, 178, 196, 0, 178, 196, 238, 164, 235, 238, 164, 215, 238, 228,
0, 0, 228, 196, 0, 226, 58, 0, 226, 255, 0, 230, 255, 0, 230, 255, 238, 255, 196, 0, 244, 196, 0, 244, 196,
238, 244, 255, 0, 244, 255, 238, 255, 255, 0, 230, 196, 0, 230, 196, 238, 244, 235, 238, 244, 215, 238, 1,
0, 2, 3, 5, 4, 6, 7, 36, 34, 9, 35, 37, 38, 10, 11, 15, 40, 41, 14, 38, 40, 15, 10, 43, 44, 28, 29, 34, 36,
39, 33, 49, 50, 47, 48, 47, 50, 46, 51, 45, 49, 48, 52, 42, 53, 4, 5, 1, 17, 54, 0, 37, 55, 56, 38, 57, 59,
60, 61, 63, 62, 64, 55, 55, 64, 61, 60, 10, 58, 64, 11, 40, 67, 66, 41, 71, 69, 68, 72, 66, 74, 73, 75, 72,
74, 66, 71, 14, 74, 70, 15, 38, 56, 67, 40, 69, 67, 56, 59, 15, 70, 58, 10, 70, 68, 57, 58, 18, 77, 54, 17,
12, 65, 78, 16, 29, 79, 80, 30, 76, 13, 7, 6, 25, 81, 82, 26, 78, 81, 44, 43, 20, 86, 83, 24, 21, 88, 87,
22, 27, 90, 89, 28, 28, 89, 79, 29, 47, 96, 95, 48, 96, 94, 93, 95, 99, 100, 50, 49, 98, 97, 100, 99, 51,
102, 96, 47, 104, 102, 92, 103, 92, 102, 101, 91, 103, 97, 94, 104, 50, 100, 92, 46, 106, 52, 48, 95, 85,
106, 108, 107, 84, 105, 106, 85, 93, 98, 107, 108, 99, 49, 45, 85, 31, 109, 110, 32, 22, 87, 109, 31, 23,
112, 111, 19, 19, 111, 86, 20, 33, 122, 117, 34, 124, 122, 114, 123, 114, 122, 121, 113, 123, 118, 116,
124, 36, 119, 114, 39, 1, 3, 42, 5, 7, 1, 11, 12, 37, 13, 14, 41, 12, 16, 29, 17, 1, 7, 18, 17, 13, 19, 20,
22, 21, 22, 20, 23, 19, 32, 20, 24, 21, 25, 26, 28, 7, 13, 17, 27, 28, 26, 29, 30, 12, 22, 31, 19, 32, 33,
39, 31, 32, 19, 39, 23, 32, 41, 18, 13, 30, 37, 12, 42, 5, 1, 8, 42, 3, 16, 43, 29, 44, 25, 28, 24, 45, 21,
46, 27, 26, 26, 51, 46, 52, 21, 45, 42, 8, 53, 58, 57, 61, 59, 56, 60, 56, 55, 60, 58, 61, 64, 12, 11, 64,
64, 62, 65, 65, 12, 64, 68, 70, 72, 67, 69, 71, 66, 67, 71, 72, 70, 74, 14, 13, 74, 73, 74, 76, 13, 76, 74,
18, 41, 66, 66, 75, 77, 77, 18, 66, 37, 30, 55, 63, 55, 80, 30, 80, 55, 43, 16, 78, 25, 44, 81, 85, 45, 24,
84, 85, 83, 24, 83, 85, 90, 27, 92, 46, 92, 27, 92, 91, 90, 26, 82, 102, 51, 26, 102, 101, 102, 82, 96,
102, 104, 92, 100, 103, 94, 96, 104, 100, 97, 103, 88, 21, 106, 21, 52, 106, 106, 105, 88, 106, 95, 108,
99, 85, 107, 95, 93, 108, 98, 99, 107, 23, 39, 114, 112, 23, 114, 114, 113, 112, 9, 34, 117, 117, 116, 115,
115, 9, 117, 36, 35, 119, 118, 119, 120, 35, 120, 119, 33, 32, 122, 32, 110, 122, 121, 122, 110, 117, 122,
124, 114, 119, 123, 116, 117, 124, 119, 118, 123}
   };

#endif
```



Obrázek 12: Chodba v prvním patře, v souboru výše je zakódována stěna vlevo

4 Zobrazovací program

Program pro zobrazení minimalizované scény je striktně oddělen od vytvořeného minimalizovaného modelu, což je velmi výhodné pro další vývoj projektu. Nebudeme-li spokojeni s možnostmi programu, můžeme vzít hotovou scénu a vytvořit pouze nový program, který ji zobrazí např. s podporou měkkých stínů, bez nutnosti znát kompresní algoritmy či strukturu komprimovaných dat. Výhodný je tento přístup i pro porovnání jednotlivých přístupů z hlediska datové velikosti. Můžeme využít pro zobrazení knihovnu OpenGL (nebo jakoukoli jinou grafickou knihovnu) i implementovat vlastní vykreslovací mechanismy založené na metodách sledování paprsku. Snadno by pak šlo porovnávat výhody obou přístupů.

Pro účely této práce byl zvolen přístup pomocí knihovny OpenGL, o které toho již bylo napsáno mnoho a pokud se čtenář nespokojí se zdvořilostním úvodem v další kapilote, odkáži ho na příslušnou literaturu [5] nebo [7]. Následuje několik řádků věnovaných kameře ve scéně a v dalších kapitolách se potom věnuji optimalizacím při kompilaci a následnou možnost komprese hotového spustitelného souboru.

4.1 OpenGL

OpenGL je průmyslový standard pro tvorbu nejrůznějších grafických aplikací. Ač v této aplikaci nevyužijeme jeho přenositelnost mezi různými počítačovými platformami, protože dalšími optimalizačními technikami svážeme program s OS Windows, je pro naše potřeby logickou volbou pro svoji jednoduchost a nezávislost na podporovaném hardware. Nedosáhneme sice takových vizuelně působivých výsledků jako použitím některé pokročilejší metody založené na sledování paprsku, ale to není předmětem tohoto projektu.

4.2 Pohyb kamery

Pohyb kamery lze v aplikaci řešit několika přístupy. Při tvorbě programu byly v různých fázích ve hře tyto 3:

- CADlike přístup
- volný pohyb avatara ve 3D prostředí
- automatická kamera

Níže ve stručnosti popíši výhody a nevýhody každého z nich a nakonec se podrobněji budu věnovat automatickému pohybu kamery, protože ten byl implementován ve výsledné aplikaci.

4.2.1 CADlike přístup

Toto ovládaní kamery se nejvíce rozšířilo a vešlo ve známost díky CAD programům. Podstata je v pevném středu otáčení ležícím před kamerou. Kamera se většinou ovládá myší. Hlavní nevýhody tohoto přístupu jsou:

- nutnost uživateli popsat netriviální ovládání (klávesová kombinace pro translaci, zoom, …)
- uživatel se může dostat „pod terén“ nebo procházet zdmi
- většina lidí není zvyklá CAD programy ovládat
- obtížně definovatelný střed otáčení, pro zlepšení ovládání nutnost kombinace s metodou výběru sledovaného objektu

Mezi výhody metody můžeme zařadit:

- poskytnutí volnosti uživateli, který si může prohlédnout každý detail
- vhodné pro vývoj minimalistického modelu

4.2.2 Avatar s volným pohybem

Druhou metodou je vytvoření kamery jako tzv. avatara, se kterým uživatel může volně hýbat. Tento přístup je pro běžného uživatele pravděpodobně nejintuitivnější, protože simuluje pohyb v běžném světě. Oproti prvnímu přístupu zůstává při rotaci zachována pozice kamery a mění se pouze natočení. Hlavními nevýhodami jsou již zmiňované:

- nutnost uživateli popsat netriviální ovládání (vzhledem k absenci detekce kolizí či uložení výšky terénu je třeba avatara implementovat jako „vznášedlo“ a tedy umožnit uživatelům pohyb ve všech 3 osách, včetně rotací a „úkroku“, což přináší nutnost použití více kláves)
- uživatel se může dostat „pod terén“ nebo procházet zdmi

Mezi výhody jednoznačně patří:

- poskytnutí volnosti uživateli, který si může prohlédnout každý detail
- již zmíněný intuitivní pohyb v prostoru

4.2.3 Automatická kamera

Posledním a také implementovaným přístupem je automatický předdefinovaný pohyb kamery. Tento přístup má dvě malé nevýhody:

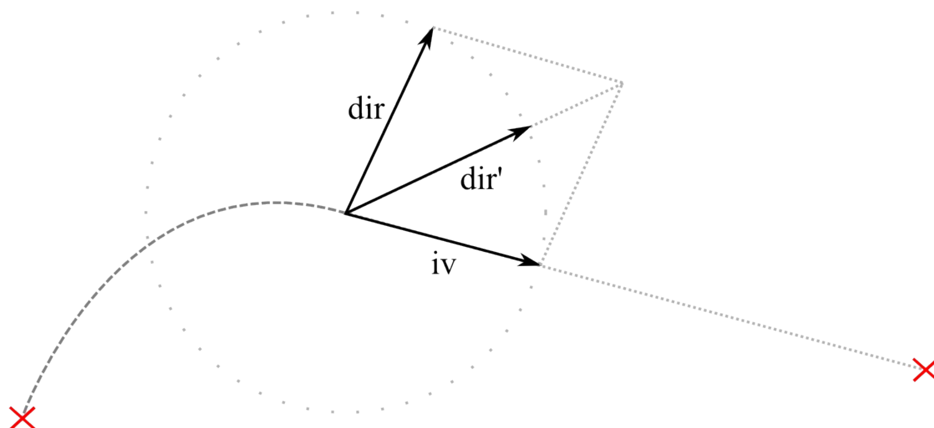
- uživatel nemá volnost pohybu, nemůže řídit prohlídku
- nutnost nadefinovat trasu pohybu

Tyto lze ale rázem přeměnit ve výhody, protože:

- uživatel se nemusí učit ovládat program
- předem lze optimalizovat trasu kamery, aby se nikdy nedostala pod terén nebo neprošla zdí a tedy i bez implementace složitějších algoritmů detekcí kolizí uživatele „udržet ve scéně“
- s předchozím bodem souvisí i možnost optimalizace LOD jak je zmíněno v kapitole 2.1.3 tak, že místa kam se kamera nedostane nebo která vidí z dálky, budou realizována s nižší úrovní detailu a naopak.

Zbývá tedy vybrat vhodnou implementaci pohybu kamery po předdefinované křivce. Jaké bychom si měli klást požadavky? Vhodnější bude vybrat křivku interpolační, protože je intuitivnější zadávat body kudy kamera projde. Bude nás také zajímat spojitost křivky, aby se kamera při pohybu příliš netrhala. Zajímavou možností jsou Hermitovské kubiky, které jsou definované právě dvěma řídícími body a dvěma tečnými vektory. Spojitost dvou úseků Hermitovské kubiky je zajištěna tečnými vektory. Lze tedy každý úsek pohybu kamery počítat lokálně. Základní informace o těchto kubikách naleznete v [1], podrobněji se těmto (a samozřejmě i jiným) křivkám věnuje [9].

Protože cílem tohoto projektu je minimalistický program, zvolil jsem na základě inspirace Hermitovskýma kubikama následně popsané řešení. Z programu Rhinoceros budeme exportovat pouze řídící body bez tečných vektorů. Tečný vektor v každém bodě bude jednotkový a jeho směr bude odpovídat směru opačném k předchozímu bodu. S optimalizací jsem šel nakonec ještě dál a místo spočtení přesného tvaru křivky v programu pouze inkrementálně měním aktuální směr pohybu kamery dokud neodpovídá směru ideálnímu. Kamera v definovaném bodě změní postupně směr k dalšímu bodu a poté se místo po křivce pohybuje již pouze po přímce. Toto řešení nemá nakonec s definicí jakékoli křivky mnoho společného, ale na druhou stranu pro jeho implementaci stačí pouze několik řádků kódu.



Obrázek 13: naznačení změny směru kamery, červené body označují vyznačenou trasu, ***iv*** *je jednotkový vektor ideálního směru,* ***dir*** *je směr kamery,* ***dir'*** *je nový směr*

Poloha i směr natočení kamery je tedy daný, zbývá určit směr tzv. *up vektoru* kamery. Jednoduchou matematikou jej umístíme do stejné roviny jako vektor natoceni kamery, která je navíc kolmá na rovinu danou osami x,y. Ze dvou možností vybereme tu s kladnou z-tovou souřadnicí, abychom neměli model vzhůru nohama.

$$up_z = \sqrt{(1 - dir_z^2)}$$

$$up_x = -dir_x * dir_z/up_z$$

$$up_y = -dir_y * dir_z/up_z$$

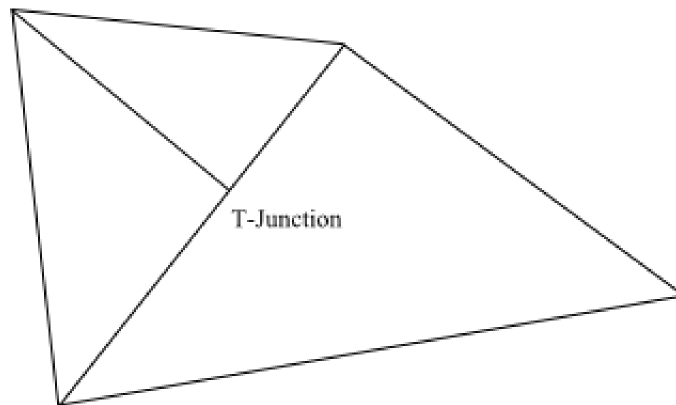
4.3 Teselace

Teselace je proces, při kterém se obecný polygon převání na nepravidelnou trojúhelníkovou síť. Jak je popsáno v kapitole 3.1.2, při exportu jsou sice pro snížení datového objemu použity čtyřúhelníky, ale tyto jsou převedeny již při dekompresi dat na trojúhelníky. Tedy v samotném zobrazovacím programu nepotřebujeme provádět teselaci v pravém slova smyslu, protože již máme k dispozici nepravidelnou síť trojúhelníků.

V kapitole 2.1.3 jsme ovšem popsali jak budeme pro snížení počtu uložených vrcholů se snažit vytvářet co největší polygony. Tato datová úspora se však nyní projevuje negativně při vykreslování scény, především z hlediska osvětlení (resp. stínování). Nejvíce patrné by bylo při použití konstantního stínování, kdy by vznikly velké plochy s konstantní barvou a byly by viditelné hrany polygonů. V OpenGL se ovšem běžně používá Gouraudovo stínování. Zapnout lze jediným příkazem *glShadeModel(GL_SMOOTH).* Při použití Gouraudova stínování vypočítá OpenGL hodnotu osvětlení ve všech vrcholech polygonu a hodnotu barvy plošky určí jako barevný přechod mezi nimi. Výhodou jsou jemnější hrany zaoblených objektů a absence viditelné hrany mezi sousedícími ploškami, které leží ve stejné rovině. Vizuelním chybám se však nevyhneme ani v tomto případě. Tento typ stínování vypočítává hodnotu osvětlení pouze ve vrcholech plošky, tedy zanedbává světelné podmínky na ploškách. Výsledkem je absence odlesků na velkých plochách, kterých je ovšem v exportovaném modelu velmi mnoho, jak jsem zmínil výše. Více o stínování se můžete dočíst v knize Moderní počítačová grafika [1] nebo přímo o použití s OpenGL pojednává Průvodce programátora [5].

Z tohoto důvodu je třeba získanou síť trojúhelníků opětovně teselovat (i trojúhelník je obecný polygon, lze tedy dle mého názoru použít pojem teselace i v tomto případě, kdy již vstupem je nepravidelná trojúhelníková síť a výstupem pouze její jemnější varianta). Zbývá tedy určit způsob. Možnosti použití standardních algoritmů se nám snižují absencí topologických informací

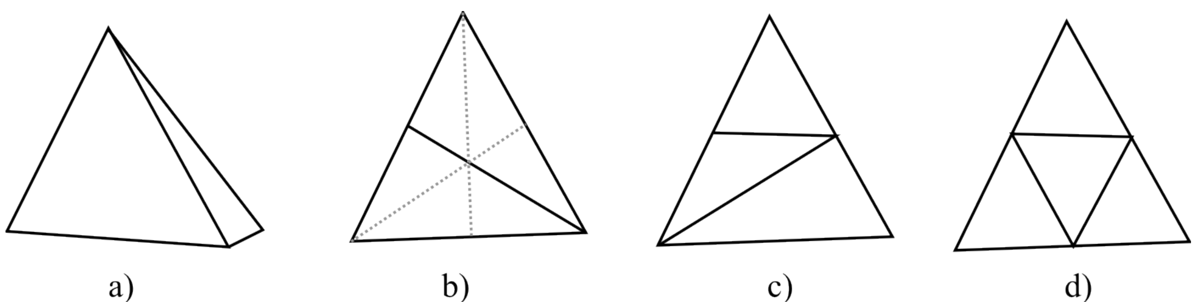
o sousedních polygonech. Je také nebezpečí vzniku tzv. T-Junctions. To jsou vrcholy trojúhelníkové sítě, které zároveň leží na hraně další plošky nebo vlivem dalších úprav sítě vznikne vedle nich díra. Nejlépe je to vidět na obrázku 14.



Obrázek 14: T-Junction

Abychom předešli vzniku T-Junctions bez topologických informací o sousedních polygonech, musíme provést teselaci tak, že budeme ke každému polygonu přistupovat stejně. Nabízí se triviální řešení rozdělit každý trojúhelník na 4 menší tak, že každou jeho hranu rozdělíme v polovině. Společná hrana dvou polygonů bude tedy pro každý z nich rozdělena na 2 části a byť toto rozdělení proběhne u obou polygonů nezávisle, nedojde ke vzniku T-Junction, protože proběhne u obou stejným způsobem. Tento algoritmus můžeme aplikovat samozřejmě opakovaně. Výhodou je jednoduché a efektivní dosáhnutí našeho cíle, velkou nevýhodou ovšem nerovnoměrnost vzniklé trojúhelníkové sítě. Stále bude obsahovat plošky s velmi odlišným obsahem.

Potřebovali bychom tedy algoritmus řídit tak, že trojúhelníky s velkou plochou budou rozděleny a ty malé ponechány. Pokud budeme ovšem algoritmus řídit na základě plochy trojúhelníku, dojde ke vzniku T-Junctions, protože sousední trojúhelníky zřejmě nemusí mít podobný obsah, jak ukazuje obrázek 15 a). Existuje ovšem jedno kritérium, které ovlivňuje obsah plošky a navíc jej mají sousedící polygony stejné, a tím je délka hrany. Algoritmus je tedy postaven tak, že prochází pole trojúhelníků a každý trojúhelník nahradí 1-4 novými trojúhelníky na základě délky jeho hran jak ukazuje obrázek 15 b) až d). Délku hrany budeme pro určování zda hranu rozdělit či ne porovnávat s předem definovanou konstantou `max`. Algoritmus budeme opakovat tak dlouho, dokud dojde k nahrazení alespoň jednoho trojúhelníku větším počtem. Vznikne síť trojúhelníků, jejichž hrany budou mít délku z intervalu `(max/2, max>`, kde `max` je kritérium pro rozdělení hrany – její maximální délka. Nejmenším možným trojúhelníkem, který takto vznikne je teoreticky rovnostranný trojúhelník o hraně `max/2`, největším zase rovnostranný trojúhelník o délce hrany `max`. Obsah rovnostraného trojúhelníku je závislý na délce jeho strany v druhé mocnině. Poměr obsahu ploch bude tedy v nejhorším možném případě 4:1, což je uspokojivý výsledek.



Obrázek 15: rozdíl mezi sousedy (a) a možnosti vzniku nových trojúhelníků (b-d)

Námi vytvořený model obsahuje před teselací 24 508 trojúhelníků (pozor, nejedná se o počet uložených polygonů, ale o počet polygonů scény po rozkopírování opakovaných částí). Teselací se jejich počet změní v závislosti na nastavené maximální délce hrany jak ukazuje tabulka 2. Při délce hrany 1 m jsem již pozoroval lehké snížení výkonu celé aplikace, protože zobrazovací program je implementován naivně bez řešení viditelnosti polygonů, apod. Při maximální délce hrany 5 m zase vznikají příliš velké trojúhelníky a efekt teselace je téměř neznatelný. Ideální hodnotou jsou tedy 2 metry, které jsou nastaveny i v ukázkovém programu.

Max. délka hrany	Polygonů po teselaci	Nárůst [%]
2 m	112 786	460 %
1 m	371 305	1515 %
5 m	38 783	158 %

Tabulka 2: výsledky teselace

Pro úplnost ještě uvádím algoritmus pro provedení teselace.

```
Pro každý objekt prováděj:
      projdi v cyklu všechny trojúhelníky:
            spočti délku všech 3 hran
            na základě počtu hran delších než limit vyber 1 z 8 šablon
            odstraň původní trojúhelník
            vlož nové trojúhelníky dle předdefinované šablony
dokud vznikají nové polygony.
```

4.4 Optimalizace kódu

Oblast, kterou nelze opomíjet, je optimalizace samotného zdrojového kódu v jazyce C a způsob jeho kompilace. Vytvořme si takovýto ukázkový soubor („prázdný program“):

```
int main(int argc, char *argv[]) {
  return 0;
}
```

Po prosté kompilaci tohoto programu získáme EXE soubor o velikosti cca 15 kB. To je v porovnání s požadovanou koncovou velikostí řádově v desítkách kilobajtů velmi mnoho. Velikost můžeme ještě snížit použitím programu *strip.exe*, který odstraní ze souboru zbytečné symboly a sekce. I přesto má „prázdný program“ velikost 5 kB. To je způsobeno výchozím přilinkováním některých základních knihoven. Tyto knihovny obsahují mnoho funkcí, které v programu nevyužijeme, ale ve výsledném zdrojovém kódu zabírají místo. Zakázat linkování těchto knihoven můžeme pomocí přepínačů kompilátoru gcc *-nodefaultlibs* a *-nostdlib*. Tím ovšem přijdeme i o některé užitečné funkce. Po spuštění nám kompilátor, resp. linker, hlásí následující chybu:

```
C:\MinGW\bin\..\lib\gcc\mingw32\3.4.2\..\..\..\..\mingw32\bin\ld.exe: warning:
cannot find entry symbol _mainCRTStartup; defaulting to 00401000
```

Což znamená absenci funkce pro spuštění programu – vstupní bod do programu. Nejsnažším řešením je přejmenování funkce *main* na *mainCRTStartup*. Poté již kompilace projde a vznikne program o velikosti 1536 bytů. Velmi malý, ale zcela neužitečný.

Při implementaci programu k nějakému konkrétnímu účelu nám zcela jistě budou chybět i další funkce ze standardních knohoven. Tyto můžeme nahradit funkcemi z WinAPI (správa paměti, IO). V našem programu jde pouze o funkce správy paměti, jejichž ekvivalenty jsou:

malloc → GlobalAlloc
free → GlobalFree
memcpy → CopyMemory

Část práce zajistí samotné OpenGL (zobrazení okna a vykreslování) a některé funkce si můžeme nadefinovat sami voláním instrukcí FPU – matematické operace, apod... Níže uvádím ukázku implementace funkce pro výpočet cosinu. Další matematické funkce používané v programu vytvoříme obdobně jen záměnou instrukce FCOS za FSQRT a FSIN. Jiné funkce nepoužíváme.

```
float cos(float val) {
  asm("fld %1;"
      "fcos;"
      "fstp %0;"
      : "=g" (val)
      : "g" (val)
     );
  return val;
}
```

Na druhou stranu rozdíl 3 KB ve výsledném programu (a to za ideálních podmínek) není až tak markantní. V praxi ještě musíme odečíst velikost doimplementovaných funkcí. Tyto techniky jsou tedy spíše vhodné pro *demo aplikace* přihlášené do soutěží, kde je striktní velikostní omezení, jak popisuji v kapitole 1.1 a záleží na každém kilobajtu. Pokud se snažíme pouze optimalizovat velikost programu za účelem kupříkladu rychlejšího stažení skrze bezdrátovou síť, není pro nás datová úspora až tak lákavá, aby vyvážila pracnost implementace.

4.5 Kompresní programy

V momentě, kdy jsme vytěžili maximum z komprese statických dat, optimalizovali zdrojové kódy a nejsme spokojeni s výslednou velikostí souboru, máme ještě jeden nástroj pro její snížení. Stejně tak jako lze komprimovat obecná data do archivu a lze vytvářet samorozbalovací archivy, můžeme tento postup aplikovat i na náš exe soubor. Vzhledem ke specializovanému použití ovšem nevyužijeme služeb známých archivátorů, ale podíváme se na možnosti programů přímo pro kompresi spustitelných souborů.

Stejně jako běžný samorozbalovací program, přidá i tento specializovaný do spustitelného souboru kód pro jeho dekompresi. Jejím cílem není ovšem disk, nýbrž přímo operační paměť počítače. Uživatel tedy kromě delšího spuštění programu nic nepozná. Časové zpoždění při startu aplikace není však na dnešních výkonných strojích také příliš znatelné. Vzhledem k úzké specializaci těchto programů – na spustitelný kód – dosahují tyto mnohem lepších výsledků, co se stupně komprese týče. Více informací je k dispozici na internetových stránkách [4].

Pro tento účel existuje mnoho programů, lišících se především v algoritmech komprese a dekomprese a z toho plynoucí velikosti přidaného kódu. Toto je stěžejní především v aplikacích s omezenou velikostí, kde ideální komprese za cenu přidání relativně většího množství kódu pro dekompresi je ve výsledku horší než opak. Existují proto kompresní programy orientované přímo pro intra a dema s omezenou velikostí. V tabulce 3 vidíte porovnání několika z nich i s těmi komplexnějšími jako je např. UPX.

	Po kompilaci	[%]	Po kompilaci a stripnutí	[%]
Bez komprese	111 249 B	100 %	79 360 B	71 %
UPX 3.03w	68 753 B	62 %	36 864 B	46 %
Kkrunchy 0.23 alpha	30 720 B	28 %	**30 720 B**	39 %
MPRESS v2.12	33 280 B	30 %	33 280 B	42 %
FSG v2.0	35 869 B	32 %	35 869 B	45 %

Tabulka 3: porovnání stupně komprese jednotlivých programů

V tabulce je zachycena jak absolutní velikost, tak i procentuelní vyjádření. Porovnával jsem také míru komprese s a bez předchozího použití programu *strip.exe*. Z tabulky je zřejmé, že všechny kompresní programy kromě UPX 3.03w odstraní z kódu přebytečné symboly a sekce stejně jako to dělá program *strip.exe* a až potom aplikují kompresi. Nejlepších výsledků dosáhl program *Kkrunchy 0.23 alpha*.

To je způsobeno především tím, že jeho autor jej naprogramoval přímo pro kompresi inter a demo aplikací s velikostním omezením pro soutěže, takže se vyznačuje velmi malým přidaným kódem pro dekompresi.

Je třeba také podotknout, že vzhledem k charakteru tohoto druhu komprese ji nelze pro dosažení lepších výsledků aplikovat vícekrát. Kromě toho, že opětovným přidáním kódu pro dekompresi by velikost spíše narostla, tak samotná komprese neproběhne. Kompresní programy totiž odstraní z programu symboly, které používají ke zjištění typu souboru a jeho následnou kompresi. Nelze tedy ani komprimovat jeden soubor postupně různými kompresními programy.

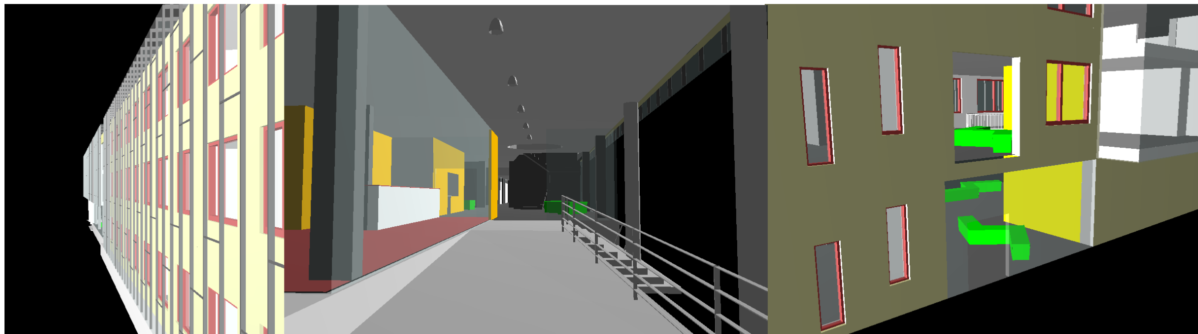
5 Závěr

Text této práce pojednává o problematice tvorby grafických aplikací s omezenou velikostí, včetně algoritmů vhodných pro kompresi polygonální reprezentace modelů. Seznamuje čtenáře s postupem tvorby takového minimalistickeho modelu a popisuje nástroje k tomu použité. V závěrečných kapitolách se potom věnuje programu pro zobrazení vytvořeného modelu a možnostem jeho další komprese.

V rámci řešení práce bylo nutné prostudovat různé techniky komprese, nejen zde uvedené, a zhodnotit jejich efektivnost a použitelnost pro zobrazování architektury – tedy konkrétně budov areálu Božetěchova. Důležité pro pokračování projektu bylo také navržení obecného postupu a nástrojů pro snadnou tvorbu i dalších minimalistických modelů podobného charakteru.

Projekt se může dále ubírat směrem vývoje a zdokonalování jednotlivých částí s cílem vytvoření univerzálního *demotool*, tedy nástroje pro vytváření inter. Techniky, které popisuje, mohou být využity právě v těchto soutěžních disciplínách. Ovšem jako plnohodnotný nástroj by musel být rozšířen i o další oblasti jako je práce se zvukem, apod. Nevýhodou popsaných postupů a eventuelního demotoolu, který by znich mohl vzniknout by ovšem byla skutečnost, že zobrazuje realitu. Ostatní intra se soustředí především na minimální velikost a maximální dojem a podle toho i volí použití algoritmů. Výsledky tohoto projektu by byly lépe porovnatelné s intry, kde je nutné zobrazovat konkrétní model.

Z praktického hlediska se tento projekt ukázal tedy spíše nepoužitelný, komerčně nezajímavý. Práce spojená s vytvořením minimalistické reprezentace není vyvážena přínosem výsledků. Pro nějaké obecnější použití v praxi je zase problém svázání s konkrétní platformou. Použité techniky také nutí autora vytvořit model jako celek a tak i s ním pracovat, což je zase nevýhodné při potřebě zobrazování náhledů či samostatných částí scény.



Obrázek 16: Ukázka z výsledné aplikace

Pokud bych měl navrhnout další pokračování projektu, doporučil bych spíše pozastavení této vývojové větve a zvolení jiného přístupu. Mnohem zajímavější mi přijde myšlenka vytvoření serverové aplikace (je o diskuzi zda by se mělo jednat o robustní spustitelnou aplikaci nebo o jednodušší naskriptování potřebné programové logiky v rozšířeném jazyku PHP, což by mohlo pomoci masovějšímu rozšíření), která by pracovala s nově vytvořeným otevřeným datovým formátem. Na ten bych kladl požadavky, aby kromě uložení 3D scény (geometrie, topologie, materiály, světla, textury, ...) umožňoval i vyznačení skutečných geografických souřadnic a mohl tedy sloužit k navigaci. Mohl by také obsahovat značky pro vložení externích informací, představte si například virtuální procházku po úřadě, kde byste narazili na úřední desku s aktuálními informacemi z webových stránek nebo po nákupu ve virtuálním obchodě přejít k pokladně se skutečnou pokladní nasnímanou kamerami a vloženou do scény. Další věc je kdo by chtěl takový program používat, ale to už asi není otázka pro jeho vývojáře. Říkáte si jistě, jak toto souvisí s prací zaměřenou na minimalistickou reprezentaci scény. Velmi úzce. Jak bylo zmíněno v kapitole 1.2, při masovém rozšíření takové aplikace bude každý ušetřený byte násoben milióny uživatelů, kteří by si jej stáhli. Má tedy spíše smysl zabývat se kompresí takových dat a formátů, které mají šanci na masové

rozšíření. Jak jsem ovšem zmínil výše, při velkém objemu dat a složitých datových formátech, je nereálné zabývat se každým polygonem. Samotné fungování bych si tedy představil spíše tak, že budeme mít na serveru uložený propracovaný model, jehož velikost bude „obrovská“ (což ničemu nevadí v kontextu cen datových médií), s ním bude pracovat serverová část programu, která bude muset zpracovávat velké množství požadavků. Nebude moci tedy dělat žádné velké přepočty. Samotná úspora pak bude spočívat v možnosti vrácení jen části scény, kde se zrovna bude klient nacházet a případně v nějaké další standardní kompresi, která ovšem nebude závislá na konkrétních komprimovaných datech a bude s nimi pracovat jako s celkem.

Takový projekt si myslím, že má smysl. Svým rozsahem by ovšem přesáhl rozsah jedné práce. Své by si na něm našli jak experti na zobrazovací programy, kteří by je vyvíjeli pro různé platformy a metody zobrazení od OpenGL po implementaci sledování paprsku, přes zkušené tvůrce komunikačních protokolů, kteří by se museli vyrovnat s možností klientské části požadovat pouze taková data, která potřebuje, až k programátorům vícevláknových serverových aplikací a návrhářům datových struktur.

Literatura

[1] Žára J., Beneš B., Sochor J., Felkel P. Moderní počítačová grafika.
Brno: Computer Press, a.s., 2004.

[2] David Rutten, Robert McNeel & Associates Rhinoscript101 for Rhinoceros 4.0
http://download.mcneel.com/s3/mcneel/rhino/4.0/docs/en/RhinoScript101.zip (leden 2009)

[3] Rhinoscript reference. Dokument na internetu
http://www.kxcad.net/Rhinoceros/RhinoScript/index.htm (leden 2009)

[4] Executable compression. Dokument na internetu
http://en.wikipedia.org/wiki/Executable_compression (leden 2009)

[5] Shreiner D., Woo M., Neider J., Davis T. OpenGL Průvodce programátora.
Překlad Jiří Fadrný. Brno: Computer Press, a.s., 2006.

[6] Tomas Moeller, Eric Haines, Real-Time Rendering.
A K Peters, Ltd., 1999

[7] Richard S. Wright, Jr., Michael Sweet, OpenGL SuperBible
Waite Group Press, 2000

[8] M. E. Mortenson, Mathematics for Computer Graphics Applications
Industrial Press, Inc., 1999

[9] M. E. Mortenson, Geometric Modeling
John Wiley & Sons, Inc., 1997

[10] Rhino3D.cz, dokument na internetu
http://www.rhino3d.cz/ (květen 2009)

[11] Rhinoceros – developer tools, dokument na internetu
http://www.rhino3d.com/developer.htm (květen 2009)

[12] Energy and the Internet, dokument na internetu
http://googleblog.blogspot.com/2009/05/energy-and-internet.html (květen 2009)

[13] Triangle strip, dokument na internetu
http://en.wikipedia.org/wiki/Triangle_strip (květen 2009)

[14] D. Zorin, P. Schröder, DeRose, Kobbelt, Levin, Sweldens, Subdivision for Modeling and Animations
SIGGRAPH Course Notes, 8 2000.

[15] S. D. Ebert, F. K. Musgrave, D. R. Peachey, K. Perlin and S. Worley. Texturing & Modeling
A procedural Aproach, third edition. Morgan Kaufmann Publishers, 2003