Rok odevzdání 2006

Diplomová práce

Optimalizace detekce kolizí ve virtuální městské zástavbě

Vypracoval Bc. Jan Kytýr

Zadání:

Zadání:

1. Nastudujte optimalizační techniky pro virtuální scény založené na stromových strukturách typu quadtree a octree.
2. Vytvořte aplikaci, která vygeneruje rozsáhlé "virtuální město" a implementuje pohyb uživatele v této scéně.
3. Navrhněte a implementujte algoritmy pro detekci kolizí. Pro optimalizaci těchto algoritmů použijte stromové struktury typu quadtree nebo octree.
4. Proměřte výkonnost algoritmu v závislosti na použité hloubce stromových struktur.
5. Vyhodnoťte výsledky práce.

Prohlášení:

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením Ing. Jana Pečivy. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal. Tímto bych mu rád poděkoval za vedení a odborné rady při vypracovávání mojí diplomové práce.

Podpis ……

Abstrakt :

Diplomová práce se zabývá generováním a zobrazováním schématu městské zástavby, detekcí kolizí ve scénách, reakcemi na ně, jejich optimalizací a algoritmy OctTree a QuadTree. Cílem bylo vytvořit program, který náhodně vygeneruje a vykreslí město podle daných parametrů, který implementuje pohyb uživatele ve scéně a řeší optimalizace detekce kolizí pomocí stromových algoritmů OctTree, nebo QuadTree.

Klíčová slova :

Zástavba města, generování scén, zobrazování scén, textury, budovy, silnice, křižovatky, pohyb kamery, OctTree, QuadTree, VRML, Coin3D

Abstract :

This graduation theses discuss generation and viewing of urban development, collision detection in scenes and reaction onto them, their optimization and with algorithm QuadTree and OctTree. The point is to create application which randomly generate and view urban development based on input parameters. Which implements user motion in scene and solving collision detection with tree algorithm OctTree or QuadTree.

Key Words :

Urban development, scene generation, view scene, texture, building, road, crossroad, camera motion, OctTree, QuadTree, VRML, Coin3D

Obsah:

Číslo stránky

Obsah:

Číslo stránky

Úvod

Má diplomová práce se skládá z dvou hlavních bloků, prvním je vytvoření rozsáhlého virtuálního města. Jedná se o vygenerování struktur vzhledově se blížících městské zástavbě a vytvoření prostředků na zobrazování těchto algoritmem vygenerovaných struktur. Druhým blokem je studium algoritmů detekce kolizí, struktur QuadTree a OctTree a následná aplikace této teorie do vytvořeného virtuálního města.

V první kapitole se věnuji algoritmům detekce kolizí. Druhá kapitola hovoří o algoritmech QuadTree a OctTree. Třetí kapitola se zabývá softwarem použitým k vypracování projektu, jeho stručným popisem a naznačením historie jeho vývoje. V další kapitole pokračuji popisem použitých knihoven Coin3D, jejich využitím a posléze se věnuji samotné implementaci mého programu. Pátá kapitola je věnována fyzikálnímu modelu a prohlížení scény. Další kapitola pak popisuje samotnou strukturu programu, rozvržení tříd a metod. V sedmé kapitole popisuji použité algoritmy a implementaci. Osmá kapitola zachycuje konkrétní aplikaci detekce kolizí v mém projektu a jejich složitosti. Předposlední kapitola se zabývá testováním vlastností detekce kolizí. Poslední kapitola nejprve stručně zhodnotí výsledek práce a potom rozebere možnosti rozšiřování a vývoje tohoto projektu.

Pohled do města

Obrázek 1

1. Kapitola - Datové struktury a algoritmy detekce kolizí

1.1 - Detekce kolizí v prostoru

Nejpodstatnější ve fyzikálních modelech pracujících v reálném čase je zajistit realističnost scény a fyzikálního modelu. S tím souvisí i interakce jednotlivých objektů ve scéně, detekce jejich kolizí a řešení těchto kolizí. V mém diplomovém projektu jsou řešeny všechny jednotlivé aspekty, přestože se práce zaměřuje hlavně na detekci kolizí, nemohly tyto aspekty zůstat opomenuty.

Jsou dva základní přístupy, jak algoritmicky řešit detekce kolizí. Jsou to boundingSphere (v překladu z angličtiny by se dalo psát například obalová koule, ale běžně se tento výraz nepřekládá), detekce a detekce Triangle-to-Triangle[1] . Přístup boundingSphere znamená, že máme vybrané obalové koule tělesa (obalové těleso objektu znamená, že celý objekt se vyskytuje uvnitř tohoto tělesa) a jednoduchým testem na vzdálenost středů odhalujeme možnost kolize (v případě boundingBox je obalovým tělesem krychle nebo kvádr a provádí se test na průnik). Přístup Triangle to Triangle znamená, že používáme parametrická srovnávání na odhalení kolizních bodů mezi jedním trojúhelníkem a rovinou trojúhelníku druhého. Tak se rozhodne které kolizní body leží uvnitř protějšího trojúhelníku a které ne[1].

1.2 - Detekce kolizí algoritmem boundingSphere

Detekce kolizí je nejlepší v hierarchických krocích. Nejprve tedy testovat kolizi obalových těles objektů, potom obalových těles polygonů a na závěr jednotlivé trojúhelníky. Začneme generováním boundingSphere (obalových koulí), jejichž výpočet je velmi výkonově jednoduchý. Všechno, co potřebujeme, je najít střed objektu, potom spočítat nejvzdálenější bod objektu k získání poloměru koule. Uložením poloměrů pak můžeme testovat kolize pomocí poloměrů dvou objektů (pokud jsou středy blíže než součet poloměrů, nastává kolize).

Pojďme si tedy tento algoritmus projít krok po kroku. Nejdříve potřebujeme zjistit centrální bod. Jednou z možností je vytvořit boundingBox a najít požadovaný střed pomocí průniku diagonál protilehlých vrcholů. Na spočítání boundingBoxu potřebujeme v daném objektu najít minimální a maximální hodnoty x, y, z. Ty můžeme zjistit například iteračním procházením od vrcholu k vrcholu a aktualizováním lokálního maxima a minima. Po zkontrolování všech vrcholů vytvoříme z výsledných hodnot boundingBox. Po vytvoření boundingBoxu najdeme střed boundingSphere zprůměrováním maximálních a minimálních hodnot v boundingBoxu. Poloměr boundingSphere lehce spočítáme projitím všech vrcholů a nalezením nejvzdálenějšího od středu objektu (potřebný poloměr je vzdálenost mezi těmito dvěmi body).

Nyní máme boundingSphere jednotlivých objektů a jednoduchý nástroj na testování jejich kolizí. V případě, že vzniká kolize na úrovni boundingSphere, tak se dostáváme k testům jednotlivých trojúhelníků a metodě Triangle-to-Triangle.

1.3 - Detekce kolizí algoritmem Triangle-to-Triangle

Metoda detekcí Triangle-to-Triangle je také docela jednoduchá na pochopení, ale obsahuje pár matematických triků, které raději podrobněji popíšu. Máme-li dva trojúhelníky v třírozměrném prostoru, tak o nich potřebujeme nashromáždit mnoho informací. Začneme s tím, že najdeme roviny ve kterých trojúhelníky leží. Rovnice roviny v prostoru je $A*x + B*y + C*z + D = 0$. Hodnoty A, B, C, D zjistíme vektorovým součinem jednotlivých vektorů. První dva vektory roviny spočítáme jednoduše jako rozdíly souřadnic bodů : $v_1 = b-a$, $v_2=c-a$, kde a,b,c jsou vrcholy trojúhelníku. $v_3 = v_1 \times v_2$, kde operace $\times$ je vektorovým součinem obou vektorů. Hodnoty A, B ,C jsou potom hodnoty souřadnic x, y, z vektoru v_3. Potom tedy pokud bod p (x_0 ,y_0 ,z_0) je bodem na polygonu, pak podle pravidla $A*x + B*y + C*z + D = 0$ je $D = - A*x_0 - B*y_0 - C*z_0$.

1.3.1 - Průnik přímky s rovinou

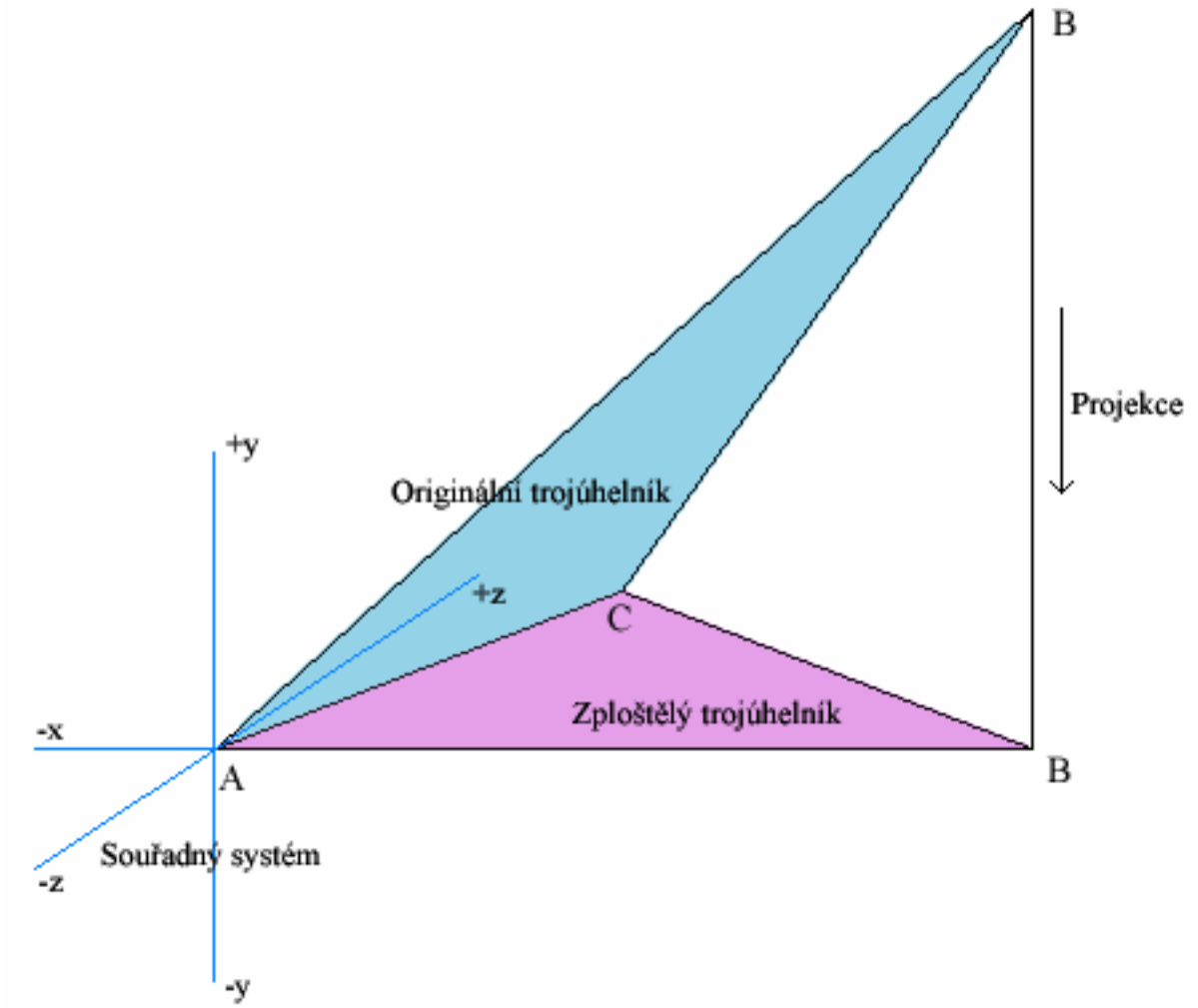
Jakmile máme vyčísleny roviny trojúhelníků, můžeme přistoupit k dalšímu kroku a hledat, zda trojúhelník$_2$ koliduje s rovinou trojúhleníku$_1$. Na to potřebujeme několik kroků. Hlavní myšlenka je v tom, že vezmeme přímku, kterou definují dva vektory trojúhelníku$_2$ a zjistíme, který bod přímky protíná rovinu trojúhelníku$_1$. Pokud kolidující bod je mezi dvěmi vektory, pak trojúhelník$_2$ koliduje s rovinou trojúhelníku$_1$. Pokud mezi dvěma vektory není, opakujeme tento postup pro další dvě přímky trojúhelníku$_1$.

Výpočet průniku přímky s rovinou provedeme pomocí parametrických rovnic. Vezmeme dva vektory a(x_0, y_0, z_0) a b(x_1, y_1, z_1). Nastavíme a(x_0, y_0, z_0)*t = b(x_1, y_1, z_1)*(t-1), kde t je interpolační faktor od nuly do jedné. Pokud t je 0 jsme v bodě b a pokud t je jedna, jsme v bodě a. Odtud $A*(x_0*t + x_1*t*(1-t)) + B*(y_0*t + y_1*t*(1-t)) + C*(z_0*t + z_1*t*(1-t)) + D = 0$. Po úpravě dostáváme $t = - (A*x_1 + B*y_1 + C* z_1 + D) / (A*(x_0 - x_1) + B*(y_0 - y_1) + C*(z_0 - z_1))$. Tím získáme výsledný bod průniku na přímky a roviny $c = a*t + b*(1-t)$.

1.3.2 - Triangle „flattening“ (zploštění)

Předpokládejme pravotočivý souřadný systém, pak si můžeme zploštění přestavit jako projekci trojúhelníku do plochy jedné z rovin souřadného systému (viz. obrázek 2). Tím můžeme například ztratit y koordináty a přitom zachovat x i z. V našem algoritmu potřebujeme vždy ztratit tu souřadnici, která má být zploštěna. Dobrou cestou při rozhodování, která koordináta má být zploštěna, je sledování normál roviny. Pokud zjistíme, které hodnoty souřadnic mají v absolutní hodnotě nejvyšší hodnotu, můžeme najít rovinu ke zploštění naproti k nim, kde trojúhelník nebude v „přímé linii“ (například vertikálně položený trojúhelník ztrácí koordináty y souřadnice). Pokud tedy bude největší hodnota x, budeme provádět projekci y a z. Bez ohledu na orientaci trojúhelník, který bude vytvořen, bude zploštěný. Dosažené výsledky jsou vhodné pro následující použití lineární algebry na zjištění bodu průniku. Pokud zplošťujeme tímto jednoduchým způsobem, tak požadovaný průnik leží uvnitř zploštěného trojúhelníku.

Projekce / zploštění trojúhelníku do roviny xz eliminací souřadnice y



Obrázek 2

1.3.3 - Point-in-Triangle test

Ke zjištění, zda zploštěný bod průniku leží ve zploštěném trojúhelníku, je několik populárních cest. Jednou z nich je vytvoření rovnic pro každý ze zplošťovaných trojúhelníků. Všimněme si, že bez ohledu na to, kterou rovinu promítáme, budeme se stále pohybovat v x, y souřadnicích zploštěného trojúhelníku. To proto, že jsme projekcí vektorů efektivně zredukovali 3D problém na 2D. Nejdříve potřebujeme najít bod, který určitě náleží trojúhelníku. Nejsnazší cestou je nalezení středu trojúhelníku pomocí souřadnic jeho vrcholů: [($x_0 + x_1 + x_2$)/2, ($y_0 + y_1 + y_2$)/2)]. Tak zjistíme, kterým směrem je vnitřek trojúhelníku a kterým je vnější prostor.

Máme tedy dva vektory v_0 a v_1, nejprve najdeme rovnici přímky vedoucí skrz ně: y= m*x + b. Bod b najdeme pomocí přímky, úhlu a bodu. Když máme rovnici přímky v průsečíku, můžeme zjistit, zda zploštělý bod leží na straně přímky, která je směrem dovnitř trojúhelníku nebo mimo trojúhelník. To zjistíme porovnáváním hodnot y. Pokud přidáme x-ové koordináty zploštělého bodu průniku na přímku y= m*x + b, dostaneme hodnotu y bodu přímky na souřadnicích x. Nakonec zjistíme, zda středový bod ([($x_0 + x_1 + x_2$)/2, ($y_0 + y_1 + y_2$)/2)]) je „nad“ nebo „pod“ přímkou porovnáním jejich hodnot y. Víme, že tento bod je uvnitř trojúhelníku a pokud jeho hodnoty y jsou ve stejným směru od přímky jako bod průniku, pak i bod průniku leží uvnitř trojúhelníku vzhledem k úsečce ab. Tento postup zopakujeme i pro úsečky bc a ca. Pokud bod průniku leží uvnitř pro všechny úsečky, pak leží určitě uvnitř trojúhelníku.

1.3.4 - Závěr

Postupně kontrolujeme všechny strany obou trojúhelníků, dokud nenarazíme na některou stranu, která by kolidovala s druhým trojúhelníkem. Pokud žádná ze stran nekoliduje, pak oba trojúhelníky spolu jistě nekolidují. Jak je vidět z předchozího, detekce kolizí jednotlivých trojúhelníků už je trochu komplikovanější, se znatelně složitějším algoritmem a větším množstvím matematických operací, ale samotný výpočet je poměrně rychlý a efektivní. Obsahuje pouze operace sčítání, odčítání, násobení a dělení, které nejsou příliš náročné na výkon počítače.

2. Kapitola - Algoritmy OctTree a QuadTree

2.1 - Datové struktury a definice QuadTree a OctTree

V aplikacích počítačové grafiky potřebujeme zpracovávat dvě odlišné datové struktury - rastrové a vektorové. Rastrový přístup umožňuje modelování obrazu jako souboru čtvercových buněk stejného rozměru - pixelů, přičemž každému pixelu je přiřazena nějaká barva. Snaží se tím dosáhnout maximální flexibility modelu se světelnými body obrazovky. Každý pixel je přiřazen jednomu světelnému bodu obrazovky.

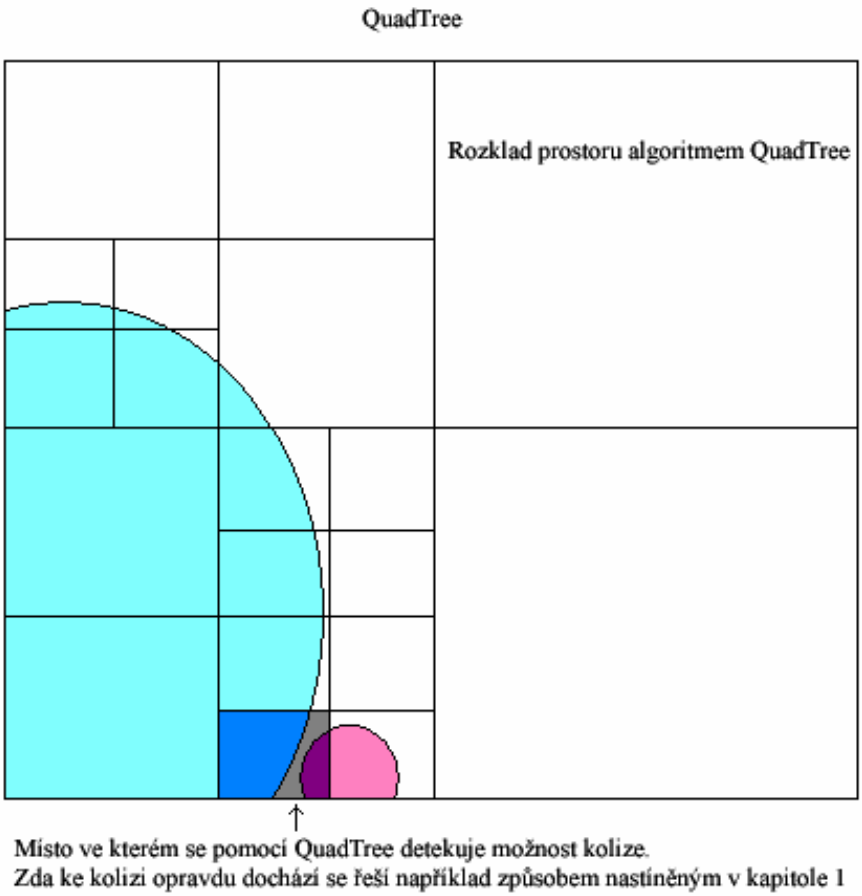
Narozdíl od přímého generování obrazu, vektorový formát je ideální pro vyjádření geometrie prostoru. Vektorová data obsahují body, přímky, polygony a další… Oba datové formáty mají zřejmé reprezentace, které jsou minimální ve smyslu poskytnutí struktury dostačující k provedení aplikace.

Pokud jsou objekty, které se snažíme zobrazit na obrazovku širší, než je síť obrazovky, pak nastává větší logistický problém. Jsou dva způsoby, jak jej řešit. První cesta je založena na object-space hierarchii a druhá na image-space hierarchii, kterou řešíme pomocí OctTree.

2.1.1 Rozklad QuadTree

Jedna z běžně používaných 2D image-space hierarchií je znázorněna QuadTree datovou strukturou (viz. třetí obrázek). Je zkonstruována tímto způsobem: začneme s obrazem a zkontrolujeme, jestli má jednoduchý popis a tedy jestli nepotřebuje další hierarchické strukturovaní. Jestliže to není ten případ, pak je obrazový prostor rozdělen do 4 stejných čtvercových sekcí – kvadrantů (v případě OctTree oktantů) jejichž součet pokrývá původní prostor scény. S každým z těchto nových prostorů je zacházeno jako by byly izolovány od ostatního prostoru a každý je zkoumán, jestli má, nebo nemá jednoduchý popis (zda se bude dále dělit). Takto se rekurzivně pokračuje do té doby, než je dosaženo potřebného stavu, nebo dosažena maximální nastavená přesnost rozkladu. Test na potvrzení, jestli je či není prostor obrazu jednoduše popsán, se nazývá listové kriterium (leaf criterium[2]), protože prostor se soustřeďuje do listových uzlů (stejný postup je použit v ukládání dat v mém projektu), které reprezentují hierarchickou strukturu.

Je mnoho různých variant algoritmů a datových struktur založených na QuadTree, které odlišují pouze v požadavcích na listové kriterium. To je velmi užitečné, protože to pomáhá konstruování integrovaných grafických databází, které obsahují širokou oblast dat.



Obrázek 3

Je mnoho pravděpodobných listových kriterií. Hledáme-li listové kritérium, které opravdu potřebujeme, pak hledáme podskupinu pravděpodobného obrazového prostředí, kde grafické úlohy, které chceme řešit, můžeme řešit snadno. Je také potřebujeme, aby libovolný prostor obrazu mohl být rozložen do kvadrantů, které vyhovují kriteriu. Tedy, například, kdybychom měli uložit vektorová data v prostoru obrazu, můžeme vyslovit hypotézu o specifikovaní kriteria, že v jednom segmentu může být maximálně jedna přímka (vektor). Pak budeme určitě zklamáni, protože v například v obrazech obsahujících přímky (nebo vektory) velmi často nastane situace, kdy se dvě přímky protínají a naše kritérium by tedy nebylo splněno.
Ačkoliv dříve napsané kritérium je neadekvátní jako pouhá representace vektoru,s nepatrnou modifikací může být použita. Modifikací je zařídit maximální hloubku QuadTree. Jakmile je jednou hloubka stromu dosažena v procesu konstruováni, a pokud je kritérium stále nedostačující, pak je toto pole jednoduše representováno pixelem. Výsledkem tohoto přístupu je smíšený rastrově - vektorový výstup, ve kterým mohou být některé informace o obrazu ztraceny. Tento výsledek je znám jako „edge QuadTree“[2] (hranový QuadTree).

Datová struktura OctTree je analogická struktuře QuadTree s tím, že reprezentuje data třírozměrného prostoru. Prostor je zde rozdělován do krychlí (případně kvádrů), které jsou reprezentovány svým objemem. O OctTree pojednávají více následující kapitoly.
V následujícím považujme QuadTree a OctTree zkonstruované ze dvou odlišných listových kriterií. Pro rastrová data používáme QuadTree, nebo OctTree postavená na kriteriu, že žádny prostor nemůže obsahovat data mající více, než jednu barvu. Toto pracuje pro rastrová data, protože rastrová mřížka je postavená na regionech (pixelech) jednotlivých barev.Pro vektorová

data používáme kritérium, které jsme zmiňovali výše. Tímto získáme mnoho matematických vlastností, které jsou velmi užitečné.

2.2 - Datová struktura OctTree

Rozklad geometrie pro rozhodování viditelnosti a detekce kolizí jsou problémy, které řeší vývojáři téměř v každé aplikaci s 3D enginem. Je mnoho různých datových struktur a přístupů. Jak dané problémy vyřešit. Většina těchto způsobů je omezená na konkrétní požadavky tvůrců třírozměrné grafiky. Nicméně OctTree je jednoduchá datová struktura, která je používána na rozklad prostoru v jakékoliv formě.

Následující podkapitoly se zabývají kroky, které potřebujeme k vytvoření OctTree ze vstupní množiny polygonů v závislosti na rozkladu podle geometrie. OctTree je nejvhodnější pro tvorbu statické scény, ale může být také používán na ukládání objektů pohybujících se ve scéně, nebo na řešení viditelnosti či správu objektů[1].

2.2.1 - Základní popis

Nejjednodušeji řečeno OctTree je jenom strom s maximálním počtem osmi následníků v každém uzlu. To vytváří ideální strukturu na representaci třírozměrného světa rozděleného do krychlí. Kořen stromu obsahuje geometrii celého světa. Ten je potom rozdělen na osm stejných krychlí a jejich objem je vložen do jeho následníků. Tak to pokračuje s každým uzlem dále. Dělení prostoru trvá dokud není dosaženo uživatelem definovaných mezí. Typicky dokud nemají určitou velikost, nebo neobsahují daný maximální, nebo minimální počet objektů na uzel (záleží na tom, co od OctTree v konkrétním případě požadujeme).

BoundingCube (těleso vymezující objem krychle jednotlivého oktantu) každého uzlu je klíčem k používání OctTree na dělení prostoru. Každý uzel obsahuje ukazatele na všechny objekty v něm obsažené (které leží v jeho objemu). S těmito informacemi se můžeme podívat na sílu této datové struktury. Pro výpočty viditelnosti, je strom procházen od kořenu a objekty v něm obsažené jsou testovány proti pohledu uživatele do scény. Díky struktuře OctTree je to velmi snadné. Pokud jsou plně viditelné, je zobrazena celá jejich geometrie. Pokud jsou jen částečně viditelné, pokračuje procházení stromu přes jeho následníky. Pokud je uzel (a všichni jeho následníci) kompletně mimo pohled do scény, tak může procházení skončit. Toto je jen jeden z mnoha příkladů využití OctTree. Některé z dalších budou prezentovány v dalším textu práce.

2.2.2 - Datový model OctTree

Dělení geometrie používané v OctTree je typicky krok, který je používaný ve fázi před samotným během aplikace. Některé nástroje vezmou vstupní množinu geometrie a vytvoří z nich data používaná OctTree jako výstup, která mohou být potom používaná v aplikacích běžících v reálném čase. Jako minimum musí každý uzel obsahovat následující data:

Bounding Cube- Krychle, která vymezuje objem uzlu, ve kterém se nacházíme.

Geometry List - Každý uzel obsahuje množství polygonů, které v něm musejí být nějakým způsobem uloženy (v mém projektu je to například std::vector<OctreeObjects*> OctreeObjekty;).

Následník - Každý uzel má až osm následníků a na každý z nich musí být ukazatele.

Sousedi - Každý uzel má šest sousedů. Zde je možných několik různých přístupů. Například pro některé algoritmy není nutné mít tyto ukazatele uložené a v některých se ukládá pouze ukazatel na otce a z něho se zpět vrací mezi následníky.

3. Kapitola - Programové prostředí

3.1 - Úvod

Tento diplomový projekt jsem vypracoval v Microsoft Visual C++.NET s použitím knihoven Coin3D (http://www.coin3d.org/ [4]), což je sada knihoven používaných pro tvorbu 3D grafických aplikací vystavěných nad OpenGl. Jádro těchto knihoven je nezávislé na platformě, na které běží. Může tedy běžet jak v prostředí Microsoft Windows, Mac OS X, GNU/Linux, SGI IRIX nebo na jiných platformách UNIXu. Ke svému běhu potřebuje pouze překladač C++ jazyka a knihovnu OpenGl (nebo jinou knihovnu, která implementuje OpenGl API, jako je například Mesa). Grafické uživatelské prostředí může být od běžného prostředí Microsoft Windows přes prostředí Trollech's QT / Motif X windows a nebo Mac OS X.

Aplikační a uživatelské prostředí knihoven z Coin3D je plně kompatibilní s SGI Open Inventorem (http://oss.sgi.com/projects/inventor/), což je v podstatě standardní grafické API pro komplexní vizualizaci 3D od firmy SGI (http://www.sgi.com/ [7]).

3.2 - Historie Coin3D

Začátky Coin3D sahají až do roku 1995, kdy vývojáři software firmy System in Motion vytvořili 3D grafickou knihovnu, která byla určena pro soubory formátu VRML 1.0. Tato knihovna byla určena převážně pro zobrazování těchto souborů. Během let rozšiřování, optimalizací vyvstala potřeba vše seriozně přepracovat a vytvořit i lepší design. Postupně tak vznikla knihovna, kterou autoři pojmenovali Coin3D.

Firma System in Motion (http://www.sim.no/ [3]) je v dnešní době světový producent real-timové grafiky (grafiky v reálném čase). Mezi zákazníky této firmy patří například Kawasaki, Mitsubishi, Boing, nebo Bmw. Tato firma byla založena v roce 1994 v Norsku ve městě Trondheim v úzké spolupráci s Norskou univerzitou Science and Technology. V roce 1997 si SiM (System in Motion) otevřela první kancelář v Oslu a během roku 2000 již vydala Coin 1.0. a v roce 2003 Coin 2.0.

3.3 - Licence Coin3D

Knihovna Coin3D je implementována v jazyce C++ a šířena pod třemi různými licencemi. Kromě verze Coin Free Edition, která je použita v mém projektu, jsou to licence Coin Evaluation Edition a Coin Professional Edition.

Coin Professional Edition: pro použití v komerčních nebo jinak ziskových aplikacích.
Coin Evaluation Edition: pro zhodnocení Coin3D (potřebuje registraci na stránkách SIM).
Coin Free Edition: je pod GPL pro volný vývoj software.

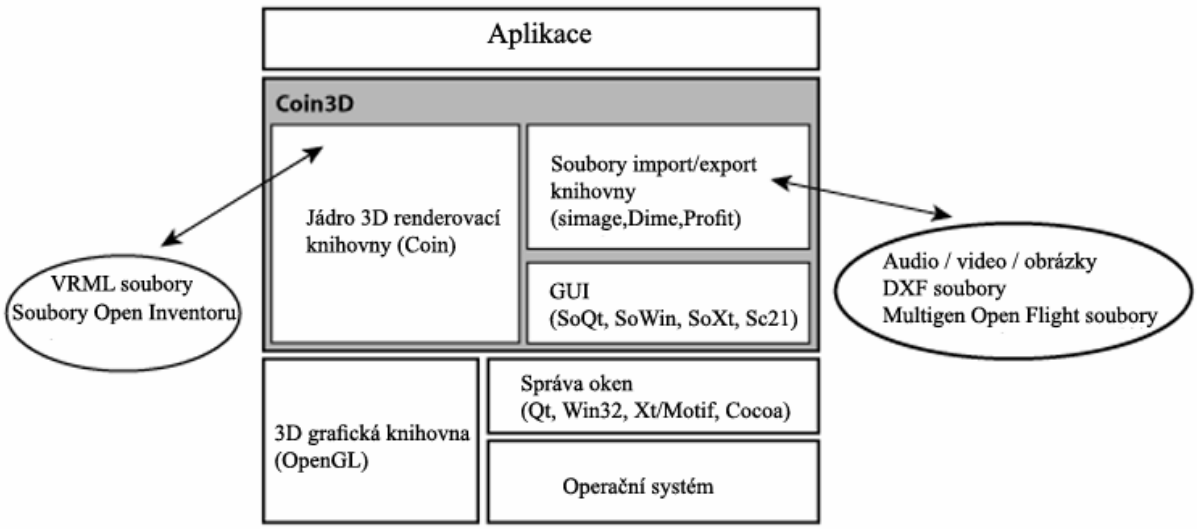
Více o licencích se lze dozvědět na webových stránkách firmy System in Motion[3]
na http://www.sim.no/products/Coin3D/licensing/

3.4 - Stručný popis Coin3D

Coin3D je knihovna postavená nad OpenGL a používá datové struktury grafu scény na zobrazování třírozměrné grafiky v reálném čase. Základní vkládání, zobrazování a interakce s třírozměrnými objekty je implementována velmi malým počtem řádků kódu a programová efektivita je navíc výrazně zvýšena tím, že pracuje přímo s OpenGL.

Jádro knihovny Coin3D používá k ukládání dat k zobrazení graf scény. Tento datově řízený design (scéna se překresluje pouze v případě, pokud jsou její data změněna) dělá Coin3D velmi vhodný pro použití v aplikacích založených na interface (například vědecké a inženýrské aplikace), které nemají přehnané požadavky na CPU.

Na tomto obrázku je znázorněno, jak jsou knihovny SoWin a SoQt začleněny do systému.



Obrázek 4

Coin3D je pouze knihovna, která je určená k zobrazení scény. Proto potřebuje navíc knihovnu pro práci v uživatelském prostředí. K tomu slouží následující knihovny (jsou roztříděné podle platforem).

SoQt je pro zobrazení v Trolltech's přes platformu Qt toolkit (UNIX, Windows, Mac OS X).
SoWin je pro zobrazení ve Win32 API na platformách Microsoft Windows.
SoXt je pro zobrazení v Xt/Motif na X Windows.
SoGtk je pro zobrazování GTK+.

Další důležitou knihovnou je SoImage, která slouží k nahrávání, ukládání a manipulacím s obrázky a videem. Mezi podporované formáty patří :

- AVI, MPEG
- JPEG (čtení a zápis přes libjpeg)
- PNG (čtení a zápis přes libpng a zlib)
- GIF, TIFF
- RGB (pouze načítání)
- PIC (pouze načítání)
- TGA (pouze načítání)
- EPS (pouze zápis)

4. Kapitola - Zobrazení scény

4.1 - Základní struktura

V Coin3D se scéna tvoří jako graf, do kterého postupně přidáváme uzly. Tento graf mívá stromovou strukturu. Nelistové uzly slouží k uspořádání scény do hierarchických struktur. Každý z listových uzlů nese informaci o grafické operaci, kterou budeme provádět. Což může být nastavení světla, posunutí souřadnic nebo vytvoření některého z grafických primitiv (např. SoCube - hranol, SoCylinder - válec, SoSphere - koule, SoCone - kužel a podobně). Tento strom scény je potom vložen do renderovací smyčky jádra Coin3D. Coin3D vezme tento strom a vykreslí jej podle pořadí vložených uzlů / listů. V dalších iteracích se přepočítávají pouze ty části stromu scény, které byly změněny, čímž se celá aplikace rapidně zrychlí.

4.2 - Tvorba grafu scény

Ve svém diplomovém projektu tvořím scénu s jedním hlavním uzlem, který jsem nazval root.

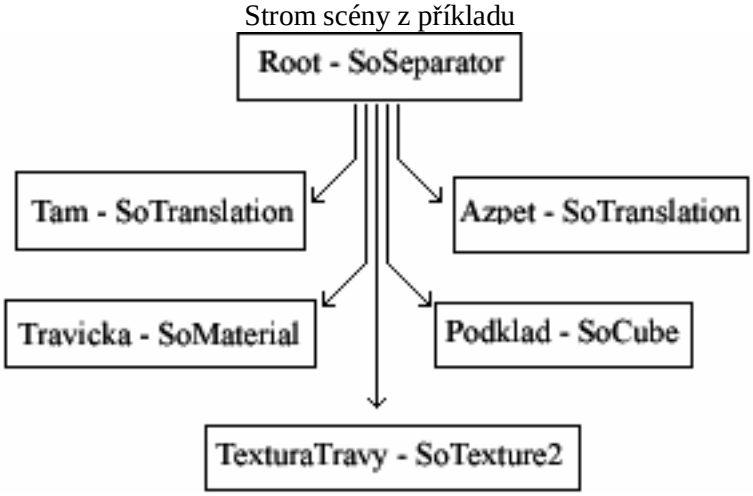
```
SoSeparator *root = new SoSeparator;
root->ref();
```

Na tento kořen postupně navazuji listy a uzly. Uzly jsou všechny typu SoSeparator a určují jednotlivé objekty jako jsou dům, silnice nebo křižovatka. Výjimkou jsou podklad, světla a kamera, které jsou navázány přímo na hlavní kořen root. Mezi listy patří například: posunutí (SoTranslation), tvorba hranolu (SoCube), tvorba bodového světla - slunce (SoPointLight), tvorba všudypřítomného světla - ambientního (SoEnvironment), nastavení materiálu - barva, odrazy světla, lesk (SoMaterial), nastavení textury - (SoTexture2) a dále SoCoordinate3 s SoIndexedTriangleStripSet k tvorbě trojúhelníkové sítě jednotlivých domů, SoPerspectiveCamera - perspektivní kamera, SoSensor - k sledování časových událostí a ještě několik dalších.

V následujícím textu, nebude-li řečeno jinak, budeme mít pod pojmem „objekt scény“ na mysli sekvenci listů náležící pod jednotlivé SoSeparator (například pro silnici je to sekvence: SoMaterial, SoTexture, SoCube / eventuelně síť trojúhelníků na dům). Pro jednoznačnost při vyjadřování souřadnic je nutné pro každý objekt začínat s translací souřadnic z bodu [0,0]. Tímto bodem pro nás bude výchozí bod [0,0] v Coin3D. Do tohoto bodu se tedy po každé translaci vrátíme translací v opačném směru. Jako demonstrativní příklad uvádím zobrazení podkladu a příslušející část stromu scény (viz obrázek 5).

```
// Nejprve se posuneme doprostřed naší scény (velikost scény pro ukázkové účely je 15).
SoTranslation *tam = new SoTranslation;
tam->translation.setValue(7.5f,0.05f,-7.5f);
root->addChild(tam);

// Materiál podkladu
SoMaterial *travicka = new SoMaterial;
travicka->ambientColor.setValue(SbColor(0.1f, 0.6f, 0.1f));
travicka->diffuseColor.setValue(SbColor(0.1f, 0.6f, 0.1f));
root->addChild(travicka);

// Textura podkladu
SoTexture2 *texturatravy = new SoTexture2;
texturatravy->filename.setValue("Trava.jpg");
root->addChild(texturatravy);

// Objekt krychle
SoCube *podklad = new SoCube;
podklad->width.setValue(float(sirka));
podklad->height.setValue(0.1f);
podklad->depth.setValue(float(delka));

// Posun zpátky na souřadnice [0,0]
root->addChild(podklad);
SoTranslation *azpet = new SoTranslation;
azpet->translation.setValue(-7.5f,-0.05f,-7.5f);
root->addChild(azpet);
```

Strom scény z příkladu

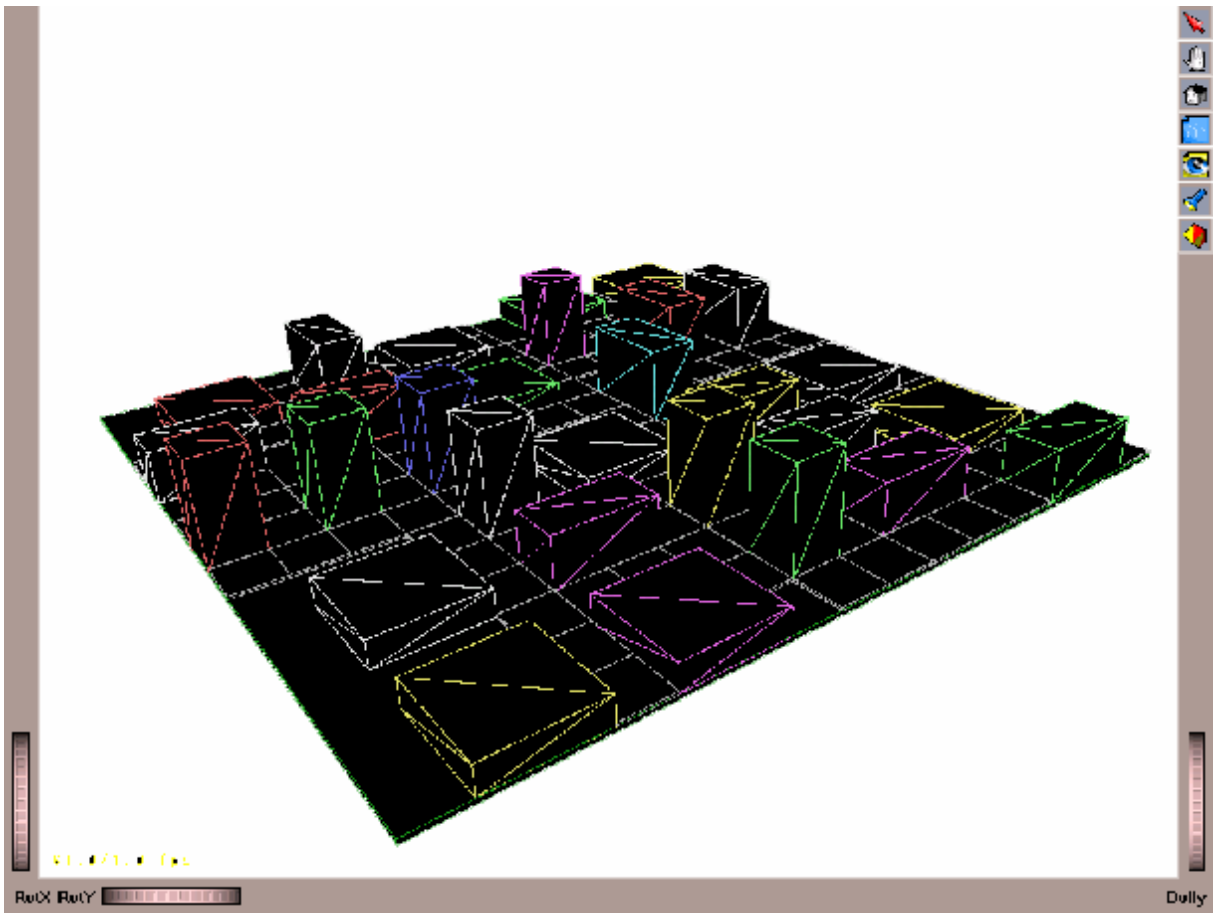


Obrázek 5.

4.3 - Tvorba budov

Pro všechny potřebné objekty v našem generátoru města nám prozatím stačí objekty SoCube. Jedinou výjimkou je budova. Od ostatních se totiž liší tím, že zde potřebujeme texturovat každou stěnu jinou texturou, což při použití SoCube nelze, protože všechny její stěny jsou texturovány stejně. Musíme si tedy pomoci jinak. Řešením je použití trojúhelníkové sítě. V OpenGL můžeme použít několik způsobů, jak renderovat tyto trojúhelníky. Nejjednodušší způsob je použití GlTriangles, kde specifikujeme vždy tři body. Každý z těchto bodů (vertexů) se skládá ze třech prostorových souřadnic x,y,z. Tyto tři vertexy předáme OpenGl, které nám zobrazí jeden trojúhelník. Většina trojúhelníků však sdílí vrcholy se svými sousedy. Proto byly vymyšleny struktury jako GlTriangleStrip, který vždy z prvních třech vertexů vykreslí první trojúhelník, ale od té chvíle již vykresluje trojúhelník s každým dalším vertexem, přičemž chybějící dva vertexy použije z předchozího trojúhelníku. Ve svém projektu jsem tedy k účelu zobrazení budov použil SoIndexedTriangleStripSet a SoCoordinate3. Každý dům je tedy sestaven ze sítě trojúhelníků, jak ukazuje obrázek 6. Každou ze stěn je třeba otexturovat zvlášť. Pouze „dno“ domu texturovat nebudeme, protože je to zbytečné. K tvorbě stěn domu by dále mohlo být použito normalizačních vektorů, aby nemusely být počítány při úvodním generování scény. V programové příloze následuje část příslušného popsaného kódu (část kódu č.1).

Ukázka trojúhelníkové a čtvercové sítě



Obrázek 6.

4.4 - Tvorba budov - načítání ze souboru

Pokud se v našem programu chceme posunout dále k obecnějším strukturám, je potřeba tvorbu budov udělat obecnější. V principu se jedná o to, že generátoru je zadáno pouze jméno souboru budovy a její rozměry. Program si ze souboru načte jména textur pro jednotlivé stěny a rozměry budovy. Podle těchto parametrů potom vytvoří potřebnou budovu (viz. předchozí kapitola). Zde by měla být nasnadě otázka, proč jsou rozměry budov jednou zadávány generátoru a podruhé si je program čte ze souboru. Čtení těchto parametrů ze souboru je přidáno hlavně proto, že jejich zadávání do generátoru v budoucnu odpadne (toto zadávání je vynuceno hierarchií tříd a je nutné pro fázi generování města do pole - viz. 7. kapitola) a bylo by zbytečné v budoucnosti předělávat všechny soubory.

4.5 - Tvorba textur

Ke zvýšení realističnosti modelu přispívají velkou měrou textury. Všechny textury použité v mé práci jsou vlastnoručně vytvořené v 3ds max 6. Nejprve jsem si vytvořil třírozměrné modely oken a dveří, ze kterých jsem následným „nafocením“ získal velmi dobře plasticky vypadající dvourozměrné stavební bloky. Každá budova se může skládat z různého množství těchto základních stavebních bloků. Tím jsem vytvořil stěny různých budov v potřebných velikostech. Tyto stěny jsem opět „nafotil“ a tím získal výsledné textury, které používám v projektu. Mají plastický vzhled a přitom jsou to obyčejné bitmapy, které nesnižují výkon oproti plastickým modelům. Jednotlivé obrázky znázorňující tvorbu textur jsou v obrazové příloze.

4.6 - Efektivita texturování

Nanášení textur na povrchy popsané v předchozí podkapitole s sebou nese zápory v podobě zvýšené náročnosti na vytížení jak paměti počítače, tak procesoru. Problém tví v tom, že pro každou stěnu je načítána textura z disku a přidána jako další list do stromu scény. Každá textura navíc tedy zabírá další místo. Tento problém lze vyřešit tak, že zavedeme virtuální odkazy na textury. Tím bude každá textura v paměti obsažena pouze jednou a zbytek budou tvořit virtuální odkazy. V praxi existují dva způsoby, jak toho dosáhnout. Prvním z nich je hlídání používání textur při samotném generování a údržba s tím spojená. Druhým je vytvoření podprogramu, který vezme výslednou scénu a vícenásobná použití textur převede na virtuální odkazy. K této variantě jsem se také přiklonil ve svém projektu. Navíc jsem se zde nechal inspirovat radami vývojářů Coin3D http://www.coin3d.org/, kteří podobné řešení doporučují při zefektivňování ukládání VRML modelů na disk. Toto řešení přineslo aplikaci vyšší zrychlení, než jsem předpokládal. Na mém počítači (AMD Sempron2200+, 1024mb RAM, Radeon9600) trvalo spouštění aplikace při scéně 30x30 v rozmezí od 110 do 120 sekund, které se snížilo na 11-14 sekund. Také využití paměti se snížilo o přibližně 300mb. Příslušný programový kód uvádím v programové příloze část druhá. Tento kód jsem upravil tak, že lze využít v každé aplikaci využívající knihovnu Coin3D a v ní provede příslušné zefektivnění.

4.7 - Pozadí scény

K vytvoření panoramatického prostředí jsem zvolil nejjednodušší variantu, která je shodou okolností také velmi efektivní výpočetně. Celá scéna je umístěna do obrovské krychle (skyBox), jejíž strany jsou texturovány tak, že textury jednotlivých stran na sebe navazují. Stejně tak na ně musí barevně navazovat „střešní“ textura. K jednotlivému smazání rozdílů na stěnových panoramatických fotkách lze použít jakýkoliv program s funkcí rozmazávání a s trochu zručnosti nepotřebujeme žádné drahé vybavení nebo speciální software k dosažení kýženého výsledku.

4.8 - Modely dynamických objektů

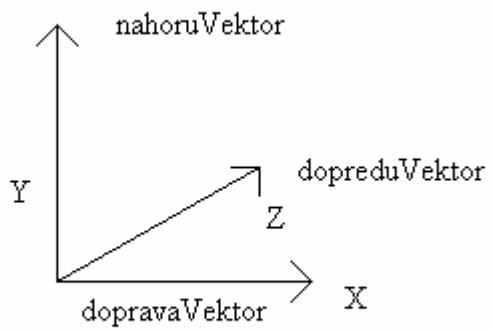
Modely použité v mém diplomovém projektu jsem vytvořil v 3ds max 6. Cílem bylo vytvořit jednoduché modely, které budou „poletovat“ ve scéně a budou vhodně doplňovat virtuální město. Zvolil jsem tedy jednoduché modely připomínající všeobecně známou verzi ufo, které mají další výhodu, že jsou středově symetrické a mohou se tedy pohybovat jakýmkoliv směrem, aniž bychom museli řešit jejich rotaci (aby byl letící objekt otočen tím směrem, kterým letí). Model je vytvořen z prstence (torus) a koule a celý je potom smrštěn na čtvrtinu své výšky. Následně je exportován do souboru formátu VRML (podporovaný formát v Coin3D). Nakonec jsem výsledný VRML soubor ručně upravil, aby byl model umístěn v nulových souřadnicích. Hotový model lze vidět na obrázku XX v obrazové příloze.

5. Kapitola - Prohlížení scény

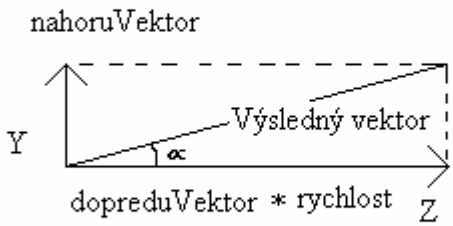
5.1 - Práce s kamerou

Pokud chceme ve scéně používat neomezeného pohybu kamery v libovolných směrech, potřebujeme znát její aktuální směr pohledu a nejen to, ale také její orientaci (otočení kolem vektoru pohledu). Tímto způsobem je též v Coin3D implementována perspektivní kamera - SoPerspectiveCamera. K programem ovládané kameře ovšem potřebujeme více, pokud se chceme vyhnout složitým a mnohdy až nesmyslným výpočtům. K přesnému určení pozice uživatele ve scéně si tedy musíme udržovat hodnoty dvou vektorů a bod, v němž se právě nacházíme. Z praktického hlediska byl přidán navíc třetí vektor (dopravaVektor) - obrázek 7.

Obrázek 7 - Zobrazení vektorů pohledu kamery



Obrázek 8 - Výpočet pootočení kamery



K samotnému výpočtu požadovaného otočení nám stačí používat pouhých jednoduchých funkcí násobení skalárem a součtu vektorů. Každý vektor na konci výpočtů je navíc normalizován (funkce normalize je v Coin3D implementována, jinak by se dal použít jednoduchý výpočet pomocí odmocniny). Princip otočení kamery je tedy takový, že pokud chceme například otočit kameru směrem nahoru, vezmeme nahoru vektor a dopředu vektor vynásobený konstantou (tato konstanta určuje výsledný úhel otočení - čím vyšší číslo, tím bude menší úhel natočení). Vyjde nám vektor směřující kolmo vzhůru ve směru pohledu kamery - viz. obrázek 8. Výsledný vektor poté normalizujeme. Podobně samozřejmě musíme upravit i další dva vektory - v závislosti na směru otáčení. Praktická ukázka je opět v programové příloze - část třetí.

5.2 - Základní fyzika a letecký simulátor

Pro začátek jsem implementoval pouze jednoduchou fyziku. Na objekt tedy působí síla gravitační, která je reprezentována gravitačním vektorem. Síla pohonu - síla působící při zapnutém pohonu ve směru dopředu vektoru. Poslední silou podílející se na výpočtu je síla odporu vzduchu, která je reprezentována koeficientem, kterým se zkracuje výsledný rychlostní vektor.

```
// odpor vzduchu
rychlostniVektor=rychlostniVektor*0.97f;
// rychlost se zvyšuje ve směru pohledu kamery
rychlostniVektor=rychlostniVektor+dopreduVektor*zrychleni;
// započítání vlivu gravitace
rychlostniVektor=rychlostniVektor+vektorGravitace;
```

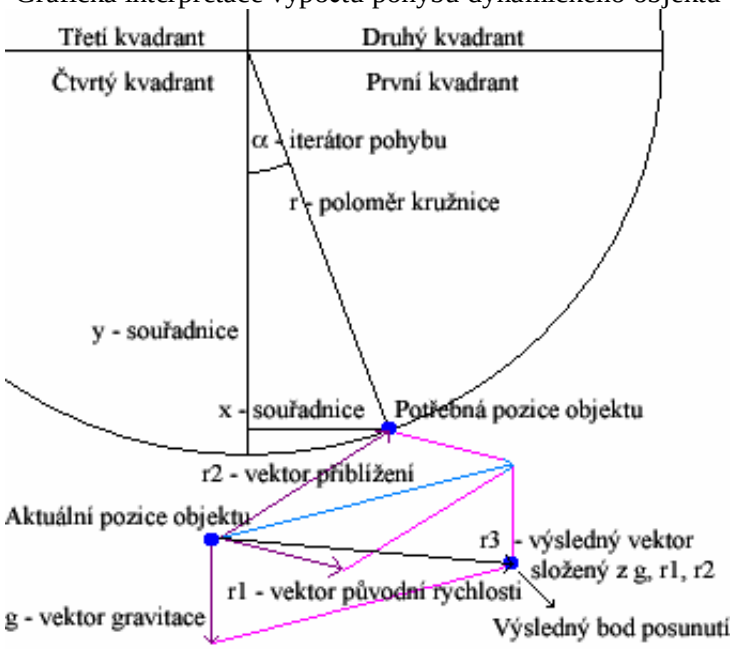
V módu simulátoru lze také přistát na zemi a projíždět se pomocí rychlosti a zatáčení vlevo a vpravo podobně jako s vozidlem (pouze v případě, že je vypnuta detekce kolizí se silnicí).

5.3 - Fyzika a pohyb dynamických objektů

Pohyb dynamického objektu ve scéně je stejně jako u kamery dán rychlostí, směrem pohybu, gravitací a odporem vzduchu. Na rozdíl od kamery ovšem pohyb dynamických objektů není určován uživatelem, ale výpočtem. Ideou bylo, aby se objekt nepohyboval pouze po primitivní dráze (přímočaře, do čtverce atd…) nebo po jakékoliv jiné - předem přesně určené. Výsledkem je, že objekt má danou dráhu, ke které se „snaží“ přiblížit, ale jakékoliv okolní vlivy můžou tuto snahu lehce ovlivňovat.

Pro snadnou implementaci jsem vybral kruhovou dráhu, ale v budoucnu by se dala například na stejném principu udělat dráha z checkpointů načítaných ze souboru (pro demonstrační účely je ovšem stávající stav zcela dostačující). Každý objekt má svůj střed oběhu a iterátor otočení (úhel určující pozici na kružnici, které se v daném okamžiku snaží objekt dosáhnout). Výpočet a schématický nákres jsou pro názornost uvedeny v dvourozměrném prostředí.

Grafická interpretace výpočtu pohybu dynamického objektu



Obrázek 9.

Souřadnice potřebné pozice objektu jsou [x,y]. Visual C++ počítá úhly v radiánech, proto je π děleno iterátorem pohybu, který je ve stupních.

$$x = (\cos(\pi / \text{IterátorPohybu})) * r$$

$$y = \sqrt{r^2 * x^2}$$

Následuje úprava souřadnic x,y podle kvadrantu. Například pro druhý kvadrant bude x zachováno a y se bude rovnat - y. Odtud získáme vektor přiblížení r_2.
([s_x , s_y] + [x, y]) - [a_x, a_y], kde [s_x , s_y] jsou souřadnice středu obletu a [a_x, a_y] jsou souřadnice aktuální pozice objektu. Vektor r_2 je před finální změnou rychlosti upraven pomocí „dorovnání“, které zohledňuje aktuální náročnost scény na výkon počítače. Kdyby tomu tak nebylo, tak by ve velké scéně s nízkým počtem fps byl let velmi pomalý a v malých scénách by byl naopak velmi rychlý. Díky „dorovnání“ je vektor r_2 o tento poměrný rozdíl zkrácen nebo prodloužen. Výpočet finálního vektoru posunutí je pouhé složení všech tří vektorů $r_3 = r_1 + r_2 + g$.

Po spočtení nových souřadnic se dynamický objekt vloží do OctTree funkcí KorenOctree->PridejPrvekAtestujKolize(*PokusnyObjektOctree), která navíc testuje, jestli nenastává kolize (více o detekci kolizí v osmé kapitole). Pokud kolize nenastává, aktualizují se všechny zbývající hodnoty na novou pozici a program pokračuje s výpočtem dalšího objektu. V případě, že by nastala kolize, se z OctTree PokusnyObjektOctree odebere a vrátí se tam jeho původní verze. K tomu se vynulují vektory jeho rychlosti. Takový objekt také z demonstračních důvodů „přijde o motor“ a v následných výpočtech jeho pohybu se započítává pouze gravitace a odpor vzduchu, objekt tedy spadne na zem nebo na budovu (podle jeho pozice).

5.4 - Ovládání

U ovládání pohybu kamery jsem se nechal inspirovat staršími leteckými simulátory. Ovládání je naprogramováno v reakcích klávesy v UrbanGenerator.cpp a klávesy na které bude

program reagovat jsou implementovány v Klávesnice.h. na Pohyb kamery je ovládán klávesami šipek a klávesami a,z, q, w, e, r. Šipka nahoru pohybuje kamerou směrem shora dolů, šipka dolů pohybuje kamerou směrem zdola nahoru (slouží k věrnějšímu simulování letu). Šipky doleva a doprava slouží k rotaci kamery kolem dopreduVektoru (naklánění letadla). Klávesy q, w jsou k pohybu kamery doleva a doprava. Konečně klávesa e slouží k zapínání fyzikálního modelu a detekce kolizí. Klávesou r se vrátíme zpět do prohlížecího módu.

6. Kapitola - Základní struktura programu

6.1 - Soubory

Projekt je sestaven z několika hlavních (programových) souborů, makefile, modelů VRML, textových souborů, v kterých jsou uloženy charakteristiky budov, a obrázků formátu *.jpg potřebných pro texturování. Tyto obrázky by měly být umístěny ve správném adresáři, jinak nám bude SoImage hlásit každý neúspěšný pokus při použití textury. Samozřejmě, že by nakonec výsledná scéna byla zobrazena (bez textur), ale jen při 10 budovách je voláno texturování sedmdesátkrát, a proto nepředpokládám, že by si někdo chtěl dát práci s „odklikáváním“ tolika chybových hlášení.

Hlavní soubor se jmenuje UrbanGenerator.cpp a obsahuje kostru programu, obsluhu klávesnice a fyzikální model s detekcí kolizí. Generátor.h slouží jak k základnímu nastavení proměnných a vygenerování kořene scény či světel, tak i pro samotné generování scény. Jak statických, tak i dynamických objektů (k tomu se vrátíme později). Dalšími soubory jsou Pole.h a Pole.cpp. Zde jsou metody na převedení vygenerovaného pole na pole odkazů na objekty (PridejObjekt) a zobrazení (Zobraz). Další dvojicí je Gobjects.h a Gobjects.cpp. Zde jsou obsaženy třídy GObject, Silnice, Krizovatka a Barak. Hlavní metodou je Vytvor, která vytvoří SoSeparator a pod něj naskládá příslušné operace. Terrain.h a Terrain.cpp obsahují tvorbu SkyBoxu (terénu scény). V Fobjects.h je třída pro tvorbu a správu dynamických objektů (o jejich tvorbě blíže v podkapitole 7.4). OctreeObject.h slouží pro správu objektů vkládaných do Octree. OcttreeNodes.h představuje třídu uzlů Octtree (o Octtree se podrobněji rozepisuje kapitola 8). Soubory SoPerfGraph.h a SoPerfGraph.cpp jsou public domain soubory, jejichž autorem je Jan Pečiva a slouží k zobrazení grafů výkonu.

6.2 - Cesta k zobrazení

Město je generováno do pole int plocha[sirka+2][delka+2], kde sirka a delka jsou konstanty, které určují rozměry území, do kterého budeme generovat. Z tohoto pole pak není problém získat informace, které potřebujeme k použití metody Pole.PridejObjekt (i-1,i2-1, Objekt, true); Tato metoda přidá objekt do speciálního pole odkazů na objekty (viz. programová příloha část 4).

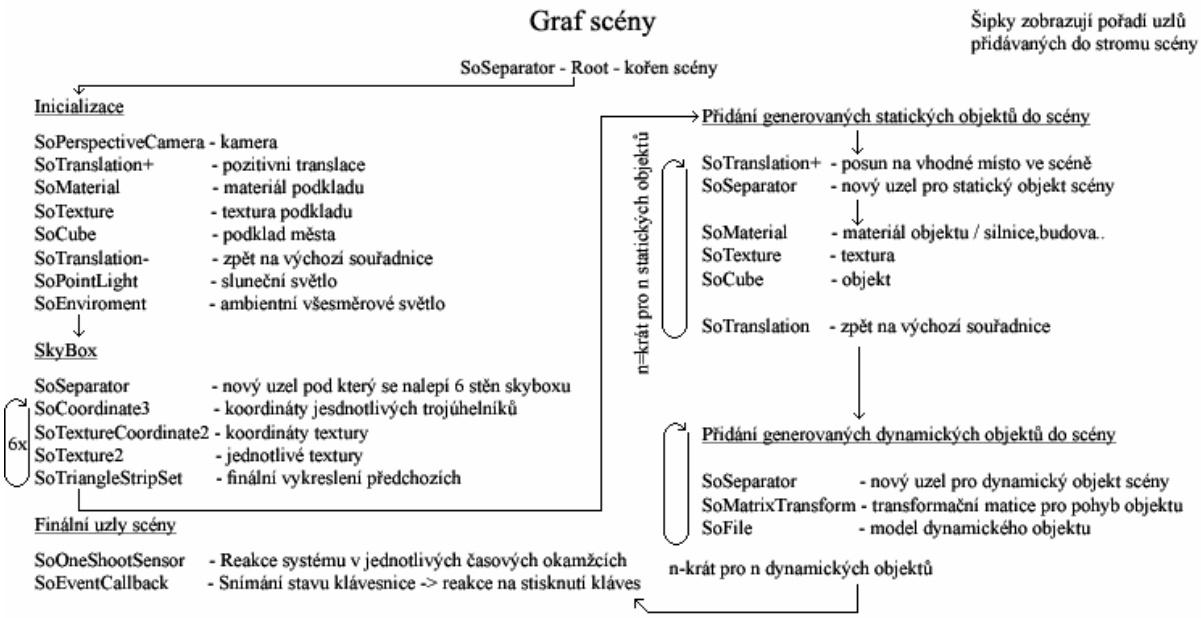
Objekt, který přidáváme, je již vytvořená specifikace objektu, který budeme vkládat. Například pokud na ploše na pozici pozicex, pozicey máme křižovatku, pak ji vytvoříme takto:
Kriz=new Krizovatka(pozicex,0,pozicey,1,0.05f,1,true,SbColor(0.7,0.7,0.7),"Asfalt1.jpg");

a potom ji do pole odkazů přidáme příkazem: Pole.PridejObjekt (i-1,i2-1, Kriz , true);

V okamžiku, kdy jsme do Pole přidali všechny objekty, můžeme zavolat metodu Pole.Zobraz(root); Až v této chvíli začíná opravdová práce s knihovnami Coin3D. Do této chvíle jsme do rootu přidali jen světla a podklad. Nyní program začíná procházet pole odkazů na objekty. Podle jejich souřadnic a jejich velikosti umístí objekt na jeho příslušné místo ve scéně (pozitivní translace). Poté je zavolána metoda vytvor, která vytvoří nový uzel SoSeparator a na něj naváže příslušné operace (materiál, textura, hranol nebo trojúhelníky). Potom je proveden návrat do bodu [0,0] pomocí zpětné translace.

Po vytvoření celé scény města jsou generovány dynamické objekty. O nich budeme blíže hovořit v kapitole 8. Princip jejich přidávání do grafu scény je podobný jako při přidávání statických objektů. Pouze s tím rozdílem, že pozice objektu není určována pomocí translace, ale pomocí SbMatrix a SbMatrixTransform. Tento přístup je u dynamických objektů výhodnější, protože pro změnu pozice objektu ve scéně stačí provést jednoduchou změnu transformační matice a Coin3D se už postará o přesunutí objektu. Procedura Zobraz, která přidává statické objekty z pole do grafu scény je v programové příloze (část kódu 4). Tvorba celého grafu scény je znázorněna na obrázku 10.

Znázorňuje výsledný graf scény a pořadí vkládání obrázků



Obrázek 10.

Poslední z operací, které je potřeba udělat, aby se nám zobrazila výsledná scéna, je vytvoření okna prohlížeče. Tomuto oknu zadáme kořen naší scény jako vstupní proměnnou. Potom již stačí zavést renderovací smyčku. Samozřejmě, že tyto operace jsou platformově závislé (viz. úvodní popis knihoven), proto opět musíme použít konstrukce #idef #else #endif. Další nám již zařídí knihovny z Coin3D, které poskytují potřebné zázemí. V mém projektu je použita SoWinRenderArea nebo SoQtRenderArea (v závislosti na platformě), která se nejlépe hodí pro práci s kamerou. Coin3D navíc například poskytuje SoWinExaminerViewer (SoWinQtViewer), který slouží prohlížení 3D modelů, nebo SoWinPlaneViewer (SoQtPlaneViewer) sloužící pro prohlížení modelu pohyblivou kamerou v ortogonálních rovinách. Jednotlivé ukázky prohlížečů jsou v obrazové příloze. Více o prohlížečích scény lze najít například na http://doc.coin3d.org/SoWin/classSoWinFullViewer.html. Příslušný kód sloužící k zobrazení scény a zavedení prohlížeče je uveden v programové příloze část 6.

7. Kapitola - Generování

7.1 - Idea

Pro generování založené na náhodě bylo na počátku základní otázkou, který z prvků vzít jako určující. Tedy jako prvek, který bude zvolen náhodně a podle těchto náhodných prvků se bude řídit zbytek procesu generování. K dispozici máme hned tři logické jednotky. Je to silnice, křižovatka a dům. Poslední variantu zmíním až v závěru, protože její využitelnost se objevila až po dokončení programové části tohoto projektu.

Nejprve se moje úvahy ubíraly logickým směrem přes křižovatky, protože právě křižovatky určují tvář města, jako uzly, jež jsou spojeny tepnami silnic. Takto poetická byla zpočátku má představa. Později při snaze dovést tuto představu k reálnému řešení jsem narazil na příliš mnoho praktických překážek.

Další variantou je tedy orientace na silnice. Základní představa je poměrně jednoduchá a i její aplikace (jak se později ukáže) nese poměrně dobré výsledky. Jde o to, že se vygenerují silnice a v místech, kde se kříží, budou křižovatky. V místech kde zůstane volné místo, bude trávník nebo domy.

7.2 - Náhodné číslo

Základem pro vygenerování náhodného čísla jsou knihovny process.h a unistd.h. (#include <process.h>, #include <unistd>). Process.h i unistd.h jsou knihovny, které obsahují funkci getpid(); která nám vrátí id procesu, čímž v podstatě získáme semínko pro proceduru srand(int seed), čímž dostaneme novou řadu pseudonáhodných čísel. Pomocí funkce rand(); potom získáme číslo mezi 0 a 1 z této řady. Pro jednoduchost jsem toto číslo násobil stem, abych mohl počítat v procentech. Obě knihovny uvádím, protože knihovna process.h je pouze pro aplikace pod systémy Microsoft Windows, které samozřejmě nemají unistd. Stejná situace nastává pod Linuxem, případně Unixem, kde není knihovna process.h.

7.3 - Popis algoritmu generování

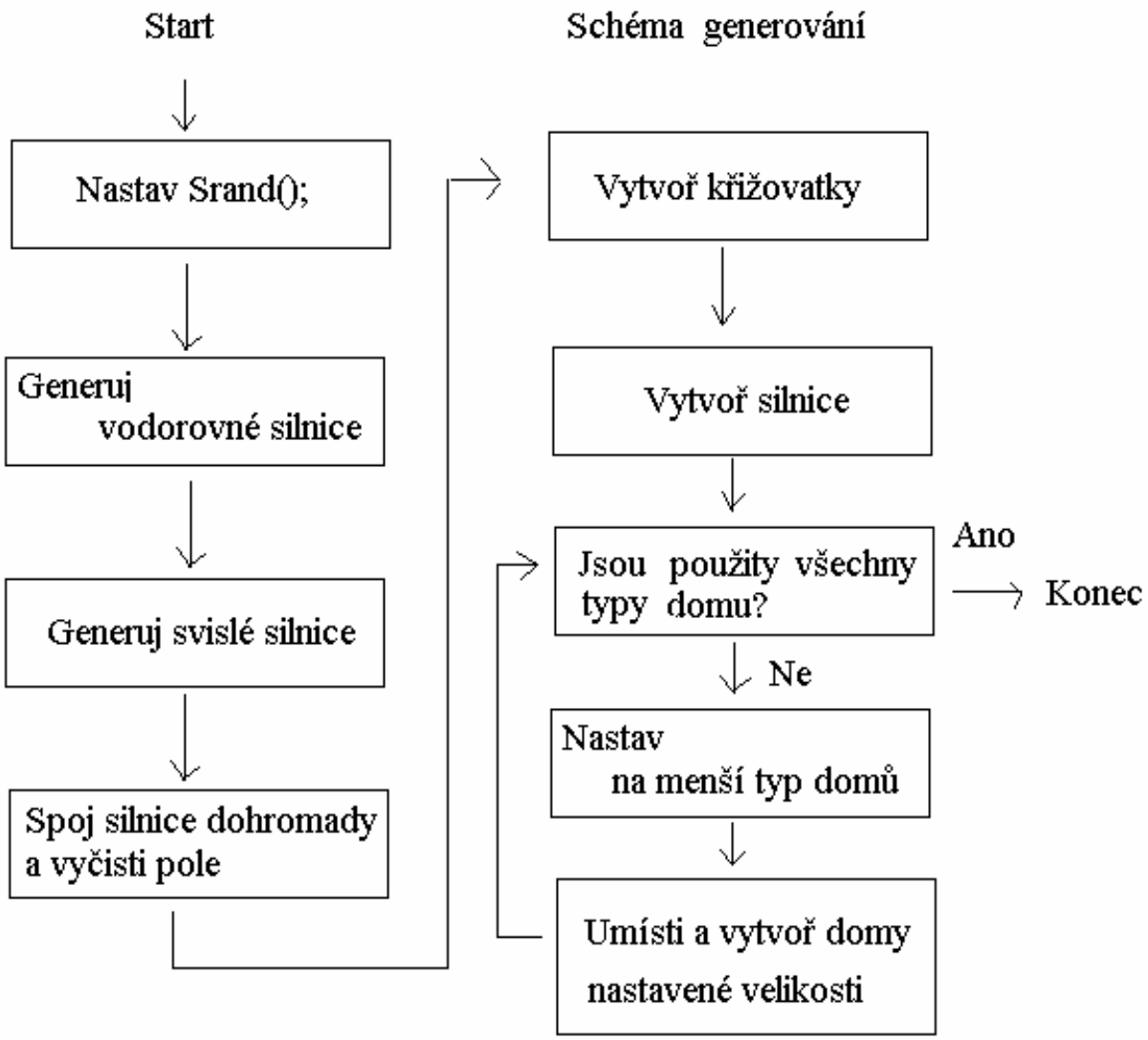
Nyní již víme, jak získat náhodná čísla, a můžeme začít uvažovat nad samotným algoritmem. Já jsem se rozhodl pro generování pomocí pravděpodobností. Stručně řečeno procházím pole po vodorovné ose a na každém novém políčku může začít silnice s jistou pravděpodobností (proto pravděpodobnosti, protože jednoduchým pohybem pravděpodobnostních vah lze ovlivňovat délku a četnost silnic). Když se rozhodnu, že silnice začne, vytvořím ihned silnici délky dvě, což jsem určil jako minimální délku. V každém dalším kroku je jistá pravděpodobnost, že silnice bude pokračovat. Pro příklad: budu-li mít 80% pravděpodobnost pokračování, potom pravděpodobnost, že silnice bude mít délku alespoň 2 pole je 100%, že bude mít délku alespoň 3 pole je 100*0.8 = 80%, že bude mít délku alespoň 3 pole je 100*0.8*0.8 = 64%, že bude mít délku alespoň 4 pole je 100*0.8*08*08 = 52% atd. Pokud by pravděpodobnost byla jen 50%, pak by se silnice s alespoň čtyřmi poli objevila jen v 12,5%.

Silnici ukládám do pole pod číslem jedna. Nuly jsou původní výplň pole. Kolem vygenerované silnice dávám příznaky (číslo 3), že na tomto místě silnice být nemůže. Tím se zabrání nelogičnostem typu dvě silnice přesně vedle sebe nebo „dotyky rohem“. Zde ještě jedna

malá poznámka: pole, do kterého generuji, je o jedno políčko do všech stran větší než budoucí pole objektů, jak kvůli testování možnosti položení silnice zde, tak i pro umisťování objektů v následující části generování.

Silnice kolmé na vodorovný směr jsou generovány stejným algoritmem nezávisle na předchozím generování. Zde by šel tento algoritmus zlepšit tím, že by byla mnohem větší pravděpodobnost, že silnice začne v místě, kde již probíhá silnice vodorovným směrem. Podobně by byla vyšší pravděpodobnost, že silnice skončí v místě křížení vodorovné silnice, než kdekoliv jinde.

Schématické zobrazení jednotlivých fází generování městské zástavby.



Obrázek 11.

V dalším kroku spojím oba vygenerované směry silnice do jednoho pole a odstraním příznakové „trojky“. Teď již mohu začít přidávat objekty silnic a křižovatek. Jednoduchým průchodem pro každé pole testuji na jedničku a dělám součet všech okolních políček. Pokud je součet větší než 3, je jasné, že mi v tomto bodě vznikla křižovatka, jinak je to silnice.

Nyní již mohu přistoupit k umisťování domů. Mám daný počet domů, u nichž vím jejich rozměry. Nejjednodušší způsob je zkoušet je od největšího k nejmenšímu, zda se vejdou. Pokud ano, pak vložím dům a označím jeho okolí, kam nepůjde dát další dům. Zde by šlo navíc umisťování domů ovlivňovat pravděpodobností, ale mělo by to podstatnější význam až pro větší množství typů domů, než je tomu u mého diplomového projektu.

Pro větší názornost jsem se rozhodl některé výše popsané algoritmy prezentovat zobrazením kódu s příslušným popisem a umístil jsem je do programové přílohy část 7 a 8.

7.4 - Generování dynamických objektů

Dynamické objekty mají svou třídu v souboru Fobjects.h, která zajišťuje jejich vytvoření a zobrazení. Podstrom scény dynamického objektu je tvořen postupně z uzlů SoMatrixTransform, SoMaterial. Potom následuje buď SoTexture a SoCube nebo Sofie. Podle toho, jestli jsou použity jednoduché hranoly, které jsou méně náročné na CPU, nebo nahrávány modely (lze určit pomocí proměnné typu objektu).

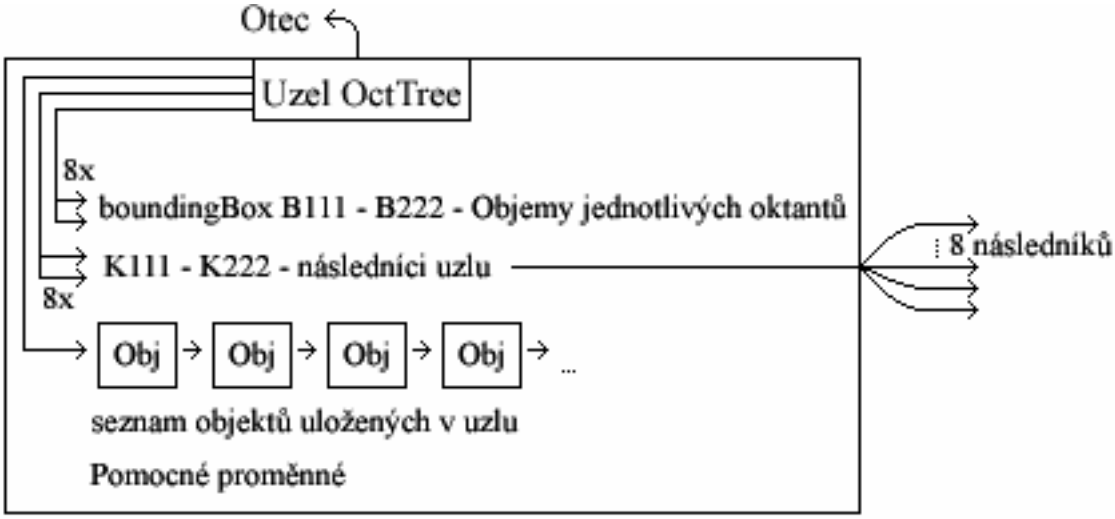
Samotné generování je potom velmi jednoduchý cyklus (Generátor.h), ve kterém jsou objekty náhodně generovány nad scénou a vkládány do seznamu objektů a do OctTree. Seznam objektů slouží k pohybu těchto objektů. Při zapnutém fyzikálním modelu je v každém časovém okamžiku procházen a je inicializována postupně rychlost a pozice každého jednotlivého objektu (Viz. kapitola 5.4 fyzika a pohyb dynamických objektů). Programový kód vygenerování jednoho typu objektů je v programové příloze část 9.

8. Řešení kolizí v projektu

8.1 - Úvod

Pro implementaci OctTree ve svém diplomovém projektu jsem zvolil OctTree s nastavitelnou velikostí počtu objektů v uzlu. Tento způsob je výhodnější pro fázi testování, kdy můžeme ovlivňovat jeho vlastnosti jako je hloubka a počet kolizních objektů v uzlu. Princip vkládání prvků do OctTree je v tom, že se prvek předloží kořenu scény a ten se ho pokusí uložit do svého uzlu. Pokud se uzel přeplní, pak jsou všechny objekty v něm obsažené rozeslány do jim příslušných podstromů a pro tyto podstromy se situace rekurzivně opakuje, dokud nejsou všechny objekty uloženy a žádný uzel není přeplněn. Dojde-li k situaci, že objekt je součástí dvou boundingBoxů, tak se samozřejmě nerozhoduje, kam bude uložen, ale dají se odkazy na něj do obou těchto následníků. Ve výsledku to potom může znamenat, že některé velké objekty se mohou vyskytovat až v desítkách listových uzlů. V mém projektu tato situace nastává pouze pro objekt podkladu scény. Je to jediný objekt dostatečně veliký v poměru vůči ostatním objektům, aby se do této situace mohl dostat. Poslední věcí, kterou zbývalo při návrhu algoritmu rozhodnout, bylo, zda budou objekty uloženy ve všech uzlech nebo pouze v těch listových. Pro statickou scénu je mnohem výhodnější používat ukládání do všech uzlů stromu, ale ve scéně dynamické se jeví jako výhodnější objekty „probublávat“ až do listů. Vygenerování takového stromu je sice časově náročnější, ale následně při pohybu dynamických objektů scénou nedochází už téměř nikdy k rozpadům uzlů na poduzly a výsledný výkon je vyšší.

Třída OctreeNodes - základní stavební kámen OctTree



Obrázek 12.

8.2 - Objekty OctTree

Objektům, které vkládám do OctTree, jsem zcela zřejmě vytvořil novou třídu OctreeObjects. Tato třída obsahuje čtyři konstruktory, každý vhodný pro jiný typ vkládaných prvků. První konstruktor je prázdný, druhý konstruktor je pro kameru, třetí pro statické objekty a poslední pro objekty dynamicky se pohybující po scéně. Jako příklad uvádím hlavičku konstruktoru dynamických objektů: OctreeObjects(float VelX, float VelY, float VelZ, float PozX, float PozY, float PozZ, SbBox3f *Obj, letajiciObjekt *DynObj). Vidíme zde, že objekt má svoji velikost, pozici, objem (SbBox3f *Obj) a graficko-fyzikální reprezentaci (letajiciObjekt *DynObj). U statických objektů je místo graficko-fyzikální reprezentace odkaz na příslušný SoSeparator objektu, pro který v mém projektu zatím není uplatnění, ale z hlediska výhledu na snadnější použití a možné rozšíření v budoucnosti, jsem provedl kompletní implementaci.

Třída kromě konstruktorů obsahuje dvě jednoduché metody equal a Ja. Equal slouží k zjištění, zda objekt na jejím vstupu je totožný s naším objektem, který byl tázán. Metoda Ja vrací ukazatel na tázaný objekt. Obě funkce pro jejich jednoduchost a názornost uvádím přímo.

```
bool equal ( OctreeObjects Prvek )
{
	return ( Prvek.Objem == Objem );
}

OctreeObjects Ja()
{
	return *this;
}
```

8.3 - Třída dynamických objektů

Než-li začneme psát o samotném OctTree, tak se blíže podíváme na třídu dynamických objektů, které se pohybují ve scéně, jejich generování a uložení do paměti. Pro tvorbu dynamických objektů nám slouží třída Fobjects.h. V této třídě je nejpodstatnější její konstruktor, který vytváří dynamické objekty podle vstupních proměnných: letajiciObjekt (float pozicex, float pozicey, float pozicez, float rychlostObj, float zrychleniObj, SbVec3f smerPohybu, int cojeto, SoSeparator* root). Nový objekt tedy zná svoje dynamické vlastnosti - pozici, rychlost, směr pohybu a zrychlení. Dále má svoje grafické vlastnosti, které jsou určovány vstupní proměnnou cojeto. Podle této proměnné se konstruktor rozhoduje, jaký vzhled bude mít dynamicky se pohybující objekt. Mohou zde být použity jednoduché primitivní objekty Coin3D, jako jsou například SoCube, SoSphere a jiné, nebo složitější modely načítané ze souboru. Pokud je modelů načítaných ze souboru vyšší množství (desítky až stovky), pak se znatelně prodlužuje rychlost spouštění aplikace při jejich načítání. Její samotný běh není již o mnoho pomalejší (přibližně od pěti do deseti procent, podle složitosti modelu).

Mezi vnitřní proměnné třídy patří SbMatrix maticeee; a SoMatrixTransform* transform; které v konstruktoru slouží k umístění dynamického objektu ve scéně. K tomu by nám sice stačila SoTranslation, ale k dynamickému pohybu potřebujeme přístup k transformační matici, která je umístěna ve stromu scény před uzlem grafiky (SoCube, SoSpehere, Sofile atd…). V dalším průběhu programu, když potřebujeme změnit pozici nebo natočení objektu, nám stačí vhodně transformovat matici a o zbytek se postará jádro Coin3D. Třída SbMatrix nám poskytuje například následující užitečné metody: setTranslate (const SbVec3f &t), setTransform (const SbVec3f &t, const SbRotation &r, const SbVec3f &s), setRotate (const SbRotation &q) (kontruktor třídy dynamických je v programové příloze část kódu 10).

Nyní máme všechny potřebné nástroje pro vytvoření a práci s dynamickými objekty. K jejich reálnému použití je potřebujeme vhodně uložit do struktury, kterou pak v každém časovém okamžiku budeme procházet a aktualizovat jejich pozice, rychlosti nebo jiné požadované vlastnosti. Za tuto strukturu jsem zvolil třídu vektor: vector<OctreeObjects*> OctreeObjekty; Naše dynamické objekty jsou zde zapouzdřeny v třídě OctreeObjects (viz. předcházející kapitola). Samotná tvorba dynamických objektů je inicializována v bloku hlavního programu (UrbanGenerator.cpp). Pro příklad uvádím příkaz, který vygeneruje deset kolizních objektů, které mají model „Ufo“ (viz. kapitola 4.8). Mestecko.GenerujKolizniObjekty(2, 10, root, KorenOctree); Mestecko je zde instance třídy Generátor, která slouží k vygenerování všeho potřebného. Její metoda GenerujKolizniObjekty je v programové příloze číslo 11. Všimněme si zde, že kromě typu objektů a jejich počtů jsou navíc zadávány proměnné root a KorenOctree. Proměnná root označuje kořen scény a slouží konstruktoru Fobjects při umísťování objektů do grafu scény a KorenOctree slouží k umístění dynamických objektů do OctTree.

Vlastní generování objektů GenerujKolizniObjekty je poměrně jednoduchá metoda, která v cyklu náhodně umisťuje objekty ve scéně v předem definovaných mezích výšky, délky i šířky. Řídí se typem objektu při stanovování velikostí boundingBoxů a dále tento typ předává konstruktoru v Fobjects, který se stará o grafickou stránku věci, jak jsem zmínil v předchozím. Po nastavení všech potřebných vlastností jsou naše nové dynamické (kolizní) objekty přidávány do vektoru objektů, který slouží k jejich procházení, jak jsem již zmínil výše, a do OctTree. O pohybu dynamických objektů se zmiňuje podkapitola 5.3 a následující podkapitoly, ve kterých je vysvětlen jejich pohyb v OctTree.

8.4 - Třída OctTree uzlů

Třída OctreeNodes obsahuje dva jednoduché konstruktory. První z nich nastaví všechny počáteční hodnoty na nulu, případně NULL a druhý nastaví počáteční hodnoty podle svých vstupních proměnných. Dále obsahuje metody SbBool Prunik(SbBox3f Prvni, SbBox3f Druhy), void PridejPrvek(OctreeObjects Prvek), bool PridejPrvekAtestujKolize(OctreeObjects Prvek), void ZrusPrvek(OctreeObjects Prvek), const OctreeNodes* ZjistiKolize(const OctreeNodes &Uzel).

V těchto metodách je uložena veškerá podstatná práce s OctTree. Průnik je jednoduchá metoda zjišťující, zda dva boundingBoxy mají společný průnik. Třída SbBox3f sice obsahuje metodu itersection, ale v naší konkrétní aplikaci je těžko použitelná, protože detekuje průniky v místech, kde se dva objekty dotýkají. V naší aplikaci například všechny budovy stojí na zemi. Proto jsem vytvořil vlastní jednoduchou metodu průnik, která přesně splňuje naše požadavky. Tato metoda není příliš zajímavá, proto ji ani neuvádím v programové příloze.

8.4.1 - Metody PridejPrvek a PridejPrvekAtestujKolize

Metody PridejPrvek a PridejPrvekAtestujKolize se liší tím, že v druhé z nich je navíc testování kolizí, a proto je tato metoda časově a výkonově náročnější. Při tvorbě OctTree by nám stačila pouze metoda s testováním kolizí, ale v naší aplikaci je mnoho situací, kde předem víme, že vkládaný prvek nebude v kolizi, a proto by použití jediné funkce PridejPrvekAtestujKolize bylo neefektivní. A to nejenom při generování scény, kdy se vkládají statické objekty, které jistě nebudou v kolizi, ale i při pohybu dynamických k objektů. K tomu se ovšem dostaneme později.

Jak tedy fungují zmíněné dvě metody? Přijde-li prvek do uzlu, nejprve se naše metoda ptá, zda je listem nebo prázdným uzlem, protože, jak již bylo zmíněno výše, prvky se ukládají pouze do listových uzlů. Pokud jsme v listovém uzlu a není jeho kapacita naplněna, pak se prvek normálně přidá do uzlu. V případě metody PridejPrvekAtestujKolize se po přidání prvku spustí test kolizí na prvky v daném listu. Pokud kolize není nalezena, celá metoda vrátí false, jinak vrátí true. Pokud je kapacita listového uzlu naplněna, pak se vytvoří osm jeho následníků, mezi něž se nakopírují jednotlivé prvky podle toho, kam náleží (metoda přidání prvku se rekurzivně spouští na jednotlivé následníky). Pokud například jeden z prvků zabírá tři oktanty, pak je předán do všech tří příslušných nových listových uzlů. Stejně se metoda chová, i když je nelistovým uzlem. Prvek pošle rekurzivně svým následníkům. Díky této rekurzivitě jsou kolize, které byly detekovány (pokud je detekujeme), z nižších úrovní „probublávány“ zpět nahoru.

Poslední otázkou nám zůstává, jak metoda rozezná, do kterých oktantů má své prvky rozeslat. Odpověď je poměrně jednoduchá. V každém uzlu při jeho konstrukci vytvoříme osm boundingBoxů příslušejících k jednotlivým oktantům. Pak nám stačí boundingBox prvku otestovat na průnik se všemi osmi boundingBoxy oktantů. Kde nám nastanou průniky, tam příslušný prvek pošleme. Je to rychlé, jednoduché a nenáročné.

8.4.2 - Metoda ZjistiKolize a složitost detekce kolizí

Metoda ZjistiKolize je rekurzivní. Je volána na příslušný uzel a pak se rekurzivně dívá do svých následníku, zda nenastává náhodou kolize v nich (je například používána v metodě PridejPrvekAtestujKolize, ale zde je volána vždy jen na listové uzly a její rekurzivní vlastnost zůstává nevyužita).

První, co tato metoda udělá, je, že otestuje kolize v aktuálně testovaném uzlu. Pokud kolize nenastává nebo pokud v uzlu nejsou žádné prvky, jdeme testovat rekurzivně do následníků.

Jak se tedy jednotlivé kolize testují? Vezmou se všechny prvky v uzlu a každý se testuje s každým, jestli nemají společný průnik (viz. pár následujících řádků kódu).

```
for (int i=0 ; i < Uzel.PocetPrvku-1 ; i++)
{
	for (int i2=i+1 ; i2 < Uzel.PocetPrvku ; i2++)
	{
		if (Prunik(*Uzel.Prvky[i].Objem,*Uzel.Prvky[i2].Objem))
		{ // Nastala Kolize
			kolize=true;
			return &Uzel;
		}
	}
}
```

V každém uzlu se tedy provede (N * (N-1))/2 testů, což je kvadratická složitost. Toto je nejlepší složitost, které lze dosáhnout bez použití algoritmů sloužících k optimalizaci detekce kolizí. Tím, že používáme OctTree o nastavitelném počtu prvků v jednotlivých uzlech, je výsledná složitost složená z logaritmické složitosti stromu a kvadratické složitosti v uzlech (názorně v kapitole 9 i s výsledky). Poměr logaritmické a kvadratické složitosti se dá v projektu snadno ovlivňovat změnou konstanty PocetPrvku. Zjevně čím větší je počet prvků, tím více získává na váze kvadratická složka výsledné složitosti. Na druhou stranu čím méně prvků je možné uložit do stromu, tím více roste jeho velikost a tím i paměťová náročnost. Tento vliv by se dal částečně omezit v aplikacích, kde by se prvky ukládali do všech uzlů OctTree. To se nám ale v naší aplikaci nevyplatí, jak jsme si již vysvětlili dříve. Otázkou zůstává, jaký počet prvků v uzlu je optimální. Ve scénách s menším celkovým počtem objektů dávají lepší výsledky uzly s malým počtem prvků. Ve scénách velikých je třeba za cenu nárůstu časové složitosti ulevit paměťové náročnosti a počet prvků poněkud zvýšit (více v kapitole 9 - testování).

8.4.3 - Metoda ZrusPrvek a její složitost

Stejně tak, jako se prvky přidávají do všech uzlů, musíme při odebírání prvků projít celý strom a všechny prvky v něm obsažené. To nám dává stejnou logaritmickou složitost pro průchod stromem jako přidání prvku, k tomu se přidává lineární složka na průchod všemi prvky obsaženými v uzlech. Kdyby se tedy OctTree nepoužíval, byla by metoda rušení prvku pouze lineární. Tedy o něco rychlejší, ale vyšší náročnost v logaritmické části v operaci ZrusPrvek je mnohem menší než úspora na kvadratické části při přidávání prvků. Myslím, že kód této metody hovoří sám za sebe.

```
void ZrusPrvek(OctreeObjects Prvek)
{
int i=0;
if (PocetPrvku>0)
	{do
		{
		if (Prvek.equal(Prvky[i])) // Je hledaný prvek nalezen?
			{ Prvky.removeFast(i); PocetPrvku=PocetPrvku-1; } // Je nalezen
			else
			{ i=i+1; } // Není nalezen
		}
```

```
        while (i!=PocetPrvku);
        }
        // Rekurzivní projití všech větví stromu
        if (K111!=NULL) { K111->ZrusPrvek(Prvek);}
        if (K112!=NULL) { K112->ZrusPrvek(Prvek);}
        if (K121!=NULL) { K121->ZrusPrvek(Prvek);}
        if (K122!=NULL) { K122->ZrusPrvek(Prvek);}
        if (K211!=NULL) { K211->ZrusPrvek(Prvek);}
        if (K212!=NULL) { K212->ZrusPrvek(Prvek);}
        if (K221!=NULL) { K221->ZrusPrvek(Prvek);}
        if (K222!=NULL) { K222->ZrusPrvek(Prvek);}
}
```

8.5 - Detekce kolizí v projektu

Nyní jsme se seznámili se všemi potřebnými nástroji k zprovoznění detekce kolizí v našem městě. Máme vytvořené město, dynamické objekty, kameru a pomocí metody PridejPrvek vytvořený celý OctTree naplněný příslušnými prvky. Metoda PridejPrvekAtestujKolize nám provádí test na kolizi při přidávání prvku. Nyní, zapne-li se mód detekce kolizí (fyzikální mód), tak se detekují kolize ve dvou nezávislých úrovních. První detekcí je detekce kolizí při pohybu kamery a druhá je detekce kolizí dynamických objektů. Princip je u obou zcela stejný. Liší se pouze ve výpočtu pohybu a reakci na kolizi. Rozdíl ve výpočtech pohybu je v tom, že kamera je ovládána uživatelem a dynamické objekty jsou ovládány výpočetním modelem popsaným v kapitole 5.3. Rozdíl v reakci na kolizi je v tom, že kamera je pouze zastavena (jsou vynulovány její vektory rychlosti a zrychlení), ale dynamický objekt je nejenom zastaven, je mu „zničen motor“. To znamená, že je u něj v následujících iteracích programu vynechán výpočet jeho rychlosti a pak na něj už působí pouze vektor gravitace a objekt přestává letět - padá k zemi.

Pohyb objektů a detekce kolizí je naprogramována tak, že se nejdříve odebere objekt z OctTree. Potom se aktualizuje podle rychlosti a směru jeho pozice ve scéně a metodou PridejPrvekAtestujKolize se zjistí, zda na nových souřadnicích nebude objekt v kolizi. Pokud ke kolizi nedojde, tak je vše v pořádku. Pokud ke kolizi dojde, pak je tento pokusně vložený objekt odebrán a zpět je vložen objekt původní. Již bez testu na kolize, protože o něm víme, že v kolizi nebyl. Konkrétní kód je v programové příloze, část 12. Jedná se o test kolize kamery a následné reakce na výsledek.

9. Testování

K tomu, abychom mohli testovat výkonnost, potřebujeme příslušné nástroje. Potřebujeme měřit časové body v různých místech programu a naměřené výsledky prezentovat vhodným způsobem. Aktuální čas v daném okamžiku získáme metodou Coin3D: SbTime Ted = SbTime::getTimeOfDay(); Nyní si můžeme změřit čas ve dvou bodech průběhu programu a pouhým rozdílem zjistíme dobu trvání měřeného úseku. Já jsem ve své aplikaci proměřoval dva nejpodstatnější údaje, kterými jsou počet snímků za vteřinu (FPS) a dobu, kterou program potřebuje na operace s OctTree. K tomu bych poznamenal, že program může provést více snímků, než je nakonec zobrazeno. Například v jednoduchých scénách může být provedeno a vypočteno 300 snímků za sekundu, ale obnovovací frekvence monitoru je například 100hz. Potom se zobrazuje každý třetí snímek, který by vypočítán jádrem Coin3D.

K rozumné prezentaci výsledků jsem použil třídu SoPerphGraph obsaženou v souborech SoPerphGraph.h a SoPerphGraph.cpp, jejíž autorem je Ing. Jan Pečiva. Tato třída je public domain a její použití mi doporučil sám autor. Tato třída dostává na svůj vstup číselné hodnoty a pak je v čase zobrazuje jako graf. Před jejím použitím musíme provést její inicializaci a nastavení jednotlivých atributů. Vytvoření jednoho takového grafu v pravém dolním rohu obrazovky prezentuje následující blok kódu.

```
SoPerfGraph::initClass();

    fpsGraph = new SoPerfGraph();
    fpsGraph->horScreenOrigin.setValue(SoPerfGraph::RIGHT);
    fpsGraph->horAlignment.setValue(SoPerfGraph::RIGHT);
    fpsGraph->position = SbVec2f(-0.03f, 0.03f);
    fpsGraph->size = SbVec2f(0.23f,0.11f);
    root->insertChild(fpsGraph,0);
```

Hodnoty se potom do grafu vkládají následujícím způsobem:

```
fpsGraph->appendValue(ST2.getValue()-ST1.getValue());
```

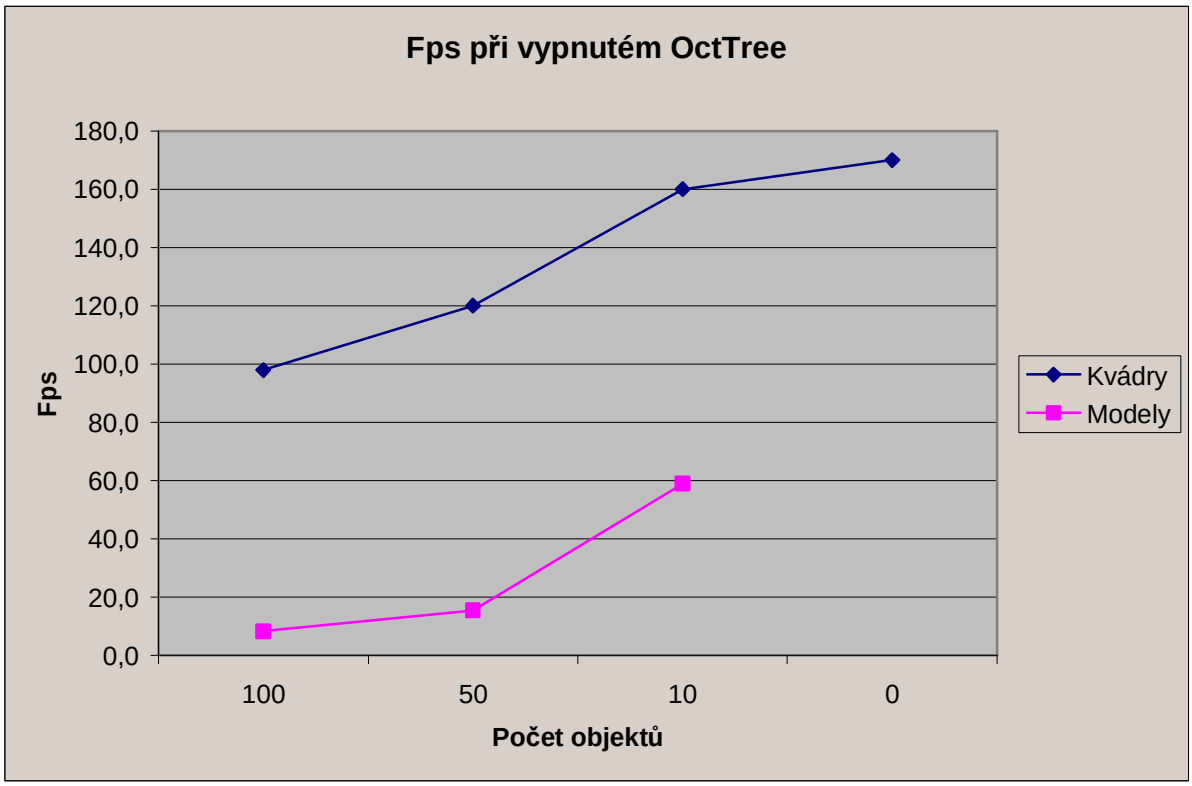
9.1 - Grafická náročnost

Při testování výsledné scény jsou výsledky znatelně ovlivněny její velikostí a tím na jednu stranu počtem operací v OctTree a na druhou stranu její grafickou náročností. O tom hovoří i následující měření. V první tabulce je rozdíl v náročnosti scény v závislosti na modelu použitém jako dynamický objekt. Jedná se o scénu o velikosti 10x10, když je vypnutá detekce kolizí. Již zde je znatelný velký rozdíl v náročnosti scény. Všimněme si, že na rychlost výpočtu operací v OctTree nemá rozdíl mezi modely vliv.

	Vypnuté OctTree, FPS		Zapnuté OctTree, FPS		Zapnuté OctTree, doba výpočtu v ms.	
Počet Objektů	Kvádry	Modely	Kvádry	Modely	Kvádry	Modely
100	98,0	8,3	4,50	2,30	0,28	0,24
50	120,0	15,5	7,80	5,40	0,11	0,11
10	160,0	59,0	34,00	24,00	0,020	0,020
0	170,0		106,00		0,0028	

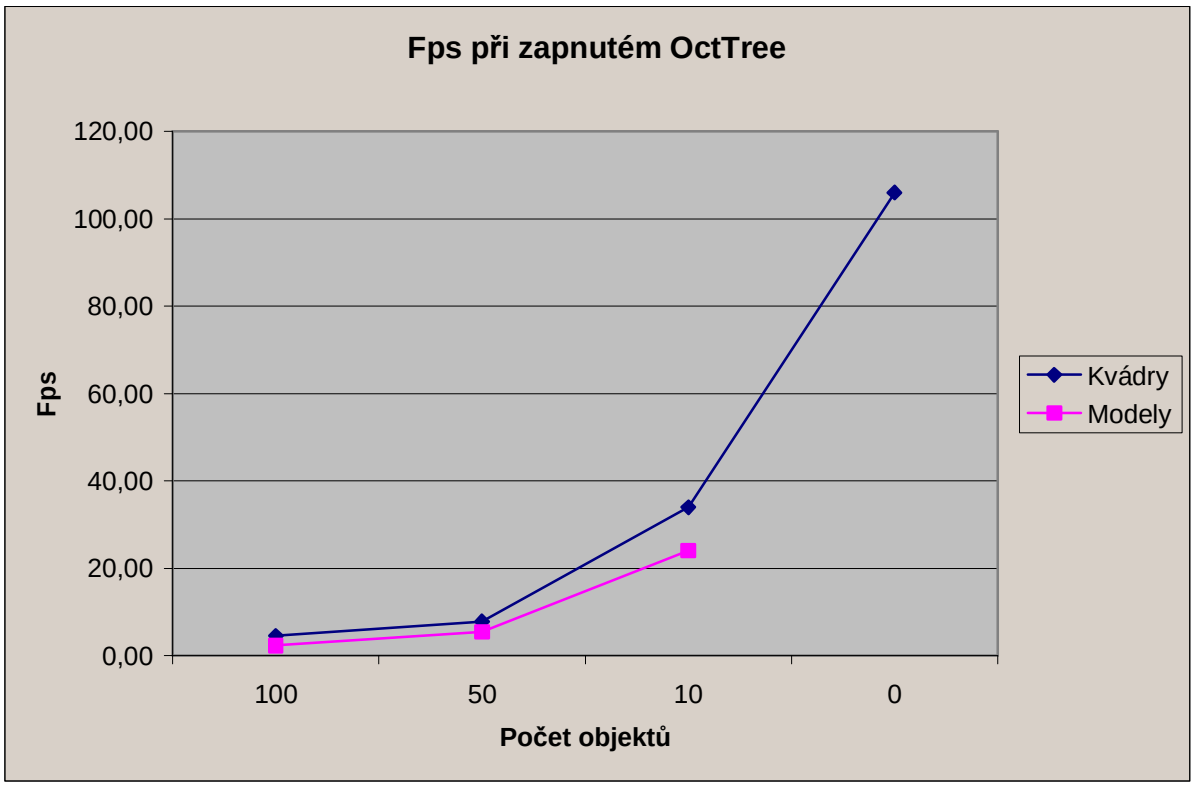
Tabulka 01

Na prvním grafu jsou hodnoty FPS naměřené s dynamickými objekty ve tvaru kvádru a stejné testy s dynamickými objekty ve tvaru „Ufo“. Jedná se o scénu o velikosti 10x10, když je vypnutá detekce kolizí. Již zde je znatelný velký rozdíl v náročnosti scény.



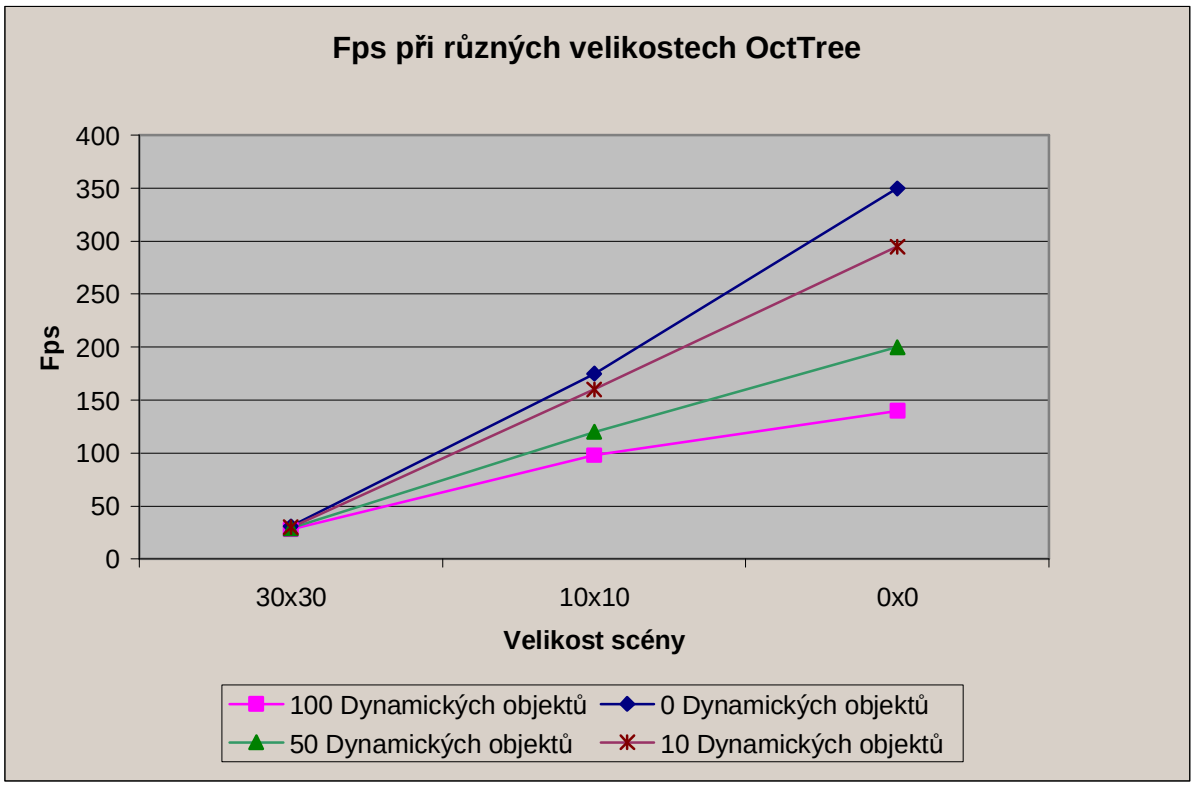
Graf 01

Na druhém grafu jsou hodnoty FPS naměřené s dynamickými objekty ve tvaru kvádru a stejné testy s dynamickými objekty ve tvaru „Ufo“. Jedná se o scénu o velikosti 10x10, když je zapnutá detekce kolizí. Nyní je rozdíl ve výkonu menší, protože jej stírá detekce kolizí (v obou případech jede procesor na stoprocentním vytížení). Na výkonnějším počítači, než byl testovací, by byl rozdíl znatelnější.



Graf 02

Třetí graf ukazuje náročnost scény na výkon počítače v prohlížecím módu s jednoduchými modely kvádru a rozdílným počtem dynamických objektů. Je názorně vidět, že při menší scéně hraje počet dynamických objektů mnohem větší roli (velikost scény ovlivňuje hloubku stromu). To je na jednu stranu dáno tím, že v menší scéně se počet objektů blíží celkovému počtu objektů. Jak scéna roste, je tento poměr stírán. Druhým aspektem je, že při velké scéně se dosahuje hranice výkonnosti hardwaru, který stírá rozdíly.

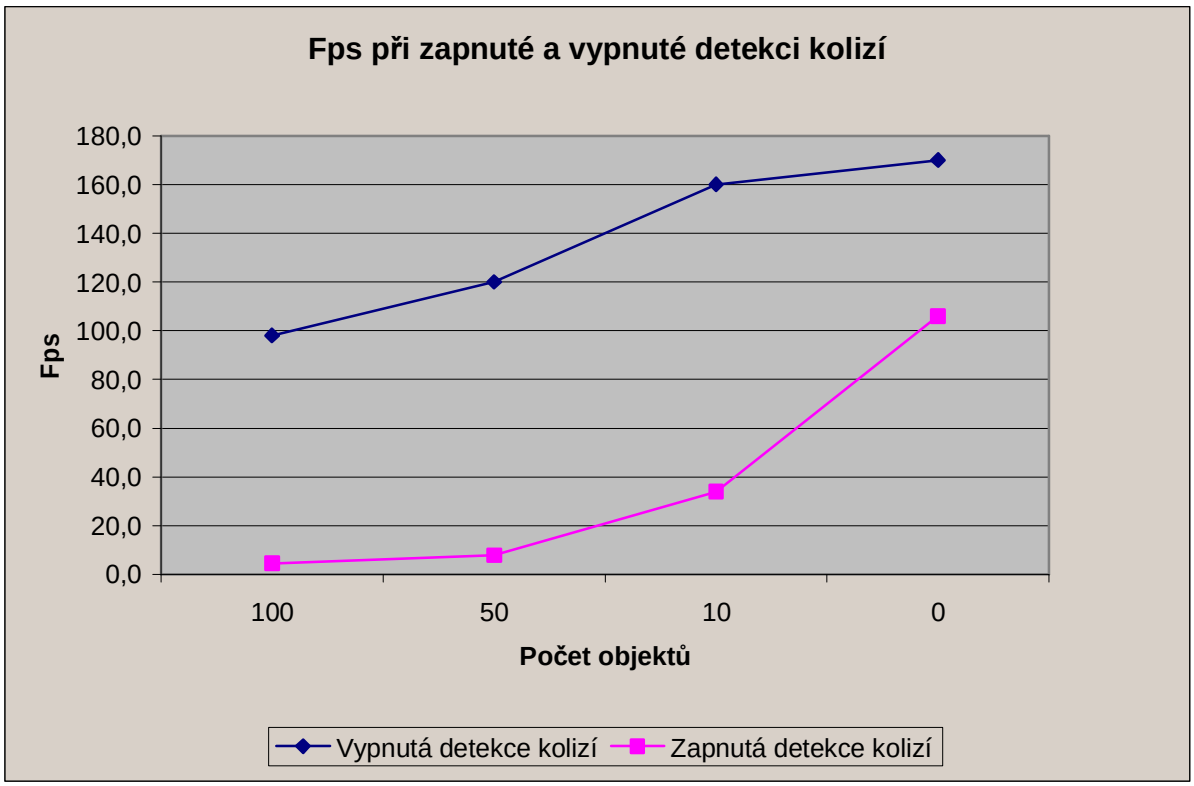


Graf 03

9.2 - Testy detekce kolizí

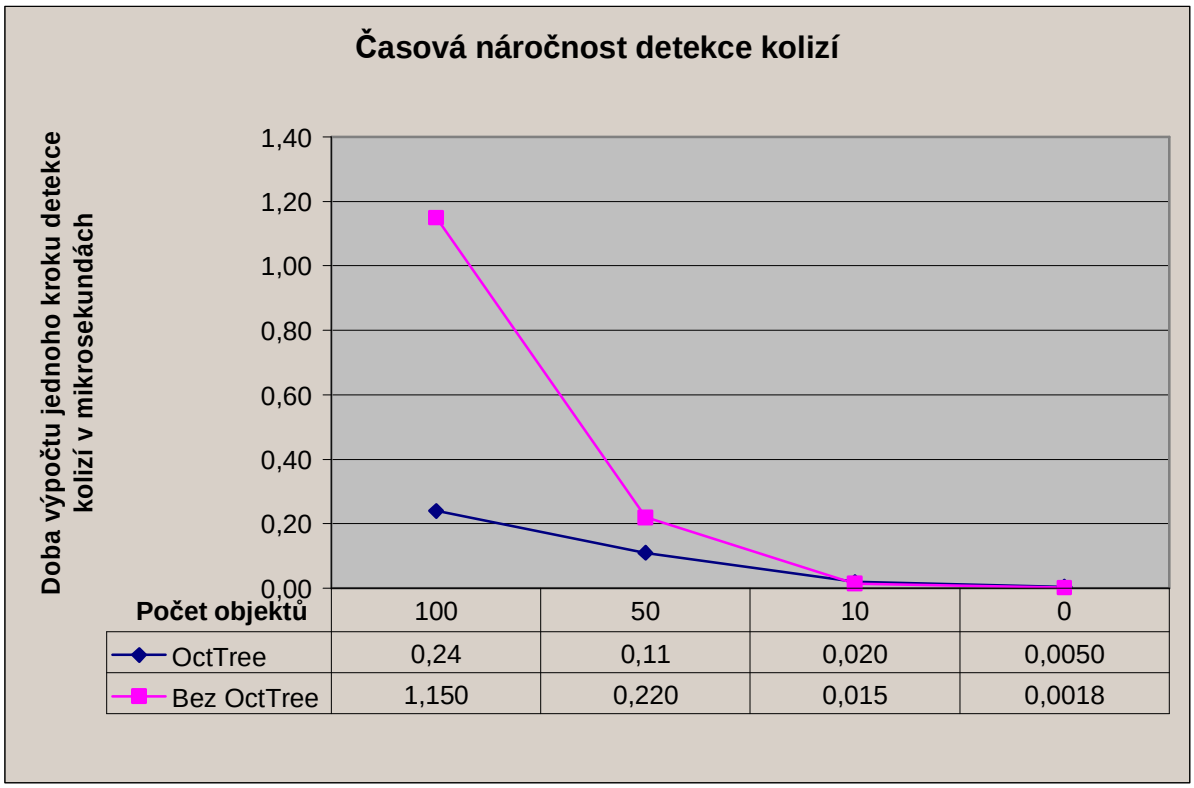
Při testování detekce kolizí jsem se zaměřil na náročnost detekce kolizí a následně na to, jaké urychlení nám poskytuje použití OctTree. Další podstatné faktory jsou například velikost scény a počet statických objektů ve scéně ovlivňujících velikost OctTree, počet dynamických objektů, které svým pohybem vyvolávají neustále operace odebírání a přidávání prvků a zatěžují systém.

Čtvrtý graf v pořadí ukazuje na snížení výkonu (FPS) při zapnutí detekce kolizí. Je zřejmé, že zpočátku klesá výkon rychleji, ale při rostoucím počtu objektů se pokles výkonu snižuje. Je to dáno tím, že strom roste do hloubky stále pomaleji (logaritmicky). Hodnoty jsou naměřeny na scéně velikosti 10x10 s dynamickými objekty ve tvaru kvádru.



Graf 04

Na grafu 5 je názorně vidět časová náročnost detekce kolizí při zvětšujícím se počtu kolizních objektů. Při nízkém počtu kolizních objektů je dokonce detekce kolizí bez OctTree o něco rychlejší. Je to dáno tím, že správa OctTree zabere trochu času navíc ve srovnání s jednoduchostí detekce bez OctTree. Zrychlení se objevuje až při vyšším počtu objektů, kde náročnost detekce bez OctTree roste kvadraticky. Hodnoty jsou naměřeny na scéně velikosti 10x10 s dynamickými objekty ve tvaru kvádru. Zde bych podotknul, že čím větší je počet objektů (třeba i statických) ve scéně, tím je detekce kolizí bez OctTree časově náročnější.



Počet objektů	100	50	10	0
OctTree	0,24	0,11	0,020	0,0050
Bez OctTree	1,150	0,220	0,015	0,0018

Graf 5

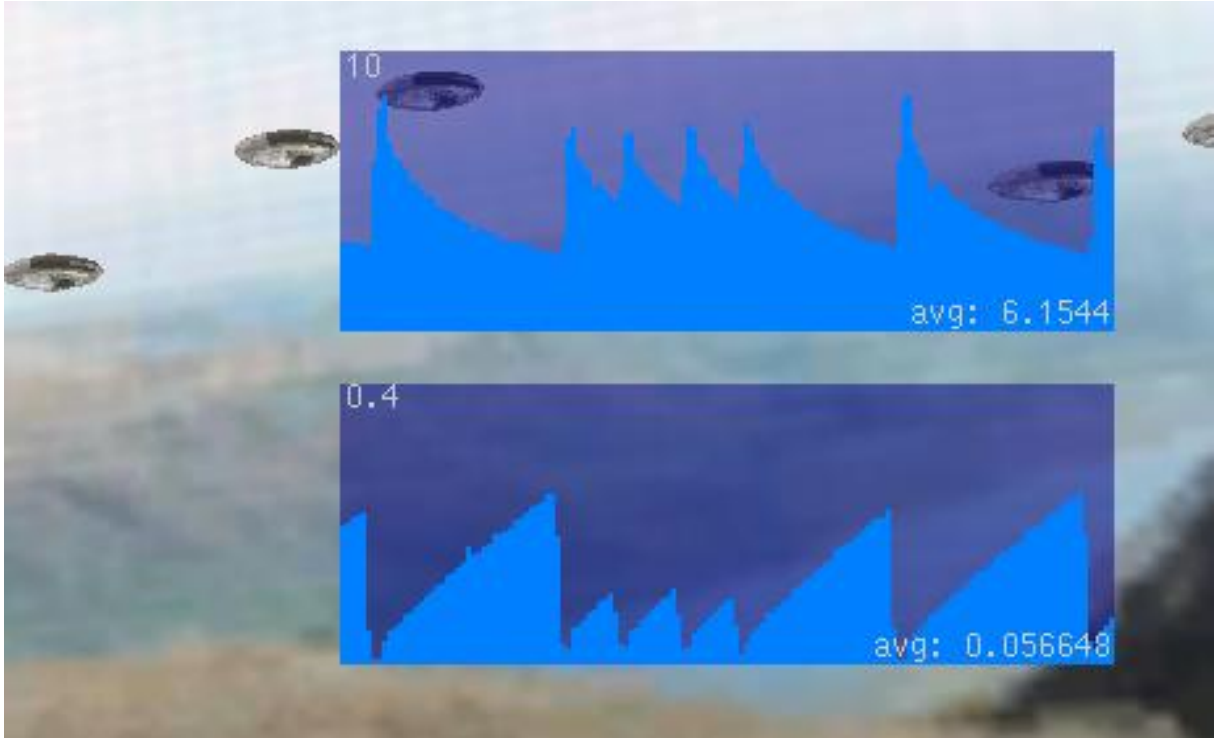
9.3 Závěr a zajímavé postřehy

Tabulka 02 uvádí přehledně některé z naměřených hodnot při testování výkonnosti. Všimněme si rozdílu hodnot u nulového počtu dynamických objektů, když je kamera v kolizi a když není. Kolizí je myšleno, že „dosedne“ na zem a v každém cyklu programu se dostane do kolize vlivem gravitace, je vrácena a opět se dostává do kolize. Pokud nemáme žádné jiné dynamické objekty, jsou hodnoty dobře měřitelné. Dobrý pozorovatel si jistě všimne, že detekce kolizí bez použití OctTree je v dané situaci o něco málo rychlejší. Proč je tomu tak, jsem již zmiňoval. Ve scéně, která obsahuje pouze podklad a kameru, není rozdíl mezi stavem kolize a bez kolize detekovatelný.

Velikost scény	Počet dynamických objektů	Lepší modely?	OctTree?	FPS s vypnutým OctTree	FPS zapnutého OctTree	Doba výpočtu OctTree	Poznámka
30x30	100	N	A	28	0,34		
30x30	50	N	A	29	0,82		
30x30	20	N	A	30	1,72		
30x30	10	N	A	30	3,74		
30x30	5	N	A	31	5,70		
30x30	2	N	A	32	9,60		
30x30	1	N	A	31	11,00		
10x10	100	A	A	8,3	2,30	0,28	
10x10	100	N	A	98,0	4,50	0,24	
10x10	50	A	A	15,5	5,40	0,11	
10x10	50	N	A	120,0	7,80	0,11	
10x10	10	A	A	59,0	24,00	0,020	
10x10	10	N	A	160,0	34,00	0,020	
10x10	0		A	170,0	106,00	0,0028	K. není v kolizi
10x10	0		A	170,0	83,00	0,0050	Kamera v kolizi
10x10	100	A	N	8,8	1,30	1,230	
10x10	100	N	N	106,0	1,60	1,150	
10x10	50	A	N	16,5	3,10	0,250	
10x10	50	N	N	115,0	3,70	0,220	
10x10	10	A	N	57,0	20,00	0,030	
10x10	10	N	N	165,0	42,00	0,015	
10x10	0		N	175,0	120,00	0,0014	K. není v kolizi
10x10	0		N	175,0	110,00	0,0018	Kamera v kolizi
0x0	100	A	A	9,1	7,80	0,015	
0x0	100	N	A	140,0	41,00	0,016	
0x0	50	A	A	17,5	15,50	0,004	
0x0	50	N	A	200,0	95,00	0,004	
0x0	10	A	A	71,0	65,00	0,0005	
0x0	10	N	A	295,0	220,00	0,0005	
0x0	0		A	350,0	280,00	0,0004	K. není v kolizi
0x0	0		A	350,0	280,00	0,0004	Kamera v kolizi
0x0	100	A	N	9,2	3,2 - 6,8	0,01 - 1,2	viz obr

Tabulka 02

Speciálním případem, který je znázorněn na obrázku 13, je detekce kolizí ve scéně pouze s dynamickými objekty a bez použití OctTree. Výkyvy jsou dány metodami ZrusPrvek, PridejPrvekAtestujKolize a metodou Coin3D removeFast. RemoveFast slouží k odstranění objektu ze seznamu objektů v OcttreeNode (v našem případě je pouze jeden OcttreeNode, ve kterém jsou všechny prvky). Odstranění funguje tak, že prvek je odstraněn ze své pozice a na jeho místo je vložen prvek, který byl v seznamu objektů na posledním místě. Výkonnost se tedy mění podle momentálního přeskládání seznamu objektů, které se postupně mění. Dostáváme tedy jak nejhorší, tak nejlepší případy kvadratického řešení.



Obrázek 13.

9.3.1 Konfigurace testovacího stroje

Všechny testy byly provedeny na následující konfiguraci:

Procesor: AMD Sempron(tm) 2200+, 1.5 GHz
Grafická karta: Radeon 9600 series
Paměť: 1GB RAM

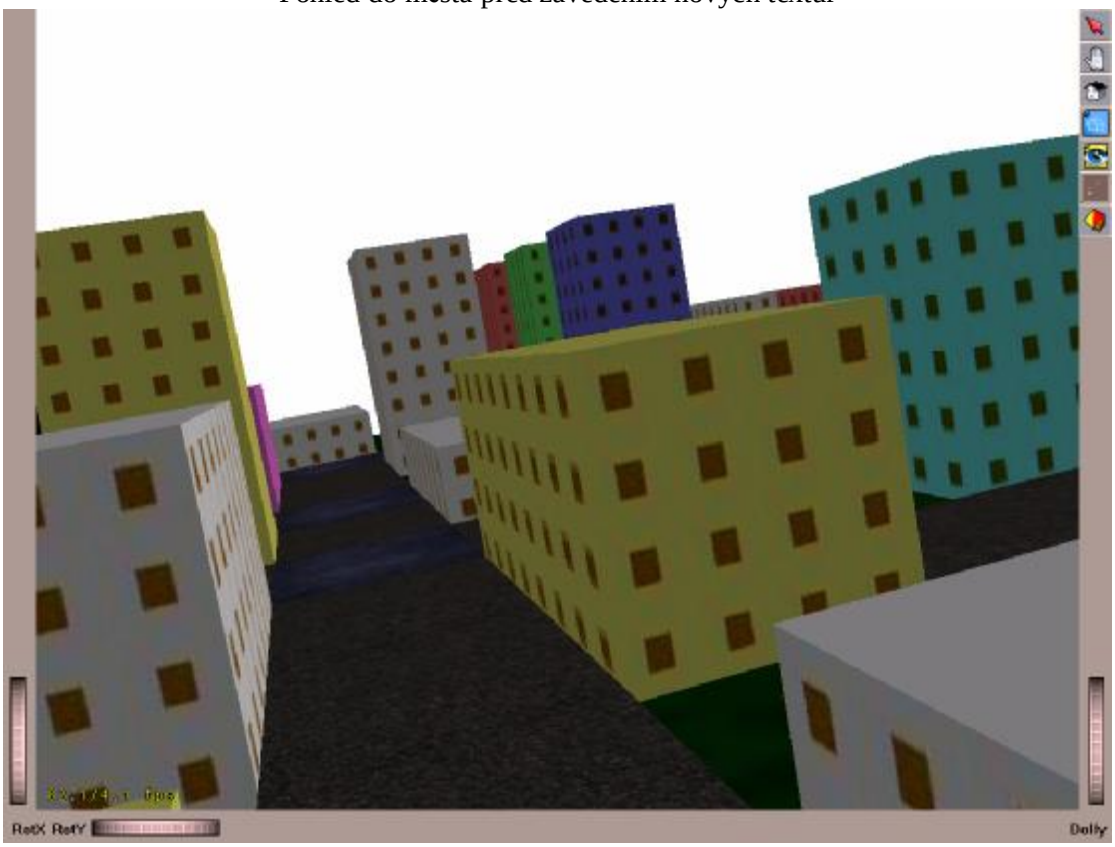
10. Kapitola - Zhodnocení

Protože veškeré umístění stavebního prvku (silnic) je založeno na náhodě, jsou města v některých svých konkrétních podobách jako by postavena „šíleným“ architektem. Hlavním nedostatkem tohoto řešení je tedy možnost určitých nelogických situací: může se stát, že se ve scéně vyskytne silnice vedoucí odnikud nikam nebo naopak, že je dům posazen „jen tak“ do trávy a nevede k němu přístupová cesta. Přes tyto občasné „nedostatky“ působí výsledky převážně jako městské čtvrti nebo průmyslové zástavby (zde záleží na vhodné volbě parametrů a textur).

Z předchozích kapitol víme, že OctTree přináší v detekci kolizí nezanedbatelné zrychlení. Vhodné je převážně pro scény obsahující velké množství objektů. Jak statických, tak i dynamických. Jeho použití je tedy velkým přínosem pro aplikace zabývající se detekcí a řešením kolizí.

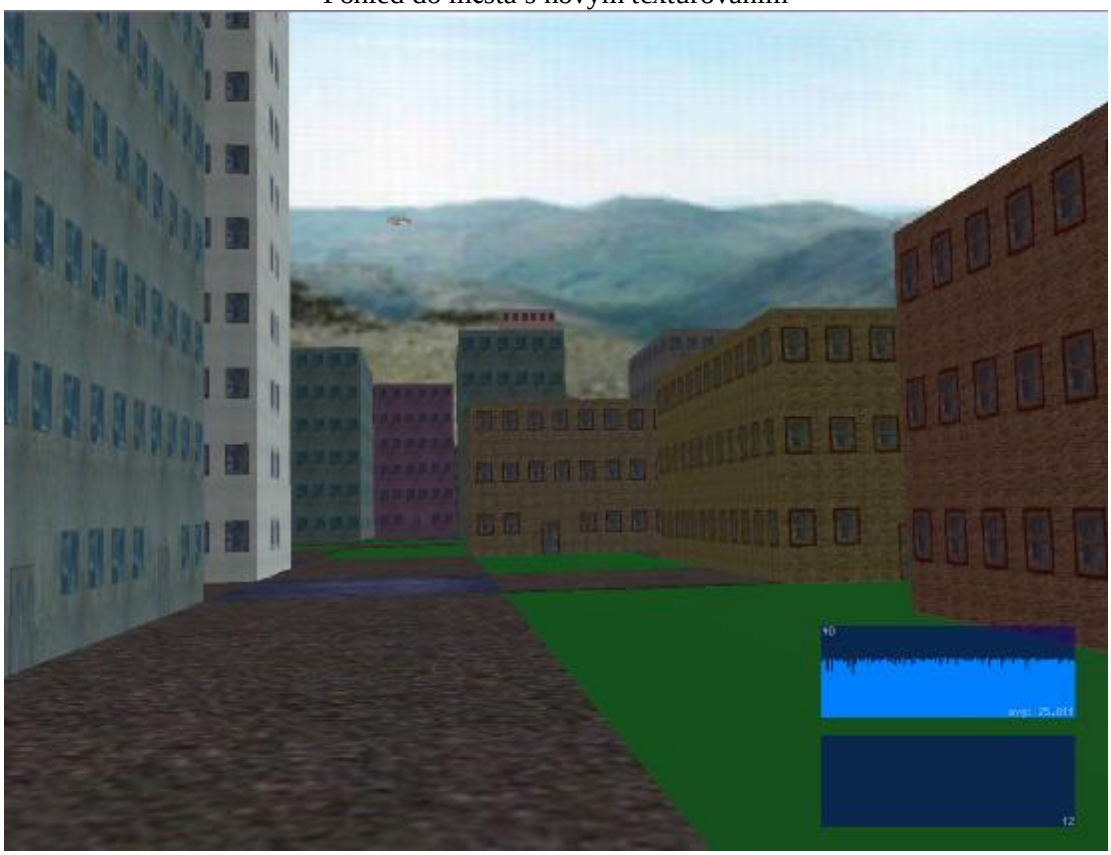
Město před zaváděním grafiky na obrázku 14 a na obrázku 15 je aktuální finální produkt aplikace s novými texturami, dynamickými objekty a SkyBoxem

Pohled do města před zavedením nových textur



Obrázek 14.

Pohled do města s novým texturováním



Obrázek 15.

10.1 - Možnosti rozšíření

Protože je aplikace sestavena z několika na sobě nezávislých tříd, lze každou z nich rozšiřovat nebo optimalizovat nezávisle na ostatních. Rozšířit by se například dalo generování města zavedením dalších omezujících podmínek na zástavbu města, aby nedocházelo k žádným logickým komplikacím a dále potom v zavedení členitosti terénu, na kterém město stojí. Zde by pravděpodobně největší problémy mohly působit konflikty textur s povrchem (okno končící v zemi a podobně). Potom bychom mohli řešit strmost terénu z pohledu možnosti zastavění (do příliš strmých úbočí se budovy nestaví). Další možnosti rozšíření by mohlo být zvětšení rozmanitosti objektů jak statických, tak i dynamických. S tím souvisí i zavedení nových textur.

Dále by se jistě dala celá aplikace postupně optimalizovat na vyšší výkon a nižší nároky na grafiku. Optimalizovat načítání modelů a textur z disku. Další specifickou kapitolou je pohyb dynamických objektů a reakce na jejich kolize, která poskytuje velký prostor fantazii. Například by nebylo nezajímavé udělat sestřelování těchto objektů a jejich exploze. Poslední kapitolou jsou detekce kolizí a zefektivnění jednotlivých algoritmů.

Závěr

Cílem práce bylo jak generování virtuální městské zástavby a její následné zobrazení, tak i řešení kolizí pomocí stromových algoritmů OctTree a testování jejich výkonnosti. Věřím, že se mi cíl práce podařilo dosáhnout a možná v některých místech i rozšířit.

Použitá literatura :

1. Mark A. DeLoura a kolektiv autorů: *Game programming gems. United States of America*, Charles river media, 2000

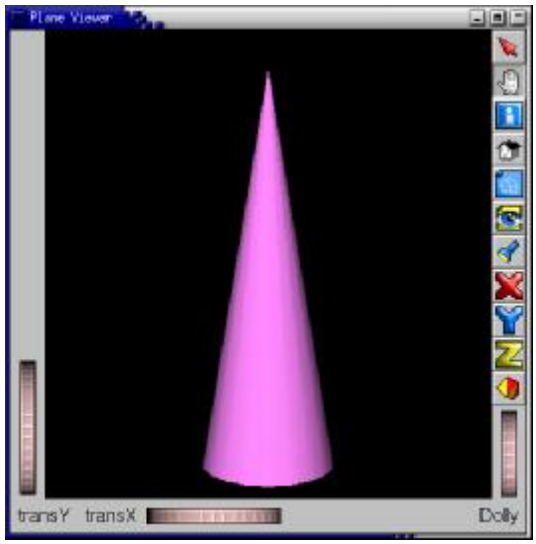
2. Haman Samet a Robert E.Webster: *Data structures: Hiearchical Data Structures and Algorithms for Compurer Graphics*, IEEE Computer Graphics & Aplications, květen 1988

3. Kolektiv autorů: *Domovské webové stránky firmy System in Motion* http://www.sim.no/
4. Kolektiv autorů: *Domovské webové stránky Coin3D* http://www.coin3d.org/
5. Kolektiv autorů: *Stránky dokumentace Kolektiv autorů: k Coin3D* http://doc.coin3d.org/
7. Kolektiv autorů: *Domovské webové stránky firmy SGI* http://www.sgi.com/
8. Kolektiv autorů: *Stránky dokumentace k Open Inventor* http://oss.sgi.com/projects/inventor

Obrazová příloha

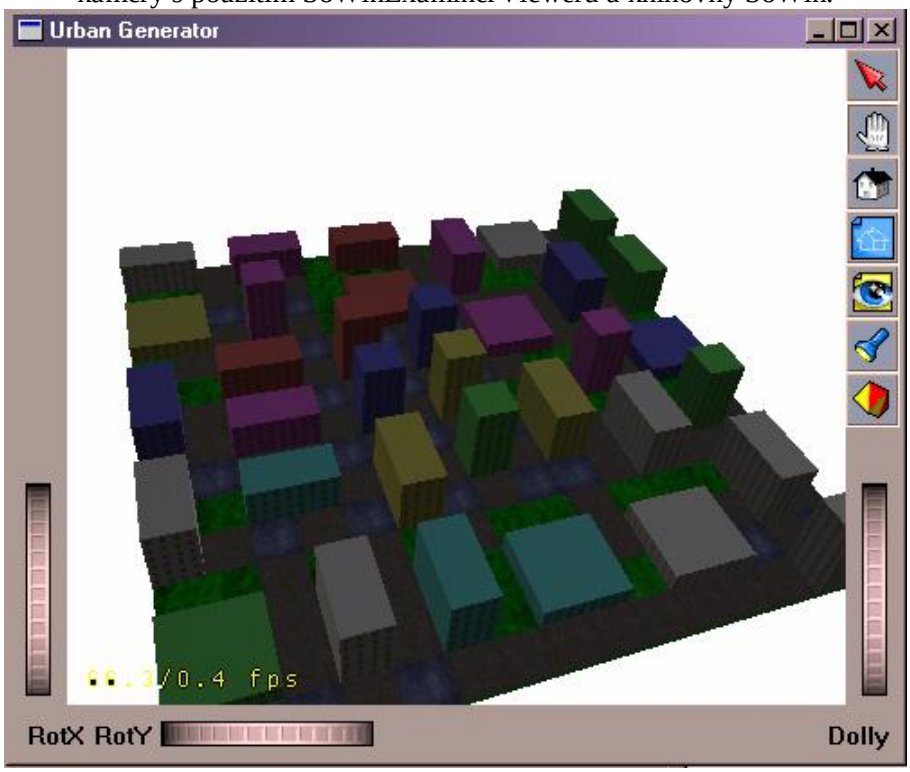
Na prvních dvou obrázcích jsou příklady prohlížečů používaných v Coin3D

Obrázek 16 - čerpáno z http://www.coin3d.org/



Obrázek 17 - čerpáno z http://www.coin3d.org/

Na tomto obrázku je znázorněno původní město se starými texturami ještě před přidáním kamery s použitím SoWinExaminerVieweru a knihovny SoWin.

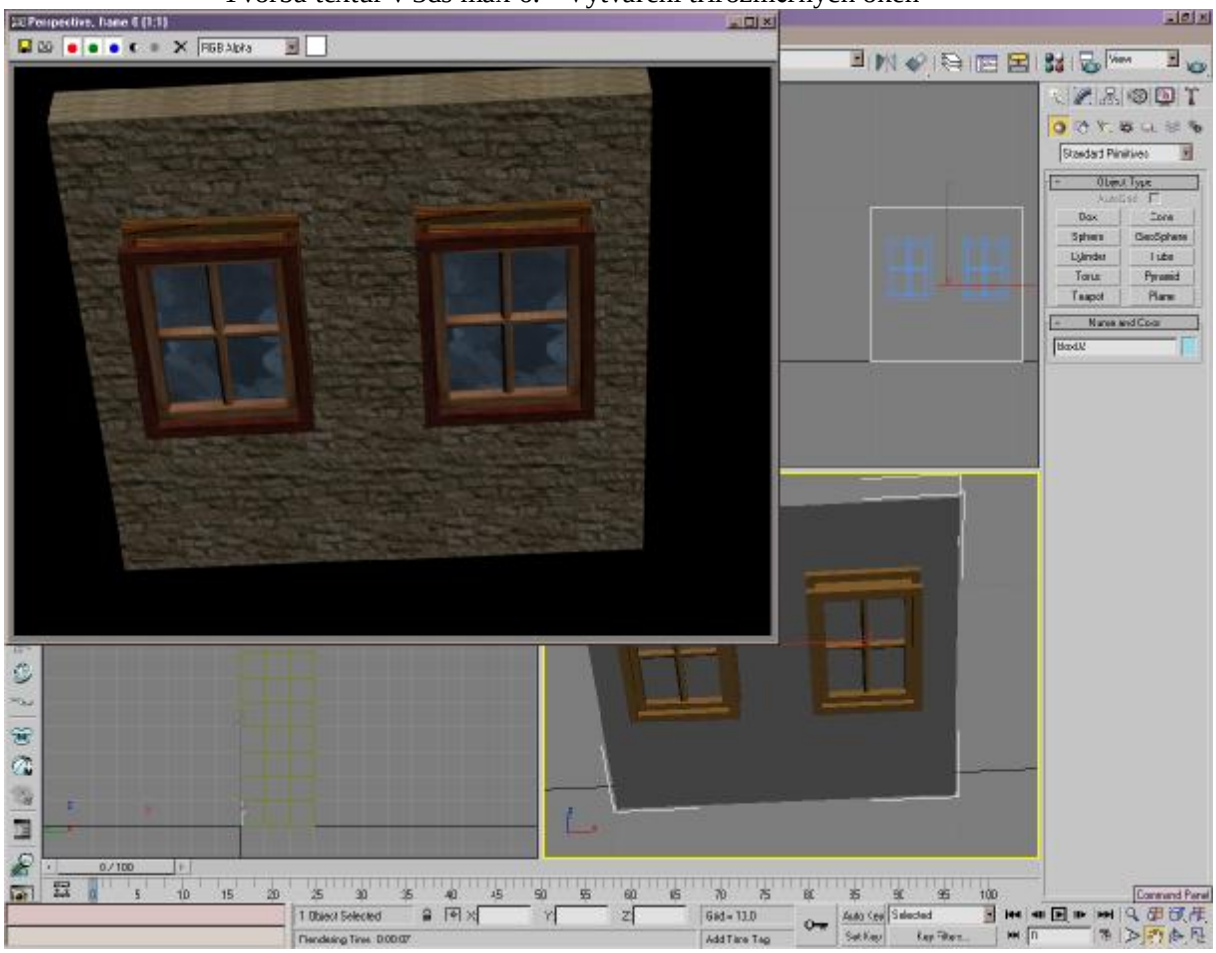


Obrázek 18.

Obrázek znázorňující základní osvětlení neotexturované scény
(stále s použitím SoWinExaminerViewer).

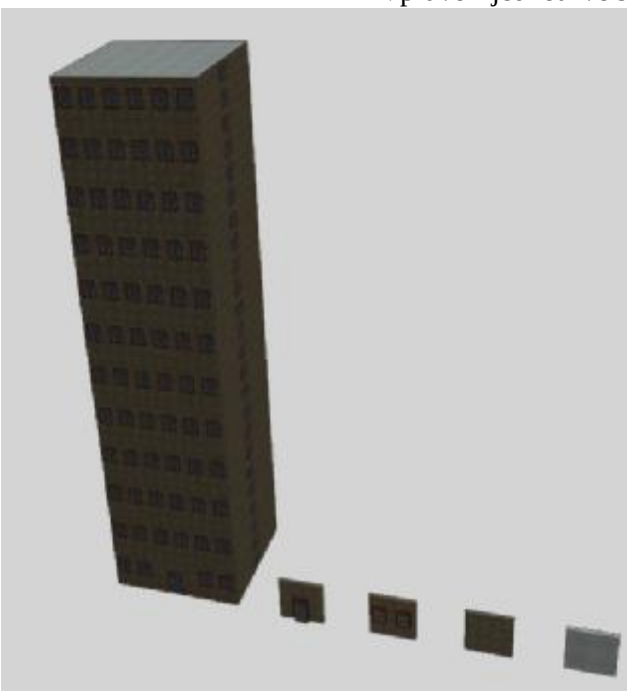
Obrázek 19.

Tvorba textur v 3ds max 6. - Vytváření třírozměrných oken



Obrázek 20.

Tvorba textur v 3ds max 6 - Vlevo - složení domu z jednotlivých panelů
Vpravo - jednotlivé stavební panely vytvořené z 3D modelu



Obrázek 21.

Obrázek 22.

Obrázek 23.

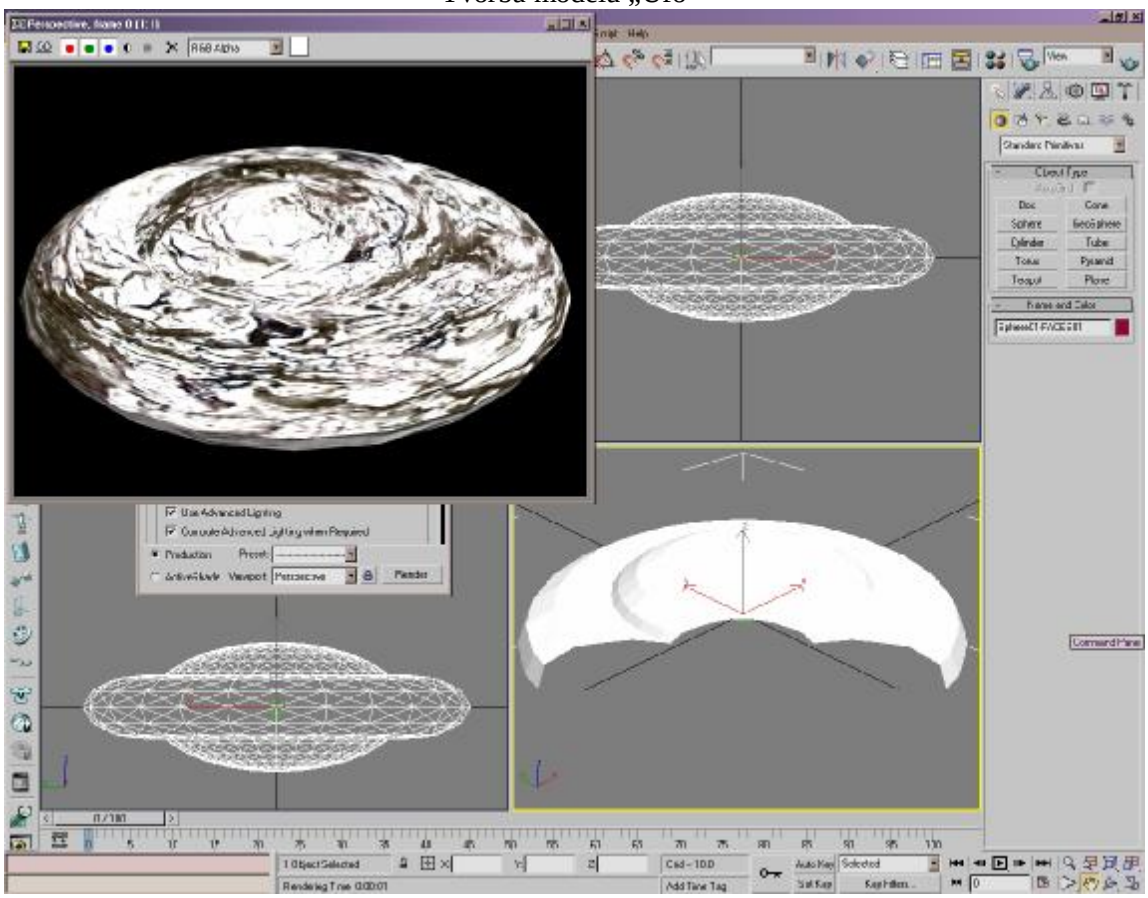
Tvorba textur v 3ds max 6 - výsledná textura na jedné stěně.

Obrázek 24.

Ukázkový model města z 3ds max 6

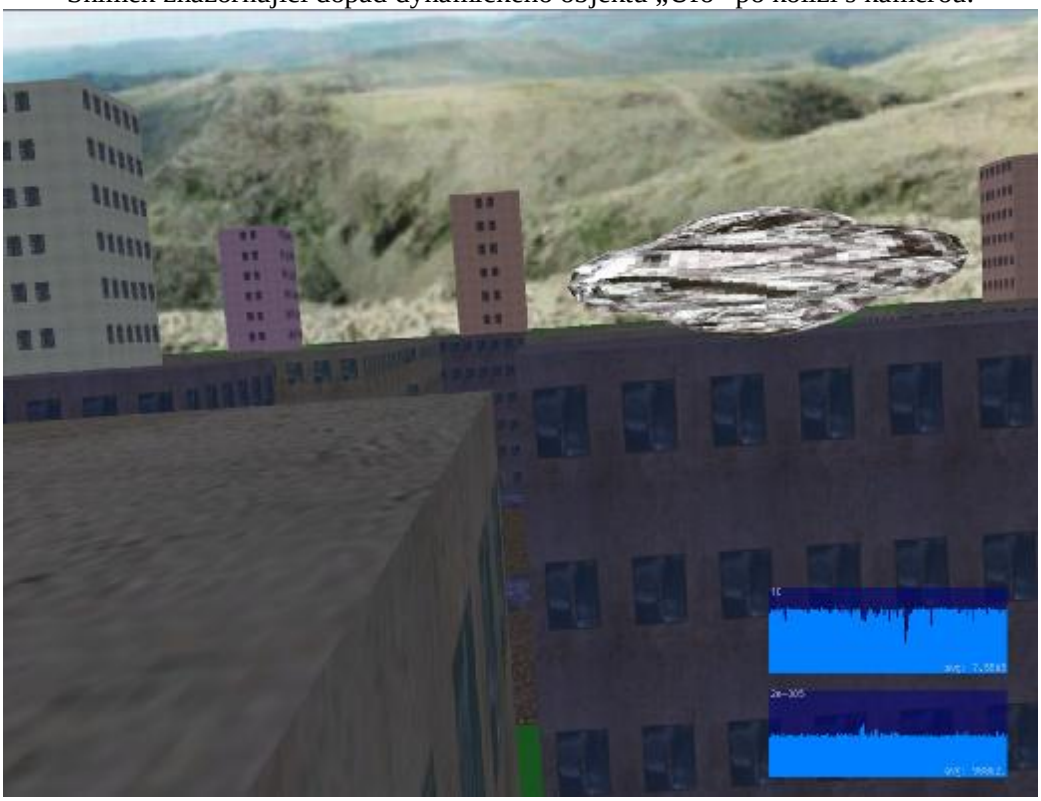
Obrázek 25.

Tvorba modelu „Ufo“



Obrázek 26.

Snímek znázorňující dopad dynamického objektu „Ufo“ po kolizi s kamerou.



Obrázek 27.

Programová příloha

Část kódu 1 - Vytvoření budovy pomocí sítě trojúhelníků

```
SoSeparator *root = new SoSeparator;

//Na vstupu máme plné rozměry domu a jsme na souřadnicích jeho těžiště.
// dx, dy, dz jsou vstupní souřadnice. K určení pozic vrcholů slouží BoxVertices

float ddx=dx / 2;
float ddy=dy / 2;
float ddz=dz / 2;

/*static*/ float BoxVertices[8][3] =
{
  -ddx, -ddy, -ddz,
   ddx, -ddy, -ddz,
  -ddx,  ddy, -ddz,
   ddx,  ddy, -ddz,
  -ddx, -ddy,  ddz,
   ddx, -ddy,  ddz,
  -ddx,  ddy,  ddz,
   ddx,  ddy,  ddz,
};

// indexy do BoxVertices pro jednotlivé vertexy, -1 ukončuje vždy jeden triangle-strip
// určuje též natočení trojúhelníků

static int32_t BoxIndices[6][5] =
{
  0, 1, 2, 3, -1,
  4, 0, 6, 2, -1,
  5, 4, 7, 6, -1,
  1, 5, 3, 7, -1,
  2, 3, 6, 7, -1,
  4, 5, 0, 1, -1,
};

// indexy pro texturovací souřadnice

static int32_t BoxTexCoordIndex[5] =
{
  0, 1, 2, 3, -1,
};
```

```
// zde přidáváme materiál. Color je vstupní proměnná

    SoMaterial *material = new SoMaterial;
    material->ambientColor.setValue(Color);
    material->diffuseColor.setValue(Color);
    root->addChild(material);

// Souřadnice pro rendrované trojúhelníky jsou předpřipravené ve BoxVertices.

  SoCoordinate3 *coords = new SoCoordinate3;
  coords->point.setValues(0, 8, BoxVertices);
  root->addChild(coords);

// Textury, pro každou stranu krychle máme jednu texturu

SoTexture2 *textures[6];
  for (int i=0; i<6; i++)
{
        textures[i] = new SoTexture2;
}

textures[0]->filename.setValue("okno.jpg");
textures[1]->filename.setValue("okno.jpg");
textures[2]->filename.setValue("okno.jpg");
textures[3]->filename.setValue("okno.jpg");
textures[5]->filename.setValue("strecha.jpg");

//  textures[4]->filename.setValue("podlaha.jpg");

  for (i=0; i<6; i++)
{
        if (i2==0)
        {
                SoTextureCoordinate2 *texCoord = new SoTextureCoordinate2;
                texCoord->point.set1Value(0, SbVec2f(0,0));
                texCoord->point.set1Value(1, SbVec2f(4*dx,0));
                texCoord->point.set1Value(2, SbVec2f(0, 4*dy));
                texCoord->point.set1Value(3, SbVec2f(4*dx,4*dy));
                root->addChild(texCoord);
        }
        else
        {
         SoTextureCoordinate2 *texCoord = new SoTextureCoordinate2;
        texCoord->point.set1Value(0, SbVec2f(0,0));
        texCoord->point.set1Value(1, SbVec2f(4*dz,0));
        texCoord->point.set1Value(2, SbVec2f(0, 4*dy));
        texCoord->point.set1Value(3, SbVec2f(4*dz,4*dy));
        root->addChild(texCoord);
        }
root->addChild(textures[i]);
```

```
// SoIndexedTriangleStripSet nám vyrendruje trojúhelníky.

SoIndexedTriangleStripSet *strip = new SoIndexedTriangleStripSet;
strip->coordIndex.setValues(0, 5, BoxIndices[i]);
strip->textureCoordIndex.setValues(0, 5, BoxTexCoordIndex);
root->addChild(strip);
```

Část kódu 2 - Zlepšení efektivity texturování

```
SoSearchAction sa;
sa.setType(SoTexture2::getClassTypeId());
sa.setInterest(SoSearchAction::ALL);
sa.setSearchingAll(TRUE);
sa.apply(root);
SoPathList & pl = sa.getPaths();
SbDict namedict;

for (int i = 0; i < pl.getLength(); i++) {
  SoFullPath * p = (SoFullPath*) pl[i];
  if (p->getTail()->isOfType(SoTexture2::getClassTypeId())) {
    SoTexture2 * tex = (SoTexture2*) p->getTail();
    if (tex->filename.getValue().getLength()) {
      SbName name = tex->filename.getValue().getString();
      unsigned long key = (unsigned long) ((void*) name.getString());
      void * tmp;
      if (!namedict.find(key, tmp)) {
        // Nová textura je přidána do listu
        (void) namedict.enter(key, tex);
      }
      else if (tmp != (void*) tex) { // Nahrazení uzlu
        SoGroup * parent = (SoGroup*) p->getNodeFromTail(1);
        int idx = p->getIndexFromTail(0);
        parent->replaceChild(idx, (SoNode*) tmp);
      }
    }
  }
}
sa.reset();
```

Část kódu 3 - Otočení kamery směrem nahoru

```
if (klavesy[Klavesa::nahoru].stisknuta) // Použije se pouze v případě stisknutí příslušné klávesy
  {
        nahoruVektor=-nahoruVektor;
        pomocnyNahoru=nahoruVektor;
        pomocnyDopredu=-dopreduVektor;
        pomocnyDopredu2=dopreduVektor*alfaDopredu;
        pomocnyNahoru2=nahoruVektor*alfaDostrany;
        dopreduVektor=pomocnyDopredu2+pomocnyNahoru2;
        dopreduVektor.normalize();

        pomocnyNahoru2=pomocnyNahoru*alfaDopredu;
        pomocnyDopredu2=pomocnyDopredu*alfaDostrany;
        nahoruVektor=pomocnyDopredu2+pomocnyNahoru2;
        nahoruVektor.normalize();
        nahoruVektor=-nahoruVektor;

        kamera->pointAt(kamera->position.getValue()+dopreduVektor,nahoruVektor);
        // nastavení kamery, aby se dívala směrem dopředu vektoru
  }
```

Část kódu 4 - Přidávání objektu do pole odkazů

```
CPole(int X, int Y)
        {
                this->X = X;
                this->Y = Y;
                for (int x = 0; x < X; x++)
                        for (int y = 0; y < Y; y++)
                        {
                                Plocha[x][y] = NULL;
                        }
        };

        virtual ~CPole();

        void PridejObjekt(int R, int S, GObject* GO, bool Hlavni)
        {
                Plocha[R][S] = GO;
                HlavniOdkaz[R][S] = Hlavni;
        }
```

Část kódu 5 - Přidání statického objektu z pole do grafu scény

```
void Zobraz(SoSeparator* root)
{
for (int x = 0; x < X; x++)
	for (int y = 0; y < Y; y++) // Procházení vygenerovaného pole
	{
		if (Plocha[x][y] != NULL)
		{
			GObject *GO = Plocha[x][y];
			if (HlavniOdkaz[x][y])
			{
				float dx = GO->x + GO->dx/2; // výpočet souřadnic k translaci
				float dy = GO->y + GO->dy/2;
				float dz = GO->z + GO->dz/2;

				// pozitivní translace - umístění ve scéně
				SoTranslation *trans = new SoTranslation;
				trans->translation.setValue(dx, -dy, dz);
				root->addChild(trans);

				root->addChild(GO->Vytvor()); // vytvoření grafiky stat. obj.

				// zpětná translace - návrat do bodu [0,0]
				SoTranslation *trans2 = new SoTranslation;
				trans2->translation.setValue(-dx, dy, -dz);
				root->addChild(trans2);
			}
		}
	}
};
```

Část kódu 6 - Tvorba okna prohlížeče a rozjetí renderovací smyčky

```
#ifdef _WIN32 // Platformová nezávislost
	SoWinRenderArea *renderArea = new SoWinRenderArea(window);// Tvorba okna
prohlížeče
#else
	SoQtRenderArea *renderArea = new SoQtRenderArea(window);
#endif

  renderArea->setSceneGraph(root); // Nastavení kořenu scény
  renderArea->setTitle("Urban Generator");
  renderArea->show();

  // Zobrazení okna a rozjetí renderovací smyčky
  #ifdef _WIN32
	SoWin::show(window);
	SoWin::mainLoop();
#else
	SoQt::show(window);
	SoQt::mainLoop();
#endif

// uvolnění prohlížeče scény z paměti
delete renderArea;
  root->unref();
```

Část kódu 7 - Generování silnic

V tomto algoritmu probíhá generování vodorovného směru silnice. Jak již bylo řečeno, pro kolmý směr je algoritmus stejný.

```
i=1;
i2=1;

while (i2<delka)        // postupuji po souřadnicích
        {
        while (i<sirka)
                {
                if (plocha[i][i2]!=3) // indikuje, zda mohu začít pokládat silnici
                        {
                        nahoda=(int)(((double)rand()/(double)(RAND_MAX+1)) * 100);
                        if ((nahoda<20) && (i<sirka-2))
                                {
                                        // 20% šance, že začne silnice
                                        // jedničky představují položení silnice
                                        // trojky, že na dané místo nemůže být
                                        // silnice položena

                                plocha[i][i2]=1;
                                plocha[i+1][i2]=1;
                                plocha[i-1][i2+1]=3;
                                plocha[i][i2+1]=3;
                                plocha[i+1][i2+1]=3;
                                i++;i++;

                        while ((80>(int)(((double)rand()/(double)(RAND_MAX+1)) * 100))
                                        && (sirka>i-1) && (plocha[i][i2]!=3))
                                                {

                                                // 80% šance, že silnice bude pokračovat

                                                plocha[i][i2+1]=3;
                                                plocha[i][i2]=1;
                                                i++;
                                                }// silnice dále nepokračuje
                                plocha[i][i2+1]=3;
                                }
                i++;
                }
        i=1;
        i2++;
        }
```

Část kódu 8 - Umístění domů

V tomto algoritmu probíhá umísťování domů. Dům, který zde umisťujeme, zabírá aktuální pole a pole v souřadnicích [x,y] a [x+1,y].

```
i=0;
i2=0;

while (i2<=delka)
        {
        while (i<=sirka)
        {
         if ((plocha[i][i2]==0)&&(plocha[i+1][i2]==0))
                {
                        // pokud je na ploše volné místo, mohu umístit tento dům

                        plocha[i][i2]=2;          // dům si značím číslem 2, v této verzi je to
                        plocha[i+1][i2]=2;        // zbytečné, ale v budoucnosti by se mohlo
                        for (int k=i-1;k<=i+2;k++)        // najít využití
                        {
                                if (plocha[k][i2-1]==0) plocha[k][i2-1]=3;
                                if (plocha[k][i2+1]==0) plocha[k][i2+1]=3;
                        }
                        if (plocha[i-1][i2]==0) plocha[i-1][i2]=3;
                        if (plocha[i+2][i2]==0) plocha[i+2][i2]=3;

                                // zde označím okolí domů, aby nedocházelo k „dotykům“
                                // sousedních budov

                          pozicex=(float)(i-1);
                          pozicey=(float)(i2-1);

                        Barak *Bar;               vytvoření objektu domu a nastavení parametrů
                        Bar = new Barak (pozicex, 0.0f, pozicey, 2.0f, 1.5f, 1.0f,true,
                        Barvicka[(int)(((double)rand()/(double)(RAND_MAX+1)) * 8)],"");

                                        // barva se domu přidává náhodně
                                        // (efekt duhového města)

                Pole.PridejObjekt (i-1,i2-1, Bar, true); // přidání objektu domu do pole
                }
                i++;
                }
        i=0;
        i2++;
        }
```

Část kódu 9 - Generování dynamických objektů jednoho typu

```
for (int i=1;i<=pocetobjektu;i++)
{
  // Nastavení nulové počáteční rychlosti
  vekturekPohybu.setValue(0,0,0);

  // Vytvoření dynamického objektu na náhodných souřadnicích v povoleném rozmezí
  letajiciObjekt *lo = new letajiciObjekt
  ((float)(((double)rand()/(double)(RAND_MAX+1)) * 25),
  (float)(((double)rand()/(double)(RAND_MAX+1)) * 3 - 10),
  (float)(((double)rand()/(double)(RAND_MAX+1)) * 25),0,0,vekturekPohybu,2,root);

 PocetLetajicichObjektu = PocetLetajicichObjektu +1;

// Vytvoření objemu tělesa ( souřadnice kdeJsem určují střed objektu )
SbBox3f *KrabiceLetajicihoObjektu =
        new SbBox3f(SbVec3f(lo->kdeJsemX-0.5f, lo->kdeJsemY-0.25f, lo->kdeJsemZ-0.f),
                    SbVec3f(lo->kdeJsemX+0.25f, lo->kdeJsemY+0.25f, lo-
        >kdeJsemZ+0.5f));

// Vytvoření OctTree objektu vkládaného do OctTree
OctreeObjects *LetajiciObjektOctree =
        new OctreeObjects(1,0.5f,1,lo->kdeJsemX, lo->kdeJsemY, lo->kdeJsemZ,
                          KrabiceLetajicihoObjektu, lo);

// Vložení OctTree objektu do OctTree
KorenOctree->PridejPrvek(*LetajiciObjektOctree);

// Vložení dynamického objektu do seznamu objektů, který je používán při fyzikálním pohybu
těles
OctreeObjekty.insert(OctreeObjekty.end(), LetajiciObjektOctree);
}
```

Část kódu 10 - Konstruktor dynamických objektů

```
letajiciObjekt(float pozicex, float pozicey, float pozicez, float rychlostObj, float zrychleniObj,
SbVec3f smerPohybu, int cojeto, SoSeparator* root)
{
JesteLetim = true; // proměnná sloužící k indikaci, zda objekt dospěl ke své první kolizi
zrychleni = zrychleniObj;
vektorPohybu = smerPohybu;
rychlostniVektorPohybu = smerPohybu;
typObjektu = cojeto;
rychlost = rychlostObj;
kdeJsemX=pozicex;
kdeJsemY=pozicey;
kdeJsemZ=pozicez;

stredObletu.setValue(pozicex, pozicey, pozicez);

// nyní se přidá grafika
SoSeparator* child = new SoSeparator;
transform = new SoMatrixTransform;
Koren=root;

maticeee.setTranslate(SbVec3f(pozicex,pozicey,pozicez));
transform->matrix.setValue(maticeee); // transformace sloužící k umístění objektu ve scéně

SoMaterial* mat = new SoMaterial;
SoCube* cube = new SoCube;
SoTexture2 *texture = new SoTexture2;
SoFile *model = new SoFile;

switch (cojeto) // rozlišujeme objekty podle typu
        {
   case 1:
                mat->diffuseColor.setValue((float)rand(),(float)rand(),(float)rand());
                texture->filename.setValue("texture/LetajiciObjekt1.jpg");

                cube->width.setValue(1);
                cube->height.setValue(0.25f);
                cube->depth.setValue(1);

                child->addChild(transform);
                child->addChild(mat);
                child->addChild(texture);
                child->addChild(cube);
                root->addChild(child);

                polomer=10; // nastavení poloměru po kterém se mají pohybovat
        break;



        case 2:
```

```
                mat->diffuseColor.setValue((float)rand(),(float)rand(),(float)rand());
                model->name.setValue("/models/Ufo1.wrl");
                child->addChild(transform);
                child->addChild(mat);
                child->addChild(model);
                root->addChild(child);

                polomer=5;

        break;
        default:
                model->name.setValue("/models/Ufo1.wrl");
        }
}
```

Část kódu 11 - Generování dynamických objektů

```
void GenerujKolizniObjekty(int typobjektu, int pocetobjektu, SoSeparator* root, OctreeNodes
*KorenOctree)
{

SbVec3f vekturekPohybu;

switch (typobjektu)
{
  case 1:
    for (int i=1;i<=pocetobjektu;i++)
    {
     vekturekPohybu.setValue(0,0,0);
     letajiciObjekt *lo = new letajiciObjekt ((float)(((double)rand()/(double)(RAND_MAX+1)) *
25),
                                (float)(((double)rand()/(double)(RAND_MAX+1)) * 3 - 10),
                                (float)(((double)rand()/(double)(RAND_MAX+1)) * 25),
                                 0,0,vekturekPohybu,1,root); // vytvoření dynamického objektu

PocetLetajicichObjektu = PocetLetajicichObjektu +1;

   SbBox3f *KrabiceLetajicihoObjektu = new SbBox3f( SbVec3f ( lo->kdeJsemX - 0.5f, lo-
>kdeJsemY-0.25f, lo->kdeJsemZ-0.5f ), SbVec3f( lo->kdeJsemX + 0.5f, lo->kdeJsemY +
0.25f, lo->kdeJsemZ + 0.5f) ); // vytvoření objemu dynamického objektu

OctreeObjects *LetajiciObjektOctree = new OctreeObjects( 1, 0.5f, 1, lo->kdeJsemX, lo-
>kdeJsemY,  lo->kdeJsemZ, KrabiceLetajicihoObjektu, lo); // vytvoření OctTree objektu.

KorenOctree->PridejPrvek(*LetajiciObjektOctree); // přidání OctTree objektu do OctTree

OctreeObjekty.insert(OctreeObjekty.end(), LetajiciObjektOctree);
// přidání dynamického objektu do vektoru objektů
 }

break;

  case 2:          // velmi podobné jako case 1, proto jej zde neuvádím
  default: ;
}

};
```

Část kódu 12 - Detekce kolizí kamery a reakce na tyto kolize

```
KorenOctree->ZrusPrvek(*KameraOctree); // Zrušení kamery v OctTree

SbBox3f *NovaKrabiceKamery = new SbBox3f(SbVec3f(koko1-0.05f,koko2-0.05f,koko3-
0.05f), SbVec3f(koko1+0.05f, koko2+0.05f, koko3+0.05f) ); // Vytvoření nového objemu
kamery

OctreeObjects *PokusnaKameraOctree = new OctreeObjects(koko1,koko2,koko3,
NovaKrabiceKamery); // Vytvoření nového objektu OctTree - kamery

	if ((KorenOctree->PridejPrvekAtestujKolize(*PokusnaKameraOctree))==NULL)
		{ // Kolize nenastává
			kamera->position = kamera->position.getValue() + rychlostniVektor;
			KameraOctree=PokusnaKameraOctree;
		}
	else
		{ // Kolize nastává
			rychlostniVektor=SbVec3f(0,0,0);
			KorenOctree->ZrusPrvek(*PokusnaKameraOctree);
			KorenOctree->PridejPrvek(*KameraOctree);
		}
```