VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PROCEDURÁLNĚ GENEROVANÉ MĚSTO

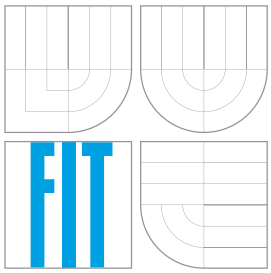
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK PAZDERA

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PROCEDURÁLNĚ GENEROVANÉ MĚSTO

PROCEDURALLY GENERATED CITY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK PAZDERA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RUDOLF KAJAN

BRNO 2011

Abstrakt

Bakalářská práce se zabývá procedurálním generováním měst. Prozkoumává již existující systémy a zabývá se používanými algoritmy a postupy v této oblasti. Dále popisuje návrh, implementaci a testování open-source systému pro procedurální generování měst, který vychází z těchto postupů. Součástí tohoto systému je knihovna libcity a demonstrační aplikace OgreCity. V závěru je zhodnocena implementace a proveden návrh možných rozšíření.

Abstract

This bachelor's thesis covers procedural generation of cities. It explores various existing systems that allow procedural generation of large urban areas. Some of the used approaches in this area are later discussed and explained. This thesis also describes design, implementation and testing of an open-source system for procedural generation of cities, which is based on the discussed techniques. The system consists of libcity library and a demo application OgreCity. At the end of this thesis is the current implementation evaluated and a few possible extensions are proposed.

Klíčová slova

procedurálně generované město, L-systémy, libcity, počítačová grafika, C++, OGRE 3D

Keywords

procedurally generated city, L-systems, libcity, computer graphics, C++, OGRE 3D

Citace

Radek Pazdera: Procedurálně generované město, bakalářská práce, Brno, FIT VUT v Brně, 2011

Procedurálně generované město

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Kajana.

.....
Radek Pazdera
13. května 2011

Poděkování

Chtěl bych poděkovat především vedoucímu mé práce, Ing. Rudolfu Kajanovi za odbornou pomoc a rady při řešení mé práce a také za čas, který mi věnoval.

Obsah

Kapitola 1

Úvod

Tvorba realistických modelů je v dnešním herním a filmovém průmyslu podstatnou součástí úspěchu. S rostoucím výkonem výpočetních zařízení rostou i naše možnosti zobrazování scén s velmi vysokou úrovní detailu. Současně rostou i požadavky diváků a hráčů na realističnost a vizuální propracování celého díla.

Modelování rozlehlých prostředí, například měst, do požadované úrovně detailu může být velmi časové náročný a také nákladný proces. Dále je třeba si uvědomit, že prostředí většinou není ústředním prvkem díla a bylo by tak velmi nevýhodné věnovat významnou část rozpočtu projektu do této oblasti na úkor jiných.

Zábavní průmysl však není jedinou oblastí, kde je poptávka po tvorbě virtuálních měst. Dalším polem mohou být například simulace, či geografické informační systémy.

Tato práce má za cíl prostudovat existující postupy procedurální tvorby vizuálního 3D modelu města a ze shromážděných poznatků vybrat vhodný způsob generování jednotlivých částí města. Nastudované poznatky následně využít je pro implementaci demonstrační aplikace, která předvede možnosti, jenž skýtají zvolené algoritmy.

Výsledkem této práce bude sada volně dostupných nástrojů, na které mohou stavět další vývojáři, kteří se zabývají oblastí modelování měst.

Kapitola 2

Související práce

V následující kapitole jsou popsány některé existující projekty z oblasti procedurálního generování měst. Dále jsou zde shrnuty práce, které se zabývají tímto tématem a popisují různé přístupy k řešení tohoto problému.

2.1 CityEngine

Jednou z existujících aplikací určenou k procedurálnímu generování měst je *CityEngine* [16]. Aplikaci prezentovali Parish a Müeller [15] v roce 2001 na konferenci SIGGRAPH. Systém je stále vyvíjen a aktualizován (aktuální verze je 2010.3). Za vývojem stojí švýcarská softwarová společnost *Procedural Inc.* Při jeho vývoji bylo publikováno několik článků a studií na dané téma. Autoři v nich popisují použité postupy a algoritmy. Některé z nich jsou zmíněny níže v podkapitole 2.3. Program je distribuován pod několika komerčními licencemi.

CityEngine vytváří města na základě hierarchicky uspořádané skupiny pravidel, která může být dále rozšiřována uživatelem [15].Vstup od uživatele probíhá ve formě obrazových dat, které specifikují terén, do kterého bude město situováno, rozložení populace ve městě, územní uspořádání, výšku budov apod.

Systém nejprve vytváří síť hlavních silnic za použití L-systémů (L-systémy jsou detailně probrány v kapitole 4). Hlavní silnice (dálnice) propojují místa s nejvyšší hustotou osídlení (ta je specifikována uživatelem pomocí mapy). Dále jsou vytvářeny vedlejší silnice a ulice. Hustota ulic vychází také z hustoty populace v dané oblasti – čím větší osídlení oblasti, tím více je zde vygenerováno ulic. Po vytvoření ulic jsou mezi nimi vygenerována místa, kam budou umístěny budovy.

Generování budov probíhá také pomocí L-systémů. Axiomem (viz kapitola 4) je kvádr, který je určen plochou místa, kde má být budova umístěna a maximální výškou budovy v dané oblasti (výšku budov určuje uživatel pomocí vstupních dat). Budova je vygenerována uvnitř této „krabice“.

CityEngine je v současné době asi nejznámějším a pravděpodobně i nejúspěšnějším produktem v oblasti procedurální tvorby měst. Systém dosahuje velmi dobrých výsledků, především z pohledu realističnosti a přesvědčivosti generovaných modelů. Nevýhodou je jeho uzavřenost a nedostupnost zdrojových kódů, které by umožnily vývojářům dále upravovat program podle vlastních představ.

Další určitou nevýhodou tohoto systému jsou jeho požadavky na uživatelský vstup. Minimální vstup od uživatele, jenž je požadován k vygenerování města je geografická mapa území, kam bude město situováno. V uvedených příkladech bylo využito velké množství

dalších geostatických údajů. Závislost dobrých výsledků systému na sběru takových dat není žádoucí. [5]

2.2 Citygen

Další aplikace, *Citygen* [20], vznikla na půdě Institute of Technology Blanchardstown. Architektura systému je popsána v přízpěvku z Game and Design Technology Workshop z roku 2007 [6]. Aplikace v mnohém vychází z CityEngine (viz 2.1). Postup vytváření města je podobný – nejprve je vytvořena síť hlavních silnic, která rozdělí prostor do menších oblastí. V těchto oblastech (tzv. „buňkách“ města) jsou nadále vytvářeny sítě ulic a do nich umisťovány budovy.

Odlišnost od CityEngine spočívá ve vstupu od uživatele. Citygen nepracuje s geografickými daty, ale síť hlavních komunikací je definována určením bodů, ve kterých následně budou křižovatky velkých komunikací. Silnice mezi body jsou automaticky vygenerovány (namapovány na terén). Ulice jsou generovány pomocí L-systémů uvnitř každé „buňky“ města.

Při tvorbě budov je nejprve vygenerován půdorys. Ten je upraven tak, aby se vešel do místa pro budovu ve městě. Následně je vytlačen nahoru a tím vznikne budova. Ostatní geometrie je řešena pomocí normálových map a textur.

Program však není volně dostupný (stahování aplikace bylo zakázáno, kód je majetkem ITB [4]).

2.3 Undiscovered City

V roce 2003 prezentoval Stefan Greuter v publikaci „Real-time Procedural Generation of 'Pseudo Infinite' Cities“ [7] přístup umožňující vytvoření „nekonečně“ rozlehlého města. Tato práce je zaměřena především na optimalizaci paměťové a výpočetní náročnosti procedurálního generování měst v reálném čase a částečně zanedbává vizuální přesnost modelu města. K optimalizaci generování jsou použity různé techniky – například zobrazování a uchovávání v paměti pouze budov, které jsou v daný moment viditelné. Dále je implementována LRU vyrovnávací paměť (least recently used). Zde jsou uloženy budovy, které uživatel nedávno přešel, pro případ, že by se ještě rozhodl vrátit. Tím je předcházeno opětovnému, zbytečnému generování.

Tvorba sítě ulic a silnic je zjednodušena do prosté, pravidelné mříže (rastru). Do vzniklých buněk jsou umístěny budovy. Generování budov probíhá sjednocováním geometrických primitiv. Fasáda je poté tvořena texturami. Vzhled budovy je řízen počátečním nastavením (seed) generátoru pseudo-náhodných čísel. Každé pole v rastru má přiřazenu hodnotu pro generátor. Pokud se tedy uživatel bude vracet zpět na stejné místo, nalezne na něm vždy stejnou budovu.

Výsledné modely s jistým úspěchem připomínají města, avšak nelze mluvit o přílišné realističnosti. Především kvůli rastrovému rozložení ulic, které nereflektuje rozložení ve skutečných městech a dávají městu „umělý“ nádech [5]. V systému je také pouze 20 typů textur pro fasády domů. Pro některé aplikace však může být systém dostatečný. Výhodou jsou velmi nízké nároky na hardware. Program dokáže pracovat v reálném čase při zachování stabilní snímkové frekvence – 20 FPS při zobrazení 500 budov [7] (měřeno na běžném PC v roce 2003).

Další práce

Kromě výše uvedených kompletních systémů pro procedurální generování měst existují práce, které se zabývají menšími oblastmi v rámci tohoto tématu. Hodně prací se zabývá různými přístupy k modelování budov. Peter Wonka představil v roce 2003 *split grammars* [21] pro procedurální generování budov z databáze pravidel. Pascal Müeller a Wonka následně přišli s gramatikou *CGA Shape* [14], která řeší některé problémy v předchozích přístupech. V roce 2010 prezentoval Simon Haegler novou gramatiku *F-Shade* [8]. Ta přináší opatimalizaci generování fasád domů a je určena pro renderování celých měst v reálném čase.

Jiří Koutný představil rozšíření L-systémů, tzv. kontextovou podmínku, která zamezuje vzniku řetězců, které vedou při generování budov k nereálným výsledkům [11].

Generováním sítě silnic podle obrysů budov se zabývá článek *Street generation for city modeling* [2]. Podobně také článek *Template-based generation of road networks for virtual city modeling* [18], který řeší generování ulic jednodušším způsobem, než jsou L-systémy. Také se zabývá tvarem silnic a jejich mapováním na terén.

Komplexní průzkum oblasti procedurálního generování měst zpracoval George Kelly [5]. Vyhodnocuje zde různé techniky a přístupy z několika hledisek (realističnost, efektivita, vstupní požadavky apod.).

Kapitola 3

Model města

V této kapitole je rozebrán model města a jeho tvorba. Přitom jsou diskutovány existující způsoby a algoritmy, které byly k tomuto účelu v minulosti použity.

Města jsou velmi komplexní systémy po své funkční i vizuální stránce. Každé z nich je odrazem svého historického, kulturního, ekonomického a sociálního vývoje v čase [15]. Při modelování měst se potýkáme s velkou různorodostí jednotlivých částí města způsobenou právě jeho vývojem. Jde o rozložení sítě ulic, čtvrtí a částí města. Liší se typy domů v různých čtvrtích, umístěním parků a podobně. Našim cílem je popsat tento složitý systém co nejjednodušší množinou pravidel, při zachování co největší realističnosti a tím umožnit procedurální generování modelu tohoto systému.

3.1 Procedurální modelování

Procedurální modelování je proces tvorby modelů systémů či entit, například geometrie objektů nebo textur na základě algoritmu. Model je uložen ve formě sekvence instrukcí k vytvoření statických dat popisující model, namísto těchto dat samotných.

Procedurální modelování lze dále dělit na tři základní podtřídy. První z nich jsou metody používáne v CAD (*Computer Aided Design*) a CAGD (*Computer Aided Geometric Design*), například šablonování, či generování ploch a křivek. Druhou podtřídu tvoří automatické generování objektů. Procedurální techniky jsou části kódu či algoritmy, které určují jisté charakteristiky počítačového modelu či efektu. Tuto podčást procedurálního modelování můžeme opět dělit. První skupinou jsou algoritmy, které vycházejí z gramatik – patří sem především *L-systémy* používané zejména pro generování rostlin. Druhou skupinou je *fraktální geometrie*, ta poskytuje algoritmy pro generování hor, krajin, kamenů, korálů a tak dále. Třetí skupinou jsou tzv. *částicové systémy* (*particle systems*), které se používají především pro generování explozí, hejn ptáků, simulaci ohně a podobně. [11]

Mezi hlavní výhody procedurální počítačové grafiky patří možnost parametrizace procesu tvorby modelu a tedy zásahu do toho, jak bude výsledná entita vypadat, kde bude umístěna a podobně. Toto je velmi výhodné za předpokladu, že máme velké množství instancí objektů, jenž patří do malého množství tříd.

Ve městě jde například o budovy. Každé město je tvořeno velkým množstvím domů a staveb. Stavby si jsou podobné, ale nikdy ne úplně stejné. Bytové panelové domy se sice liší barvou, počtem pater, ale tvarem jsou si podobné. Také vždy mají okna, dveře a balkóny. My můžeme využít těchto vlastností a definovat sekvenci instrukcí takovým způsobem, abychom z jedné relativně malé množiny pravidel byli schopni vykreslit tisíce panelových

domů různých výšek a barev pomocí jednoho algoritmu pouze změnou parametrů.

3.2 Etapy tvorby virtuálního města

Modelování města můžeme rozdělit do několika po sobě jdoucích fází. V minulosti byly použity dva různé přístupy. Décoret a Sillion prezentovali algoritmus, jehož vstupem jsou půdorysy budov ve městě a výstupem síť ulic daného města [2]. Bylo by možné nejprve vygenerovat budovy a ty pak uspořádat do topologie města podle jejich vlastností (velikosti půdorysu, typu a výšky budovy – seskupovat budovy podobného typu do čtvrtí).

Problémem tohoto přístupu je jistá nepřirozenost a s tím spojené omezené možnosti konfigurace celého procesu. Postup je vhodný spíše k automatickému vytváření grafu ulic existujících měst na základě obrazových dat, než k procedurálnímu generování celého města.

Další variantou je přirozenější přístup, kdy nejprve vytváříme silnice a ulice (*street graph*), které definují podstatnou část charakteru města. A teprve poté se zabýváme tvorbou budov. Lze kombinovat různé postupy a konfigurace v různých oblastech a dosáhnout tak rozmanitějších výsledků.

V této práci se budeme dále zabývat pouze druhou variantou. Proces tvorby virtuálního města můžeme rozdělit do tří fází:

- Generování sítě silnic a ulic (3.4)
- Výběr vhodných míst pro umístění budov (3.6)
- Generování budov samotných (3.7)

V prvních dvou etapách tvorby města je definována jeho *topologie*. Je generována síť silnic a následně jsou vytvářeny a dále děleny parcely pro umístění budov. V těchto stádiích se nezabýváme vizuální stránkou města. Ta je naopak velmi těsně spjata s etapou třetí – s generováním budov. Detailnější rozbor jednotlivých fází je uveden níže.

3.3 Přizpůsobení okolnímu prostředí

Jako první, ještě před samotnou tvorbou modelu, se musíme seznámit s okolním prostředím, kde bude město umístěno a při generování brát v úvahu omezení z něj plynoucí. Nejjednodušší, avšak v praxi málo viditelné je umístění města na dokonale rovnou plochu. Tímto se problém prvních dvou fází generování zjednoduší pouze do dvou dimenzí, protože odpadá nutnost řešit adaptaci silnic k terénu.

Rozhodneme-li se stavět město do prostředí, jenž je tvořeno zvlněným terénem na místo plochy, máme k dispozici dva různé postupy, jak dosáhnout dobrých výsledků:

- *Přizpůsobení vytvářeného města terénu* – Terén je považován za neměnný, musíme přizpůsobit město, které stavíme. Vygenerujeme 2D silniční síť a následně na ni aplikujeme adaptační algoritmy, jenž zachovávají křižovatky (místa, kam silnice vedou), ale snaží se nalézt takovou cestu terénem, jenž je nejvýhodnější (existují různé postupy – například cesta nejmenšího stoupání). Tento přístup je oproti níže popsanému přirozenější, ale podstatně složitější na implementaci. Pokud chceme vygenerovat město do předem daného prostředí, tak také jediný možný.

- *Přizpůsobení terénu městu* – Chceme-li generovat realistický model města a zároveň s tím generujeme i terén (není tedy neměnný), můžeme si dovolit opačného zjednodušení. Jde o „srovnání“ terénu podle potřeb vygenerované 2D mapy pozemních komunikací. Při jejich vykreslování provádíme kontroly a příliš strmý terén vyrovnáme pomocí aritmetického průměru s okolím.

Velmi častým fenoménem je také blízkost města k nějakému vodnímu tělesu. Především starší města s delší historií ve velké většině případů leží přímo okolo a nebo velmi blízko řece, jenž sloužila v dřívějších dobách jako hlavní zdroj pitné vody pro obyvatelstvo. Dalším příkladem mohou být přístavní města, jenž leží přímo na mořském pobřeží.

Tyto parametry prostředí je nutné brát v potaz a zajistit, aby nedocházelo například k stavění měst „na vodě“, je-li v okolí nějaký vodní tok, nádrž či moře. Dalším problémem, jenž vodní toky v krajině přinášejí je stavění mostů. Pro mosty je nutno definovat speciální pravidla.

3.4 Síť pozemních komunikací ve městě

Ústřední problém, se kterým se potýkáme při vytváření rozlehlých modelů měst je tvorba ulic a silniční sítě. Na tento fakt upozornili Parish a Müeller [15]. V silniční síti se odráží mnoho aspektů, jenž ovlivňují charakter města. Hustota silniční sítě roste s hustotou osídlení dané oblasti. Velkou měrou je také ovlivněna územním plánováním v konkrétním městě. Pozorováním sítí komunikací v reálných městech lze vysledovat několik vzorů, kterými se řídí [15]:

- **Pravidelné obdélníkové uspořádání** – Tento vzor silniční sítě se vyskytuje především v nových částech města, které byly plánovány a stavěny nárazově. Silnice v takovýchto oblastech bývají uspořádány rovnoběžně v pravidelných intervalech. Křižovatky jsou pravoúhlé a síť pak připomíná šachovnici. Asi nejznámějším městem, kde převládá tento vzor je americký New York.

- **Organické uspořádání** – Převládá ve starších městech a historických částech měst. Síť ulic je výsledkem postupného historického vývoje města. Ulice jsou uspořádány nepravidelně. Tento vzor se často vyskytuje v evropských městech, například v Paříži, v Praze.

- **Kruhové uspořádání** – Uspořádání ulic do soustředných kruhů nebývá příliš časté. Vyskytuje se v historických centrech některých starších měst (například Paříž) okolo významných oblastí (náměstí, apod.).

Silnice se výrazně liší svou velikostí – dálnice, městské okruhy, až po jednosměrné silnice a slepé uličky v obytných částech města. Modelování takové škály komunikací by bylo velmi složité. Dosavadní práce [15] [6] ukazují, že pro účely vizuálního modelu lze bez větší újmy na realističnosti zjednodušit silniční síť na dva typy komunikací:

- **Hlavní silnice** – Do této kategorie spadají všechny dálnice, víceproudé silnice, městské okruhy a podobné dopravní tepny.

- **Vedlejší silnice** – Všechny ostatní menší pozemní komunikace.

(a) New York (b) Paříž (c) Praha

Obrázek 3.1: Sítě silničních komunikací ve velkých městech.

3.4.1 Hlavní silniniční síť

Síť velkých silnic hraje ve městě důležitou roli především při určení jeho hrubých rysů (tvaru, rozlohy, velikosti apod.). Velké dopravní tepny spojují hustě obydlené městské části, mezi kterými dochází k přesunům velkého množství obyvatel (nemá smysl stavět dálnici, pokud po ní nikdo nebude jezdit).

Způsoby tvorby hlavních silnic jsou různé. Parish a Müeller [15] vychází právě z hustoty obyvatelstva a modelují tuto síť tak, aby propojovala nejhustěji osídlené oblasti města (hustotu osídlení oblasi určuje obrazová mapa). Kelly a McCabe [6] oproti tomu dávají možnost uživateli interaktivně zadat body, jenž budou tvořit křižovatky hlavních silnic.

Další možností pro prvotní definici charakteru města je postup opačný. Místo stavění hlavních komunikací podle hustoty osídlení můžeme vygenerovat silnice náhodně a v jejich okolí pak umisťovat hustěji obydlené oblasti. Tento postup je výhodný v situaci, kdy chceme vytvořit město, které vyplní přesně definovanou oblast v terénu. Určíme hranice města a hlavní silnice necháme vygenerovat generátorem (většinou s organickým vzorem).

Obecně se dá říct, že síť hlavních silnic lze určit množinou bodů v rovině (získanou nějakým způsobem od uživatele). Tuto množinu je možné generovat i pomocí generátoru pseudonáhodných čísel pro vytvoření čistě náhodné topologie města.

3.4.2 Vedlejší silniční síť

Zatímco hlavní silnice propojují oblasti mezi sebou, vedlejší komunikace vedou mezi domy a tvoří ulice. Pro celkový charakter města je důležitý především vzor, jenž tyto komunikace sledují (viz 3.1). Různé městské části mohou být tvořeny rozdílnými vzory a rozdílnou hustotou silniční sítě.

Není nutné (ani žádoucí), aby měl uživatel plnou kontrolu nad tvorbou vedlejších silnic. Cílem je naopak co nejvíce tento proces automatizovat. Důležitá je však možnost parametrizace výsledku generování (hustota síťě, vzor apod.).

3.5 Městské části – čtvrtě

Další důležitou charakteristikou města je jeho dělení do územních částí a čtvrtí. Ve velké většině měst lze vysledovat jisté podobnosti mezi stavbami, jenž se nacházejí ve stejné čtvrti. Například předměstské obytné části města, sídliště, obchodní čtvrti, industriální

oblasti a další. Tato vlastnost může vyplývat například z územního plánování, kdy jsou naráz stavěna rozlehlá sídliště v jednotném architektonickém pojetí. V jiných čtvrtích může být ovlivněna sociální třídou obyvatel či už samotnou polohou čtvrti ve městě. Směrem ke středu přibývá obchodních a kulturně zaměřených budov na úkor obytných domů, továren nebo skladových komplexů. Dle oblasti se mění i vlastnosti silniční sítě, především hustota a vzor, který následuje (viz 3.4.2). Chceme-li dosáhnout co největší realističnosti modelu města, nelze tyto parametry zanedbat.

Jedním ze způsobů, jak zohlednit tyto vlastnosti a umožnit tak konfiguraci parametrů pro různé čtvrtě je využít dvouúrovňového modelu silniční sítě (3.4. Vygenerováním hlavních silnic je prostor rozdělen do několika oblastí, které jsou uzavřeny mezi komunikacemi. Právě tyto zóny lze využít právě jako čtvrtě. Jsou uzavřené s pevně danými hranicemi, jenž poskytují omezení pro generování vedlejších silnic použitím L-systémů. V každé oblasti můžeme použít jinou gramatiku a tak ovlivnit vzor, hustotu a libovolné další parametry výsledné sítě.

Při tvorbě budov jsou naopak pevné hranice nevýhodou. Ostrá linie mezi dvěma oblastmi není příliš typická. Toto se projeví nejvíce v centrech měst, kde se mísí historické budovy s nově postavenými, moderními obchodními domy či kancelářskými komplexy. Pro budovy jsou ideální oblasti s rozmazanými hranicemi tak, aby se mohly na okrajích prolínat.

Řešení tohoto problému spočívá v určení příslušnosti každé městské oblasti k určitému typu (předměstská oblast, sídliště, obchodní, industriální) pomocí fuzzy koeficientů. Tak dosáhneme možnosti prolínání zón a tvorbu smíšených čtvrtí. Zároveň zůstanou zachovány původní pevně určené hranice zón, jenž jsou výhodné při tvorbě silniční sítě.

3.6 Umístění budov – bloky

Nyní je vygenerovaná mapa primárních silnic a v každé městské oblasti je sekundární silniční síť, jenž dělí prostor města na menší území. Do nich chceme dále umisťovat objekty, jenž budou tvořit samotné město. Může se jednat o parky, náměstí, parkoviště, nejčastěji však půjde o budovy. Tyto části nazýváme *bloky*.

Vzniklé bloky jsou ještě příliš velké oblasti na to, aby byl do každé z nich vložen jeden dům. Je nutné ještě dále rozdělit na menší části pro každou budovu – parcely. K tomu byl úspěšně používán tzv. *subdivision* algoritmus, který představil Müeller [15]. Dále byl rozšířen o podporu nekonvexních oblastí [6].

Algoritmus postupně dělí oblast na menší části, dokud všechny nedosáhnou potřebných velikostí. Důležité je, že budovy jsou následně umístěny jen do těch dílů, které leží u silnice. Ostatní nejsou uvažovány a jsou označeny jako nevhodné pro umístění budovy.

Vytvořené parcely pro domy a zároveň maximální povolená výška budovy v dané oblasti definují hranici, kde může být vygenerována budova. Budova nemusí nutně zabírat celou parcelu (například rodinné domky se zahrádkou), avšak rozhodně nesmí přesáhnout přes „krabici“ do které je umístěna.

3.7 Generování budov

Poslední fází je tvorba samotných budov. Na každé parcele, která je k tomu vhodná (viz 3.6), budeme generovat budovu. Pro dosáhnutí dostatečné rozmanitosti města je nutné důležitá možnost variací mezi různými typy budov a také variace mezi budovami stejného typu. Například různé textury, různá členitost, výška, orientace a podobně. Takovýchto vlastností

lze dosáhnout použitím generovacích pravidel zapsaných v podobě gramatik. V minulosti byly k tomuto účelu používány různé techniky a také gramatiky.

V závislosti na použité metodě lze dosáhnout různých výsledků. Použitím ad-hoc technik lze dosáhnout dobrých výsledků, ale špatné budoucí rozšiřitelnosti. Chceme-li přidat další typ budovy, musíme napsat celý vykreslovací kód znovu. Tento postup by se při nárůstu počtu typů budov stal velmi náročným.

Použití formálních gramatik přinese v počátku více práce (kvůli implementaci přepisovacího systému). Toto úsilí vynaložené navíc se pak ale pozitivně projeví na možnostech modifikace a nastavení parametrů pro budovy. Po přidání nové budovy stačí definovat novou gramatiku.

3.7.1 L-systémy

Jednou z možností je použití L-systémů, stejně jako pro generování silnic. L-systémy slouží především k modelování růstu v otevřeném prostoru [21]. Tento princip sice není nejvhodnější pro tvorbu budov, ale lze aplikovat i na ně. Můžeme si představit, že budova při stavbě (generování) vyrůstá pomalu ze země. Proto jsou vhodné zejména k reprezentaci věžových a výškových budov s pevným půdorysem.

Parametrické L-systémy (viz 4.3) byly úspěšně použity pro generování budov v původní verzi projektu *CityEngine*. Pravidla použitých L-systémů umožňují transformace (zmenšení/zvětšení, posuny a rotace) základních geometrických útvarů. Pro střechy, antény a podobné detaily jsou použity předgenerované geometrické šablony.

Obrázek 3.2: Příklad generování budovy pomocí L-systému v aplikaci CityEngine [15].

Příklad gramatiky pro generování jednoduchého mrakodrapu je uveden na obrázku 4.6.2. Detailnější vysvětlení L-systémů a jejich aplikace naleznete v kapitole 4.

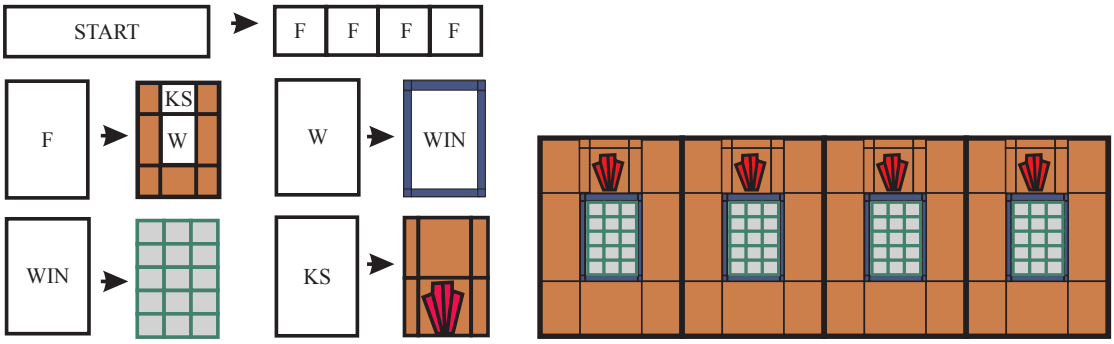
Použití L-systémů pro budovy se stává výhodným obzvláště pokud je používáme i pro generování silnic. Provedeme-li dostatečně obecný návrh přepisovacího systému, můžeme stejný kód použít i pro budovy a zjednodušit tak celý systém.

3.7.2 Split grammars

V roce 2003 prezentoval Wonka příspěvek *Instant Architecture* [21], ve kterém popisuje procedurální modelování budov a architektury pomocí nové gramatiky zvané *Split grammars*. Základním prvkem, s jimž pracuje tato gramatika je oproti ostatním podobným formálním modelům, které pracují s řetězci, *tvar*. Původní výzkum v této oblasti provedl Stiny [17] a split grammars vychází právě z jeho gramatik založených na tvarech (*shape grammars*).

Při generování budov dochází k postupnému *dělení* základních tvarů pomocí aplikace přepisovacích pravidel a nahrazování tvary menšími. Aplikovaná přepisovací pravidla jsou vybírána z databáze pravidel.

Tento koncept dělení byl převzán a zaveden také do L-systémů [11]. Detailnější popis operací *rozdělení* a *opakování* je v kapitole o L-systémech (4.6.2).



(a) Přepisovací pravidla (b) Výsledek generování

Obrázek 3.3: Příklad gramatiky typu split grammar. [21]

3.7.3 Gramatika CGA Shape

Jde o typ gramatiky, která byla vyvinuta přímo pro účely procedurálního generování budov vysoké vizuální kvality a s velkou úrovní detailu. CGA shape prezentoval Pascal Müeller v roce 2006 na konferenci Siggraph [14]. Aplikací přepisovacích pravidel této gramatiky je generovaná budova postupně rozvíjena do stále větších detailů.

V případě generování budov je nejprve vytvořen hrubý tvar budovy (*mass model*). Ten je dále strukturován, je vytvářena fasáda domu a nakonec detaily jako okna, dveře, různé ornamenty a podobně. Hlavní výhodou této metody je, že tvorba hierarchické struktury a popisu modelu je specifikována již v průběhu modelování [14].

Dřívější postupy (L-systémy, split grammars) umožňovaly efektivní tvorbu budov pouze základními geometrickými tvary. Ostatní detaily fasády byly nahrazeny texturami a shadery.

Obrázek 3.4: Příklad budovy generované pomocí *CGA shape* [13].

Kapitola 4

L-systémy

V této kapitole je vysvětlena gramatika zvaná *L-systém* (neboli *Lindenmayer system*). L-systémy dovolují na základě velmi malého množství vstupních dat vytvářet relativně složité struktury. Následná geometrická interpretace generovaných řetězců poskytuje zajímavé možnosti modelování. Informace v této kapitole týkající se L-systémů, definice, typy a jejich vlastnosti jsou volně převzaty z diplomové práce Jiřího Koutného, zabývající se L-systémy a jejich aplikací [11].

Gramatiku zavedl v roce 1968 maďarský biolog Aristid Lindenmayer jako teoretický rámec pro studium buněčných struktur. Tento rámec je však natolik obecný, přestože generované struktury mají buněčný charakter, nemusejí být nutně rostlinné. Přidání geometrické interpretace pomocí tzv. želví grafiky umožnilo realistické zobrazení modelů založených na L-systémech. [11]

Definice 1. *0L-systém.* 0L-systém je uspořádaná trojice $G = \langle V, \omega, P \rangle$, kde V je *abeceda symbolů*, $\omega \in V^+$ je neprázdná posloupnost symbolů zvaná *axiom* a $P \subset V \times V^*$ je *konečná množina přepisovacích pravidel*. Pravidlo $(a, \chi) \in P$ píšeme jako $a \to \chi$ a pravidla aplikujeme paralelně. 0L-systém je *deterministický* tehdy a jen tehdy, když pro každé $a \in V$ existuje právě jedno $\chi \in V^*$ takové, že $a \to \chi$. [11]

4.1 Kontextové L-Systémy

V 0L-systémech probíhá přepisovací proces bez ohledu na kontext, ve kterém levá strana pravidla (předchůdce) leží. Později byly zavedeny různé rozšíření L-systémů, které při aplikaci přepisovacích pravidel berou ohled na kontext.

- *1L-systémy:* Přepisovací pravidla mají jednostranný kontext, jejich formát je $a_l < a \to \chi$ nebo $a > a_r \to \chi$. Symbol a může být přepsán na χ tehdy a jen tehdy, pokud je předcházen symbolem a_l respektive následován symbolem a_r.
- *2L-systémy:* Pravidla mají oboustranný kontext a jsou ve formátu $a_l < a > a_r \to \chi$. Symbol a může být přepsán na χ pouze v případě, kdy ze předcházen symbolem a_l a zároveň následován symbolem a_r.

4.2 Stochastické L-Systémy

Všechny struktury generované jedním 0L-systémem budou stejné. Budeme-li pomocí 0L-systému, který popisuje strom, modelovat lesní porost, výsledkem bude les, kde budou

všechny stromy identické. Tato *umělá pravidelnost* působí nepřirozeně. Stejný příklad bude platný i pro budovy a město.

Potřebujeme tedy mechanismus, jenž zavede do procesu generování určitou náhodnost a umožní variaci detailů ve výsledku, ale zároveň zachová obecnou podobu generované entity. Této variace můžeme dosáhnout dvěma způsoby [11]:

- Zavedením náhodných jevů *do interpretace* generovaného řetězce: výsledný efekt je omezený, ale zachovává topologii výsledku. Stačí vygenerovat jen jeden řetězec symbolů.
- Zavedením náhodných jevů do *Struktury L-Systému*: stochastická aplikace pravidel může mít efekt jak na geometrii, tak na topologii výsledného objektu. Každé generování může vést k jinému výsledku.

Definice 2. *Stochastický 0L-systém.* Stochastický 0L-systém je uspořádaná čtveřice $G = \langle V, \omega, P, \pi \rangle$, kde V, ω a P jsou definovány stejně jako v 0L-systému (viz definice 1). Distribuční funkce pravděpodobnosti $\pi : P \rightarrow (0, 1\rangle$ mapuje množinu pravidel do množiny *pravděpodobností pravidel*. Pro každý symbol $a \in V$ se předpokládá, že součet všech pravděpodobnostní pravidel, které mají na levé straně a, je roven jedné. [11]

4.3 Parametrické L-Systémy

L-systémy umožnují generovat různé zajímavé objekty, ale jejich modelovací síla je omezena, protože číselné hodnoty jsou v L-systému zadány pevně. Číselnými hodnotami zde máme na mysli například hodnoty udávající délku či šířku generované úsečky nebo velikost úhlu rotace želvy (viz 4.5).

Řešením by bylo zavést speciální moduly pro každou myslitelnou hodnotu. Počet takovýchto modulů by v některých případech mohl narůstat nad únosnou mez (stovky, tisíce pravidel). Tyto moduly by se navíc lišily pouze v této jedné hodnotě, tedy v jednom *parametru*. Vznikaly by tak L-systémy se stovkami modulů a pravidel, čímž by se jejich specifikace stala obtížnou a jejich matematická krása by byla ztracena [11]. A. Lindenmayer navrhuje jiné řešení, které k symbolům v L-systému přidáva numerické parametry. Takto vzniklé L-systémy nazývá *parametrické*.

Definice 3. *Parametrický 0L-systém.* Parametrický 0L-systém je uspořádaná čtveřice $G = \langle V, \Sigma, \omega, P \rangle$, kde V je *abeceda* systému, Σ je množina *formálních parametrů*, $\omega \in (V \times R^*)^+$ je neprázdné *parametrické slovo*, které označujeme *axiom*, P je konečná *množina přepisovacích pravidel*. [11]

4.4 Otevřené L-Systémy

Otevřené L-systémy, přesněji řečeno nedeterministické kontextové parametrické L-systémy byly navrženy především pro potřeby simulace růstu rostlin. Jejich otevřenost spočívá v možnosti interakce s prostředím. Tato interakce je obousměrná, jak směrem do L-systému tak ven. Směrem dovnitř je možno například informovat přepisovací proces o detekci kolizí s překážkami. L-systém naopak může informovat okolí o svém rozložení v prostoru.

Otevřené L-systémy jsou parametrické bezkontextové stochastické L-systémy rozšířené o tzv. *komunikační moduly* tvaru

$$?E(x_1, \ldots, x_m),$$

které slouží k přenášení informací mezi posloupností modulů a okolím objektu, který L-systém reprezentuje. Před vlastním procesem přepsání je proveden mezikrok, ve kterém se získávají hodnoty skutečných parametrů komunikačních modulů. Na začátku tohoto kroku jsou skutečné hodnoty parametrů komunikačních modulů neznámé. Posloupnost modulů je nejprve prohlížena zleva doprava a vypočítává se stav želvy bez současné tvorby geometrického modelu. Při nalezení komunikačního modelu je vytvořena zpráva a předána prostředí. V této zprávě je obsažen dotaz na hodnoty parametrů. Okolí tyto parametry nastaví (stav želvy je znám) a L-systém pokračuje v prohlížení posloupnosti modulů. Jakmile jsou nastaveny parametry všech komunikačních modulů, začíná fáze přepisování modulů, která je shodná jako u parametrických L-systémů.

4.5 Interpretace generovaných řetězců

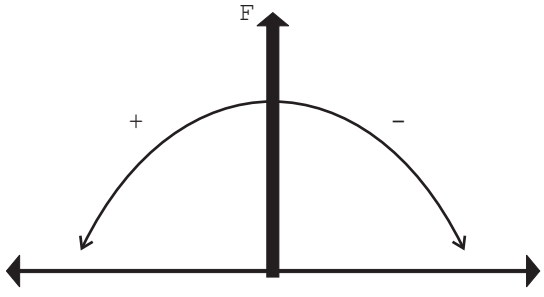
L-Systém je variantou formální gramatiky a pracuje tedy nad řetězci. K využití v počítačové grafice a k tvorbě vizuálních modelů je třeba zavést jistý způsob interpretace generovaných řetězců a vykreslování vizuální podoby modelu na základě čtení řetězce.

V dřívější době byly použity různé typy interpretace generovaných řetězců a transformace výsledků na vizuální modely. Jedním z přístupů je tzv. *želví grafika*, na kterou se zaměřil Przemyslaw Prusinkiewicz inspirován jazykem LOGO. Prusinkiewicz předvedl mnoho příkladů fraktálů a rostlinám podobných struktur modelovaných pomocí L-systémů za použití interpretace pomocí želví grafiky.

Pojmem želví grafika se rozumí vykreslování obrazu relativně od jednoho virtuálního kurzoru (želvy) nad kartézskou soustavou souřadnic. Želva má několik vlastností a je nad ni definováno několik základních operací. To se odvíjí podle toho v jakém se pohybujeme prostoru.

4.5.1 Želví grafika ve 2D

Definice 4. *Stav a příkazy želvy v rovině.* Stav pomyslné želvy je definován jako trojce (x, y, α), kde $[x, y]$ reprezenrují pozici želvy v kartézském souřadném systému a α je úhel označující směr, kterým se želva dívá, nazýváme ho hlava (heading). Dále je definována velikost kroku d a přírůstek úhlu δ. Želva dokáže reagovat na příkazy reprezentované těmito symboly v konvenční notaci:



Obrázek 4.1: Znázornění kurzoru pro želví grafiku v dvoudimenzionálním prostoru. [11]

F Posun vpřed o krok délky d. Změna stavu želvy na (x', y', α), kde $x' = x + d\cos(\alpha)$, $y' = y + d\sin(\alpha)$ a vykreslení úsečky mezi body $[x, y]$ a $[x', y']$.

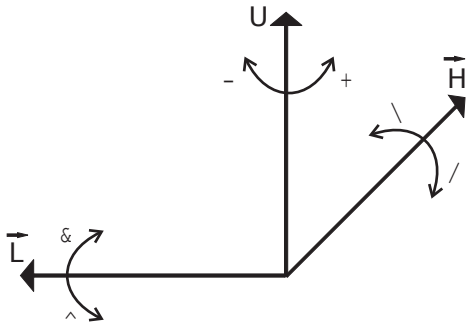
f Posun vpřed o krok délky d bez kreslení úsečky.

\+ Rotace doprava o úhel δ. Změna stavu želvy na $(x, y, \alpha + \delta)$

\- Rotace doleva o úhel δ. Změna stavu želvy na $(x, y, \alpha - \delta)$

4.5.2 Želví grafika ve 3D

Pro třídimenzionální prostor je situace složitější. Orientace želvy v prostoru je dána třemi vektory $\overrightarrow{H}$, $\overrightarrow{U}$ a $\overrightarrow{L}$ (vyobrazeny na obrázku 4.1). Tyto vektory spolu s polohou želvy tvoří její lokální souřadnicový systém. Jejich význam je:



Obrázek 4.2: Znázornění kurzoru pro želví grafiku v třidimenzionálním prostoru. [11]

$\overrightarrow{H}$ Směr, kterým se želva dívá a pohybuje dopředu (*heading*).

$\overrightarrow{U}$ Směr, ve kterém má krunýř (*up*).

$\overrightarrow{L}$ Směr, ve kterém má levou přední i zadní nožičku (*left*).

Kromě symbolů pro rotaci želvy v rovině jsou definovány další pro rotace nahoru, dolů a kolem podélné osy.

4.5.3 Závorkové rozšíření želví interpretace

Jednou z možností, jak generovat stromovou strukturu pomocí L-systémů je použití *řetězců se závorkami*. Zavádíme dva nové symboly pro vymezení větve a ty jsou želví grafikou interpretovány takto:

[Uložení aktuálního stavu želvy na zásobník. Uložená informace obsahuje polohu $[X, Y, Z]$ a orientaci $(\overrightarrow{H}, \overrightarrow{U}, \overrightarrow{L})$. Dále může obsahovat libovolné další informace, jako například barevné hodnoty a šířku kreslené čáry.

] Vyjmutí symbolu z vrcholu zásobníku a nastavení polohy, orientace a případně ostatních parametrů na hodnoty uložené v zásobníku (které odpovídaly stavu při interpretaci předcházejícího symbolu [).

4.6 Aplikace pro procedurální generování měst

Máme definován pojem *L-systém* od původních bezkontextových deterministických d0L-systémů přes k L-systémy kontextové, stochastické, parametrické až k otevřeným. Dále byl

Terminály:	$\langle road \rangle, +, -, [,]$
Neterminály:	E
Axiom:	E
Pravidla:	$E \rightarrow [[[-\langle road \rangle E] + \langle road \rangle E] \langle road \rangle E]$
Parametry:	úhel $= 90°$
	délka segmentu $= 100m$

Tabulka 4.1: Příklad jednoduché gramatiky pro rastrový vzor komunikací.

vysvětlen a předveden způsob jak podle řetězců generovaných L-systémy vykreslovat vizuální modely tzv. *želví grafiky*. Nyní zbývá popsat, jak jsou výše zmíněné nástroje využitelné v kontextu procedurálního generování měst.

L-systémy zde najdou uplatnění hned ve dvou oblastech, při generování sítě silnic a také při tvorbě budov.

4.6.1 Síť silnic

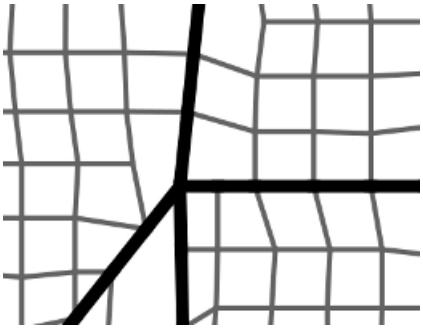
Jak již bylo řečeno výše, L-systémy jsou variantou *paralelních* přepisovacích systémů. Tato vlastnost je při tvorbě silniční sítě velmi důležitá. Díky tomu může síť narůstat paralelně v každé větvi. Gramatiky se sekvenční aplikací pravidel (Chomského gramatiky) by nejprve rozvíjely jednu větev a k generování dalších by došlo až by již nebylo možno generovat dál předcházející větev (například kvůli prostorovým omezením).

Dalším z požadavků pro generování silnic je možnost řízení a nastavení vzoru, jenž silnice v určité oblasti sledují (viz obrázek 3.1). Další důležitou vlastností je hustota silniční sítě, vzdálenosti křižovatek a podobně. Tyto možnosti poskytují parametrické L-systémy, kde můžeme nastavovat různým symbolům parametry. Změnou úhlu, o který se želva otáčí docílíme tvorby rozličných vzorů komunikací. Nastavení úhlu na 90° bude generovat pravidelný rastrový vzor, naopak úhel mezi 60° a 120° bude tvořit organické větvení silnic. Parametrem délka vykreslovaného segmentu upravíme hustotu vytvářeného vzoru.

Použitím stochastických L-systémů lze dosáhnout potřebných variací, aby vzor nebyl stejný v celém městě. Pokud je vyžadováno ve městě více vzorů, lze jej rozdělit na menší území a v každém použít generátor s jinou gramatikou, případně jinými parametry. Generování v jednotlivých oblastech je umožněno použitím otevřených L-systémů, kdy je kontrolováno zda model nepřesahuje přes vymezené hranice.

Posledním problémem, který nastává při generování silnic je příliš častý vznik slepých uliček. Slepé uličky jsou v reálných městech spíše výjmečné. Tento problém řeší zavedení kontextové podmínky (podle Koutného [11]) a pokud se právě vygenerovaný segment kříží s jiným již existujícím, či leží v jeho blízkosti, modifikujeme jeho parametry a nebo úplně odstraníme, pokud by narušoval integritu výsledné sítě.

Otevřené L-systémy s kontextovou podmínkou poskytují jednoduché řešení většiny problémů, kterým čelíme při generování silniční sítě a proto jsou velmi vhodným nástrojem pro tuto úlohu. Na obrázku 4.1 je příklad jednoduché gramatiky pro generování rastrového vzoru silniční sítě, jenž je použita v generátoru knihovny libcity. Výsledek vygenerovaný uvedenou gramatikou je na obrázku 4.3.



Obrázek 4.3: Příklad silniční sítě vygenerovaný pomocí knihovny libcity.

4.6.2 Budovy

Použití L-systémů pro budovy je o něco složitější. Nestačí pouze vykreslovat úsečky v rovině, ale musíme přejít ke kreslení tvarů v 3D prostoru.

Při generování lze postupovat několika způsoby. Nejjednodušší je situace u málo členitých výškových domů a věžových budov, jenž jsou typické pro centra velkých měst jako je New York či Hong-Kong. První způsob spočívá ve výstavbě budovy postupně po jednotlivých patrech. Generování ukončíme ve chvíli kdy narazíme na omezení definované obálkou (*bounding box*), jenž je vyhrazena pro každou budovu. Tento způsob se však hodí jen pro budovy s předem daným pevným půdorysem a velkou výškou. Protože nárůst objektu probíhá pouze do výšky, půdorys se nemění, není tato metoda schopna poskytnout dostatečnou variabilitu pro generování ostatních typů budov.

Chceme-li dosáhnout větších variací i v ostatních případech (skladové komplexy, rodinné domy, historické budovy), je nutné nejprve vytvořit objemový model pro budovu a ten dále upravovat do větších detailů. Tvorba základního tvaru budovy probíhá transformacemi a sjednocením několika základních geometrických útvarů (krychle, kvádr, válec). Tento vzniklý tvar je dále pomocí přepisovacích pravidel upravován do stále většího požadovaného detailu, je tvořena fasáda, střecha a podobně.

Pro provádění nahrazení obecnějších částí objektů za víc detailnějších (například nahrazení zdi za okna a dveře) byly zavedeny do L-systémů následující rozšiřující koncepty pro přepisovací pravidla. Tyto koncepty pracují tak, že dělí jeden tvar skupinou jiných. Jde o *prostorové dělení objektů*, *cyklické dělení objektů* a *relativní rozměry*. Tyto koncepty byly původně určeny pro sekvenční gramatiky založených na tvarech. Jde o gramatiky CGA Shape (viz 3.7.3) a Split grammar (viz 3.7.2), jenž jsou alternativou k L-systémům v oblasti procedurálního generování architektury.

Tyto koncepty nepracují s želví grafikou, ale s tvary. To s sebou přináší nutnost implementace dalších rozšíření pro interpretaci řetězců, jenž umožní nahrazování existujících objektů. Případně také dopočítávání šířky následníků, jejichž velikost byla definována pouze poměrem a podobně.

Prostorové dělení objektů

Tento typ pravidel umožňuje rozdělení modulu na levé straně přepisovacího pravidla (předchůdce) podél dané osy. Pravidla jsou pak ve tvaru

$$A \rightarrow div(axe, sizes)modules$$

kde

- A je předchůdce.
- *div* je klíčové slovo.
- *axe* je osa podél které bude probíhat dělení.
- *sizes* jsou *poměry* jednodlivých následníků oddělené čárkou.
- *modules* je seznam modulů oddělených čárkou, které vzniknou jako výsledek přepisovacího procesu. Jejich počet musí být shodný s počtem hodnot v *sizes*.

Pomocí pravidel tohoto typu jsme tedy schopni rozdělit modul na *předem známý* počet modulů, jejichž velikost je nastavena tak, aby dohromady zabíraly stejný prostor jako předchůdce.

Cyklické dělení

Cyklické dělení je druhou variantou, chceme-li rozdělit fasádu domu na okna, máme velikost okna, ale nevíme, kolik oken půjde umístit na jednu zeď dané budovy (velikost zdí se můžou lišit podle pořadí aplikace pravidel a nastavení parametrů konkrétních symbolů – při tvorbě pravidel gramatiky tedy není známo, jaký počet oken bude vhodný pro danou budovu). Pravidlo cyklického dělení má tvar

$$A \rightarrow div(axe, size)module$$

kde

- A je předchůdce.
- *div* je klíčové slovo.
- *axe* je osa podél které bude probíhat dělení.
- *sizes* udává *velikost* jednotlivých modulů.
- *modules* je použitý modul.

Terminály:	$\langle basement\rangle, \langle storey\rangle, \langle ledge\rangle, \langle rooftop\rangle, -$
Neterminály:	$\langle expansion\rangle$
Axiom:	$\langle basement\rangle\langle storey\rangle\langle expansion\rangle$
Pravidla:	$\langle expansion\rangle \rightarrow \langle storey\rangle\langle expansion\rangle$
	$\langle expansion\rangle \rightarrow -\langle rooftop\rangle\langle storey\rangle\langle expansion\rangle$
	$\langle expansion\rangle \rightarrow \langle ledge\rangle\langle storey\rangle\langle expansion\rangle$
	$\langle expansion\rangle \rightarrow \langle rooftop\rangle$

Tabulka 4.2: Příklad jednoduché gramatiky pro generování výškových budov.

Kapitola 5

Návrh aplikace

V následující kapitole je popsán proces návrhu projektu. V úvodu najdete obecné informace o koncepci projektu a způsobu jeho rozdělení na menší části. Dále je detailně popsán návrh každé části.

5.1 Koncept projektu

Při bližším pohledu na tvorbu města zjistíme, že vlastně děláme dvě na sobě nezávislé věci. Jde o *generování topologie* a *generování vizuální podoby* města.

Generováním topologie je myšlen proces určování toho, odkud kam povedou silnice, jaký budou mít tvar (tedy kudy povedou). Dále řešíme, kde budou umístěny budovy či jiné městské objekty jako parky, náměstí a tak dále. Bloky budov je třeba dělit na menší části, nevhodné odstraňovat, zkrátka pro podporu tohoto procesu je třeba implementovat velké množství různých postupů a algoritmů, jenž umožní procedurální generování města a budou velkou měrou ovlivňovat jeho vizuální charakter, ale jejich průběh vůbec nezávisí na výsledné vizuální podobě města. V této fázi je určeno odkud kam povede hlavní silnice, v jakém bodě bude protnuta silnicemi vedlejšími a kolik bude postaveno budov v okolí těchto silnic.

Konkrétní vizuální rozhodnutí provádíme až ve fázi druhé, při generování vizuální podoby. Při tomto procesu jsou zpracovávány výsledky algoritmů z předchozí fáze a na jejich základě je vykreslován výstup. Až nyní je specifikováno, jak budou vypadat konkrétní silnice a budovy, jaké budou použity textury a podobně.

Proces vykreslování je obvykle velmi těsně spjat s nějakou knihovnou či toolkitem, jenž pracuje s vykreslovacím hardware (například OpenGL, Direct3D). Pokud bychom později chtěli rozšířit aplikaci, či znovupoužít vytvořený kód pod jiným operačním systémem či jiným hardware, je dost velká pravděpodobnost, že by muselo dojít k přepsání veškerého vykreslovacího kódu. To by v případě promíchání s kódem pro tvorbu topologie nemusel být snadný úkol. Nabízí se tedy rozdělení projektu do dvou částí.

Oddělení topologické a vykreslovací části přinese i širší možnosti využití tohoto projektu. Vznikne možnost převzetí pouze samotné knihovny s algoritmy pro generování topologie města a napsání vlastních vykreslovacích algoritmů jenž budou interpretovat vygenerované výsledky a tvořit vizuální podobu města. Další výhodou je fakt, že uživatel knihovny (programátor vizuální části) má úplnou kontrolu nad vzhledem města, použitými texturami, typy a tvary domů, zón a podobně. Aplikace tak není omezena pouze ke generování jednoho typu města, například moderního. Napsáním nové vykreslovací části můžeme vytvořit

například středověké nebo naopak futuristické město pomocí stejných algoritmů.

Na základě těchto argumentů byla tvorba demonstrační aplikace (jak je specifikována v zadání) rozdělena na dvě části. První část – knihovna *libcity* obsahuje implementaci algoritmů určených k generování topologického charakteru města. Druhá část *OgreCity* využívá knihovnu *libcity* a demonstruje tak v ní obsažené algoritmy. Zároveň implementuje další, které už nelze oddělit od vykreslovaní.

V dalších sekcích této kapitoly je popsán návrh obou součástí.

5.2 Knihovna libcity

Knihovna *libcity* pokrývá první dvě etapy tvorby města, tvorbu sítě pozemních komunikací a tvorbu pozemků pro umístění budov. Tato kapitola popisuje návrh rozhraní knihovny a poté i dílčích částí a objektů.

5.2.1 Rozrhaní knihovny

Jedním z základních požadavků na návrh rozhraní bylo ponechat programátorovi, jenž bude knihovnu užívat, plnou kontrolu nad procesem tvorby města. Knihovna nabízí implementaci různých algoritmů a postupů, ale nenutí uživatele je používat. Návrh rozhraní umožňuje změnu kterékoliv etapy tvorby modelu města (viz 3.2), případně její úplné nahrazení jiným postupem.

Tento přístup může být problematický při prvním kontaktu programátora s knihovnou, protože nelze ihned generovat město pomocí jednoho příkazu, ale je nutno provést sérii nastavení parametrů. Na druhé straně je právě možnost natavení a konfigurace všech parametrů generování výhodou v případě, chceme-li definovat jisté charakteristiky města a procedurálně vygenerovat zbývající části. Může jít například o definici obrysu města, rozdělení na oblasti a podobně. K dispozici je možnost tvořit města podobající se některým existujícím či zcela náhodná.

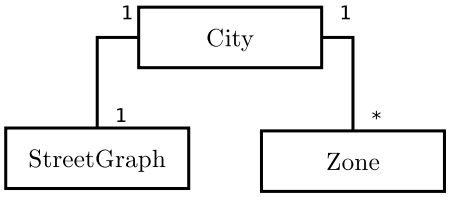
Rozhraní knihovny *libcity* je tvořeno abstraktní třídou `City`. Programátoři – uživatelé knihovny používají tak, že vytvoří vlastní třídu, jenž bude potomkem třídy `City` a implementují požadované virtuální metody. Tento návrh rozhraní splňuje všechny výše zmíněné požadavky na možnosti konfigurace.

Třída `City` propojuje dohromady mapu silnic (`StreetGraph`) s územními celky nejvyšší úrovně (`Zone`). Byly zvažovány různé varianty návrhu knihovny na této úrovni abstrakce. Například rozdělení mapy silnic pod příslušné zóny, kterými silnice procházejí, či rekurzivní dělení mapy přímo na menší části. Tyto přístupy dělení mapy nejsou vhodné, protože síť silnic ve městě tvoří jeden celek.

V nynějším návrhu (obrázek 5.1) jsou zóny města definovány nad tímto celkem. Chceme-li vytvořit silnici v určité oblasti a parametrizovat ji podle toho (různé druhy silniční sítě v různých čtvrtích) můžeme nahlédnout do seznamu zón a načíst parametry z něj.

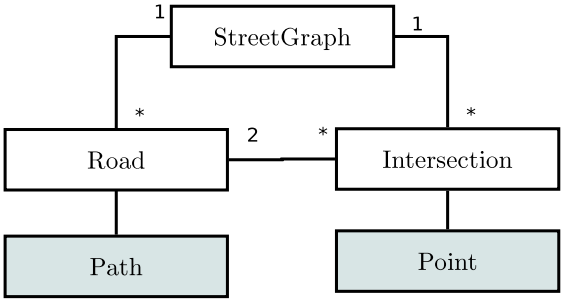
5.2.2 Mapa

Jednou částí problému generování topologie města je generování silnic. K tomu je v knihovně navrženo a implementováno několik nástrojů. Ústředním prvkem je třída `StreetGraph`, jenž slouží k uchování *planárního grafu* silnic. Její hlavní zodpovědností je zajistit, aby graf zůstal při přidávání silnic validní (správně provázaný) a planární.



Obrázek 5.1: Vztah třídy `City` k ostatním částem systému.

Graf je tvořen na dvou úrovních. První úroveň tvoří silnice a křižovatky, tedy třídy `Road` a `Intersection`. Tyto třídy jsou společně provázány. `Road` je hranou a `Intersection` uzlem v grafu. Tato část umožňuje orientaci v grafu na vyšší topologické úrovni a zjišťování, které silnice jsou propojené a na jakých místech se kříží. Vztah jednotlivých tříd je znázorněn na obrázku 5.2.



Obrázek 5.2: Vztah tříd tvořící dvouúrovňový graf silnic.

Na druhé úrovni je graf tvořen *geometrickou reprezentací* odpovídajících prvků. Jde o cestu, jenž vede každá silnice a body, kde leží křižovatky. Vytvoření grafu na těchto dvou úrovních umožňuje jeho efektivní procházení po úrovni první a zároveň možnost vzorkování na úrovni druhé. Je tedy možná tvorba různě zakřivených silnic pomocí více než dvou kontrolních uzlů, které by jinak zvyšovaly velkou měrou čas průchodu grafem.

Nad třídou `StreetGraph` pracují *generátory silnic*. Jde o třídy `OrganicRoadPattern` a `RasterRoadPattern` založené na L-systémech.

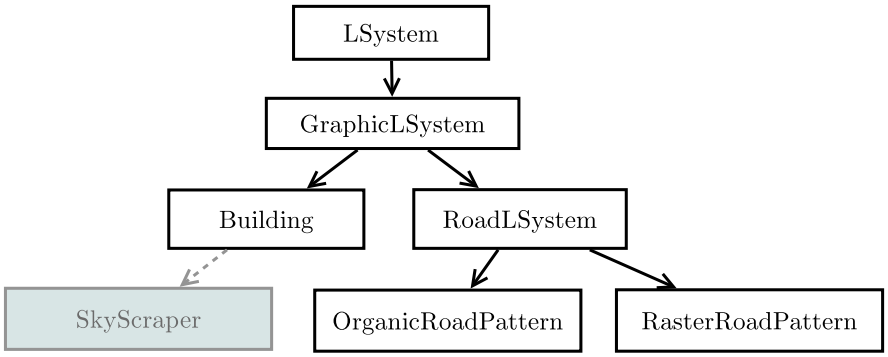
`StreetGraph` také nabízí možnost vytvářet síť pozemních komunikací různých typů. Může jít o hlavní a vedlejší silnice, dálnice a podobně. Typy lze dodatečně registrovat, proto záleží jen na programátorovi, kolik typů silnic bude ve svém městě potřebovat.

5.2.3 L-systémy

Návrh přepisovacího systému (*rewriting engine*) pro L-systémy by měl brát v potaz to, že jeho implementace bude používaná jak pro generování síťe silnic, tak i pro stavbu budov. Proto je potřeba jej udělat dostatečně obecný. Jestliže by se nám to nepodařilo, bylo by nutné implementovat přepisovací systém dvakrát, což by bylo zbytečné a neefektivní.

Pro použití v knihovně byla navrhnuta hierarchie tříd, jak je vyobrazena na obrázku 5.3. Bází ve stromu dědičnosti je třída `LSystem`. Zde je implementován pouze *stochastický parametrický* L-systém, který pracuje jen nad řetězci. Tato funkčnost je dále rozšiřována.

Třída `GraphicLSystem` implementuje rozšíření o želví grafiku (viz kapitola 4.5). Přidává do abecedy základní sadu symbolů, jímž želva rozumí a mohou být využity při tvorbě



Obrázek 5.3: Hierarchie tříd nad implementací L-systémů v knihovně libcity.

pravidel. Dále implementuje závorkové rozšíření želví interpretace, tedy zásobníkovou paměť stavů kurzoru (želvy). Z třídy `GraphicLSystem` vychází další dvě již specializované třídy pro generování silnic a budov.

`RoadLSystem` je základnou pro všechny generátory sítě komunikací v knihovně. Rozšiřuje `GraphicLSystem` o kontextovou podmínku generování silnic, tzv. snap algoritmus. Tento algoritmus řeší nevhodně vygenerované větve silniční sítě, buď provádí jejich modifikaci a nebo úplné zahození.

Třídy `RasterRoadPattern` a `OrganicRoadPattern` pouze rozšiřují o nejčastěji používané parametry a pravidla tak, aby L-systém generoval silnice v odpovídajících vzorech.

Poslední část stromu dědičnosti s kořenem v bázové třídě `LSystem`, třída `Building`, je pouze abstraktní. Rozšiřuje grafické L-systémy do třídimenzionálního prostoru a přidává další nový koncept – definici hranic pomocí obálky (*bounding box*). Všechny budovy musí být generovány uvnitř odpovídajících hranic. Implicitně se budova snaží zabrat co největší možnou oblast uvnitř obálky. Toto záleží na definici pravidel a jejich interpretace.

Bounding box je možné, stejně jako stav želvy ukládat a načítat ze zásobníku. Tím lze dosáhnout generování zajímavých tvarů.

Na obrázku 5.3 vyobrazená šedě, třída `SkyScraper` je také součástí stromu dědičnosti, ale již není součástí knihovny. Je definována pro vykreslování budov uživateli a její implementaci řeší až aplikace *OgreCity*.

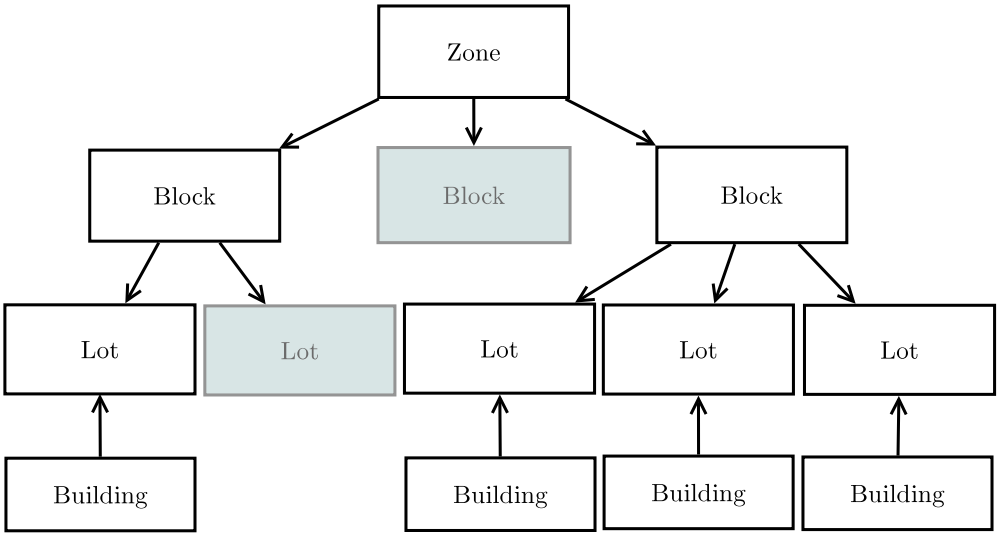
5.2.4 Územní dělení

Druhou částí problému generování topologie města je územní dělení prostoru mezi silnicemi a tvorba parcel pro umístění objektů. Zde je výhodné brát inspiraci u územního dělení reálných měst na městské části. Reprezentací územní části města je v knihovně *libcity* třída `Zone`. Městské části můžeme buď definovat ručně, chceme-li mít kontrolu nad tím, jak bude město v jednotlivých místech vypadat a nebo můžeme využít třídy `AreaExtractor`, která je schopná v grafu silnic vyhledat uzavřené cykly. Ty pak lze prohlásit za zóny.

Druhou úrovní územního dělení jsou bloky. Pro hledání bloků v zónách slouží také třída `AreaExtractor`. Bloky jsou tvořeny minimálními cykly v síťi silnic, jenž prochází danou oblastí. Reprezentovány jsou třídou `Block`. Použitím tohoto algoritmu neumožníme automatickou tvorbu budov okolo silnic, jenž jsou pouze větvemi a nejsou v grafu součástí cyklu. Takových to silnic není ve městech mnoho. Problémy mohou nastat na okrajích, kde mohou být situovány silnice, jenž vedou z města pryč. Tento problém by mohl být námětem pro budoucí rozšíření.

Po vyhledání bloků mezi silnicemi musíme ty, které jsou příliš velké rozdělit na menší části. Tyto části nazýváme parcely (*allotments*). V této knihovně jsou parcely reprezentovány třídou `Lot`. Pro dělení je použit algoritmus rekurzivního dělení polygonu na menší části. Tento tzv. *lot subdivision* algoritmus představili Parish a Müeller v roce 2001 [15]. Kelly ho v roce 2007 rozšířil o podporu nekonvexních polygonů [6]. Knihovna používá tuto modifikovanou verzi.

V průběhu hledání a dělení oblastí na bloky a parcely může dojít k tomu, že díky povaze vygenerované silniční sítě nejsou některé oblasti vhodné pro umístění budov či jiných objektů. Jde například o parcely, které nemají přístup k silnici a stavba domů by zde byla zbytečná. Dále jsou to příliš malé a nebo nevhodně tvarované parcely. Tyto je třeba rozpoznat a odstranit ze systému.



Obrázek 5.4: Ukázka rozdělení částí (zón) města na bloky a dále parcely, do kterých jsou umístěny budovy. Některé bloky a parcely jsou nevhodné a tak jsou zahozeny.

5.2.5 Budovy

Generování budov je již velkou měrou závislé na vykreslovací technologii, a tak již nespadá pod pole působnosti této knihovny. Knihovna *libcity* pouze poskytuje bázovou třídu, na základě které jsou pak odvozovány třídy pro konkrétní budovy. K tomuto účelu slouží třída `Building`. Je zde využito vícenásobné dědičnosti, třída `Building` vznikne složením tříd `UrbanEntity` a `GraphicLSystem`. První bázová třída zajišťuje to, aby mohly být výsledné budovy přiřazeny k parcelám a třída `GraphicLSystem` poskytuje budovám implementaci přepisovacího systému.

Sama třída přichází s jedním rozšiřujícím konceptem pro L-systémy. Využívá třidimenzionální obálky (*bounding box*) pro specifikaci prostorového omezení, které může být zabráno budovou. Obálka může být modifikována, stejně jako pozice želvy pomocí symbolů gramatiky.

5.2.6 Generování pseudonáhodných čísel

Generování pseudonáhodných čísel je také důležitou součástí knihovny. Pro dosažení dostatečných variací výsledků specifikujeme většinu parametrů v nějakém intervalu. Generátor pak určuje, které číslo z tohoto intervalu je použito.

Pro implementaci byla zvolena varianta kongruentního generátoru s rovnoměrným rozložením. Pro účely definice rozptylu hodnot v nějakém intervalu by bylo možno uvažovat i o implementaci generátoru s normálním rozložením.

Generátor je implementován ve třídě `Random`. Je zde využit návrhový vzor factory, pro konstrukci objektů, jenž jsou pak použity jako parametry.

Třída `Random` disponuje možností nastavení jednoho globálního seedu. Toto globální nastavení však může být při konstrukci objektu přepsáno lokálním seedem. Tato vlastnost je důležitá, chceme-li několikrát opakovat proces tvorby budovy a dosáhnout stejných výsledků.

5.3 Demonstrační aplikace OgreCity

Aplikace *OgreCity* je na konci sekvence procedurálního generování města a stará se o vykreslování výstupu. Již z názvu vyplývá, že program je napsán s pomocí grafického systému *OGRE 3D*. Účelem této aplikace je demonstrovat možnosti využití knihovny *libcity* a vykreslit pomocí ní město.

5.3.1 Vykreslovací třídy

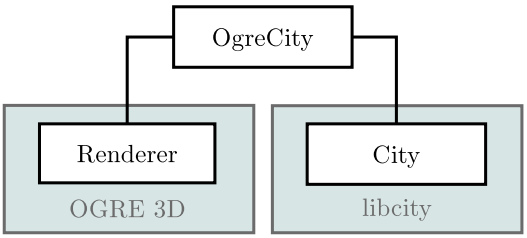
Základním stavebním kamenem je bázová třída `Renderer`. Všechny třídy, které něco vykreslují vychází z této třídy a musí implementovat její virtuální metodu `render()`. Tento návrh přináší do systému konzistenci. Vždy je jasné, v jaké části kódu nalezneme kód, který vytváří část scény.

Třídy typu `Renderer` a třídy z ní odvozené lze uspořádávat do stromové struktury. Vezměme si jako příklad vykreslení auta. Automobil je složen z velkého množství součástí. Jeho vykreslování pomocí třídy `Renderer` by mohlo být rozděleno například na vykreslení karoserie a čtyř instancí kola. Třída pro kreslení karoserie by mohla dále být rozdělena na kreslení předního nárazníku, kapoty, zadního nárazníku, střechy a tak dále.

Pokud bychom implementovali všechen vykreslovací kód pro automobil do jedné třídy, bylo by ho tolik, že by se stal nepřehledným a další úpravy či rozšiřování by bylo složité. Při rozdělení do několika tříd dosáhneme nejen vyšší čitelnosti, ale spolu s tím přijde i možnost znovupoužití vykreslovacího kódu pro určité části i jinde. Například kolo můžeme použít i při vykreslování jiného typu automobilu. Dále máme dynamicky možnost nahrazovat části kódu jiným. Nahrazením třídy pro vykreslení nárazníků jinou třídou můžeme autu jednoduše přidat spoilery.

5.3.2 Propojení s libcity

Tato aplikace tvoří jistý most mezi knihovnou určenou k procedurálnímu generování měst a grafické knihovny. Místem, kde dochází k tomuto propojení je třída `OgreCity`. Tato třída je potomkem rozhraní knihovny *libcity*, třídy `City` (popsána v kapitole 5.2.1) a zároveň také odvozenou třídou od třídy `Renderer`. Sémantika prvního vztahu říká „`OgreCity` je město“, druhý vztah značí, že „`OgreCity` je zobrazovač“. Třída `OgreCity` tedy slouží k vykreslování města a můžeme ji prohlásit za rozhraní mezi těmito dvěma knihovnami.



Obrázek 5.5: Propojení knihoven OGRE 3D a libcity.

5.3.3 Konfigurace parametrů

Možnost konfigurace parametrů pro tvorbu města je klíčovým aspektem celé aplikace. Bez možnosti parametrizace by ztratilo smysl procedurální generování a mohli bychom město vymodelovat ručně v 3D editoru. Parametrizovat lze v procesu procedurálního generování měst různé věci, na různých místech v programu. Takovéto řešení není vhodné, protože pro změnu parametrů by bylo nutné prohledávat zdrojový kód. Další cesta by mohla vést přes definice preprocesoru. Tyto makra jsou globální a tedy dostupná v celém zdrojovém kódu. Při použití maker sice budeme mít všechny hodnoty na jednom místě, ale ve zdrojových kódech zase potom nebude jasné, kde se tyto hodnoty berou. Používání globálních proměnných je nejlépe se vyhnout, je-li je to možné.

Zajímavou možností je implementace virtuální metody `configure()` do třídy `OgreCity`. V této metodě by došlo k nastavení všech potřebných hodnot libovolným způsobem, to již záleží na programátorovi. Mohly by být nastaveny přímo či načteny z nějakého konfiguračního souboru.

Výhod z tohoto řešení plyne hned několik. První je výše zmíněná rozšířitelnost. Odkud je konfigurace získána není důležité, musí však být dostupná. Další výhodou je možnost předefinování metody configure a tím změny vzhledu celého města. Máme možnost vytvořit bázovou třídu, kde implementujeme vykreslovací algoritmy a dále několik odvozených tříd, jenž vždy použijí jinou konfiguraci a budou produkovat různé výsledky.

Kapitola 6

Implementace

Kapitola Implementace rozvádí informace z předcházející kapitoly Návrh aplikace (5) a popisuje některé důležité implementační detaily celého systému. Na začátku kapitoly jsou popsány technologie, jenž byly použity k implementaci knihovny *libcity* a demonstrační aplikace OgreCity. Dále jsou rozebrány použité algoritmy a postupy ve zdrojovém kódu obou částí projektu.

Všechny zdrojové kódy, jak knihovny libcity tak demonstrační aplikace, byly zveřejněny pod licencí GNU GPLv3[1] a jsou hostovány na serveru http://www.github.com/. Každá část má samostatný repositář (libcity a OgreCity).

6.1 Použité technologie

Mezi technologickými požadavky na aplikaci byla mimo jiné zadána i multiplatformnost, tedy možnost použití napříč platformami a operačními systémy bez nutnosti podstatných změn v aplikaci. Tuto skutečnost bylo nutné brát v úvahu při výběru programovacího jazyka a s tím spojených dalších nástrojů a knihoven.

Jako programovací jazyk pro knihovnu i demonstrační aplikaci byl zvolen C++. Jazyk C++ splňuje vznesené požadavky (překladače existují pro mnoho různých architektur a platforem). Dále je objektově orientovaný, což umožňuje tvorbu výstižnějšího návrhu oproti jazykům strukturovaným/modulárním. Dalším silným argumentem byla dostupnost implementace velkého množství knihoven a grafických toolkitů pro C++, ze kterých je možno vybrat vhodný nástroj pro implementaci vykreslovací části. Jde například o GLUT, OpenSceneGraph, Panda3D, Ogre3D, Qt a další.

Při výběru použitých nástrojů byl také brán ohled na jejich otevřenost, pokud to bylo možné. I právě proto byla zvolena knihovna Ogre 3D.

6.1.1 OGRE 3D

Ogre 3D (neboli Object-Oriented Graphics Rendering Engine, volně přeloženo jako Objektově-orientovaný grafický vykreslovací systém) je zralá, stabilní, spolehlivá, flexibilní, *multiplatformní* knihovna s velkým množstvím funkcí určena pro použití při vývoji 3D grafických běžících v reálném čase. [10]

Ogre požívá scene graph, podporuje velké množství správců scén (*scene managers*), například octree, BSP a Paging Landscape scene manager. Celká knihovna je plně multiplat-

[1] http://www.gnu.org/licenses/gpl.html

```
1  TEST(VectorLength)
2  {
3    Vector v(0, 1);
4
5    CHECK(1 == v.length());
6    CHECK_CLOSE(1, v.length(), libcity::EPSILON);
7  }
```

Zdrojový kód 6.1: Příklad jednoduchého testu v UnitTest++.

formní, s podporou OpenGL a Direct3D. Programy napsané za použití knihovny Ogre by měly beze změn být spustitelné i po přenosu na jinou platformu. Ogre bývá často používán k jako herní engine k tvorbě jak nezávislých tak komerčních her (například Ankh, Torchlight nebo Garshasp). Také bývá často využíván pro vizualizace výsledků simulací.

6.1.2 UnitTest++

Pro implementaci jednotkových testů testování projektu byl použit framework pro C++ *UnitTest++* [12]. Jde o jednoduchou knihovnu určenou k testování jednotek pro jazyk C++. Franework byl navrhnut pro použití na široké paletě platforem. Tento framework klade důraz na jednoduchost a rychlost přidávání nových testů nových testů. Tyto vlastnosti jsou velmi důležité, protože tvorba jednotkových testů je při vývoji řízeném testy častou náplní práce a proto volba nevhodného frameworku může zbytečně zdržovat vývoj projektu. Použití frameworku k vytvoření jednoduchého testu je ukázáno pomocí zdrojového kódu 6.1.

6.2 libcity

Tato kapitola vysvětluje některé použité algoritmy a vybrané implementační detaily knihovny *libcity*. Cílem není podat úplné informace o všech implementačních detailech, ale popsat vybrané a zajímavé postupy, jenž byly implementovány v průběhu tohoto projektu. Úplné informace o implementaci naleznete v dokumentaci zdrojového textu buď v kódu samotném a nebo vygenerováním dokumentace pomocí nástroje *doxygen*.

6.2.1 Balíčky

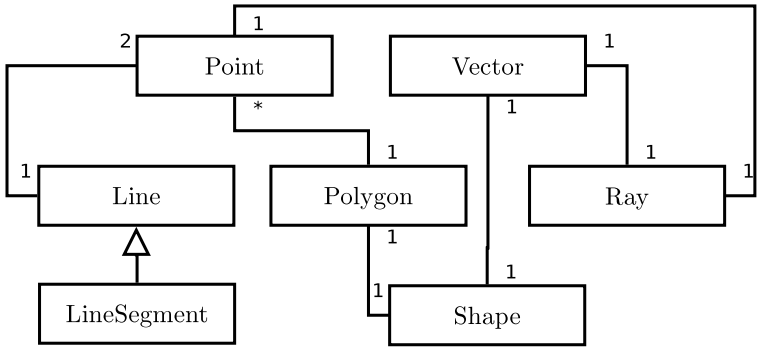
Zdrojové kódy knihovny čítají přes 30 modulů. Každý modul je tvořen jedním hlavičkovým a jedním zdrojovým souborem. Tím se dostáváme na zhruba 60 zdrojových souborů. Z tohoto důvodu byl, pro větší přehlednost, kód strukturován do několika balíčků:

- `geometry` – Obsahuje třídy, jenž reprezentují geometrické útvary jako bod, vektor, plocha, polygon a podobně.
- `lsystem` – Implementace tříd odvozených od `LSystem`.
- `streetgraph` – Třídy, které pracují na tvorbě sítě silnic, silnice, křižovatka atp.
- `area` – Územní celky na které je město děleno, bloky, parcely.
- `entities` – Objekty, jenž mají být umístěny do městské zástavby.

Balíčky, kromě fyzického rozdělení souborů se zdrojovým textem do více adresářů, neplní žádnou jinou funkci.

6.2.2 Geometrie

Balíček `geometry` obsahuje 7 tříd, každá z nich reprezentuje nějaký geometrický útvar. Jejich přehled společně se vzájemnými vztahy je na obrázku 6.1.



Obrázek 6.1: Přehled tříd v geometrickém balíčku.

K implementaci vlastní knihovny s geometrií jsme se uchýlili kvůli zachování co nejmenších externích závislostí knihovny na okolí. Nynější implementace *libcity* je tak závislá pouze na standardní knihovně jazyka C++.

V tomto balíku byly implementovány algoritmy s ohledem na jejich funkčnost. Optimalizace byla v drtivé většině případů zanedbána. Proto je v tomto místě velký prostor pro zrychlení a zvýšení výkonu celé knihovny.

6.2.3 StreetGraph

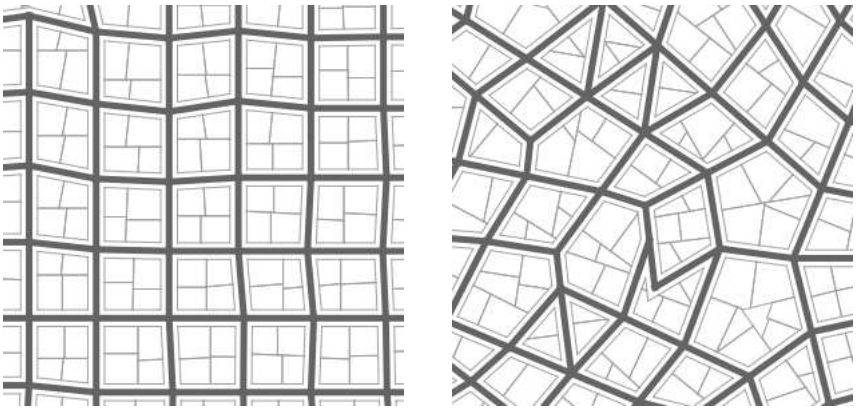
Street graph je datová struktura, jenž uchovává informace o tom, odkud kam vede silnice, s jakými ostatními silnicemi se kříží a jakou vede cestou. Současná implementace grafu je pomocí dvou tříd. První je `Road`, ta představuje segment silnice a také tvoří hranu v grafu. Druhou třídou je `Intersection`, reprezentace křižovatky a také uzlu v grafu.

Přidávání nových částí do grafu probíhá pomocí metody `addRoad()`. Metoda je také zodpovědná za zachování integrity grafu. Síť silnic musí vždy tvořit planární graf. Pokud se právě přidávaná silnice kříží s jinou, již existující, je přidána do grafu křižovatka a silnice jsou rozděleny na více segmentů. Pokud je graf správně vytvořen, má každý silniční segment (každá jedna instance třídy `Road`) *právě dvě* křižovatky. Za křižovatky jsou považovány i slepé konce ulic. Každá křižovatka má vždy jednu a více silnic. Pokud ke křižovatce již žádná silnice nevede, měla by být odstraněna.

Nad třídou `StreetGraph` také pracuje třída `AreaExtractor`. Jejím úkolem je vyhledávání území v okolí silnic, přesněji minimální cykly v grafu silnic, do kterých následně budou umístěny budovy. `AreaExtractor` využívá implementaci algoritmu hledání minimálních cyklů, jenž popisuje David Eberly v článku *The Minimal Cycle Basis for a Planar Graph* [3].

6.2.4 Generátory silniční sítě

Tvorba sítě silnic je řízena pomocí generátorů silniční sítě. Každý generátor po vytvoření dostane přiřazenu instanci třídy `StreetGenerator`, kam přidává silnice a také `Polygon` jenž definuje hranice, jenž definují, kde může generátor pracovat.



(a) Rastrový vzor (b) Organický vzor

Obrázek 6.2: Síť pozemních komunikací vytvořena pomocí knihovny libcity.

Při přidávání komunikací do mapy je na generátoru, aby zajistil konzistenci výsledného grafu. K tomu to účelu je zde implementována funkce `localConstraints()`, jenž je zavolána pro každý nový segment. Funkce zkontroluje relativní polohu nového segmentu od již existujících (jde o implementaci kontextové podmínky L-systému). Pokud není splněna, celá nová větev je zahozena (k tomu slouží funkce `cancelBranch()`).

Všechny generátory silnic jsou odvozené od třídy `RoadLSystem`. Chceme-li přidat další typ generátoru, stačí vytvořit nového potomka této třídy. V současnosti jsou dostupné dva typy generátorů. Pro rastrový vzor silnic `RasterRoadPattern` a pro organický vzor `OrganicRoadPattern`. Ukázka silniční sítě, jenž byla vygenerována pomocí těchto dvou generátorů je na obrázku 6.2.

6.2.5 Rozhraní

Rozhraní knihovny, třída `City` definuje následující virtuální metody, jenž musí odvozené třídy implementovat. Jde o tyto metody:

- `void createPrimaryRoadNetwork()` – Definice okrajů města a vygenerování hlavní silniční sítě. Příklad implementace této metody naleznete na obrázku 6.2.
- `void createZones()` – Definice městských zón. Nejčastěji jde o vyhledání minimálních cyklů v grafu silnic, ale programátor má možnost i definice vlastních oblastí, podle toho jak uzná za vhodné.
- `void createSecondaryRoadNetwork()` – Tvorba vedlejších silnic pro každou definovanou oblast.
- `void createBlocks()` – Rozdělení oblastí na bloky a parcely.
- `void createBuildings()` – Výběr oblastí, do kterých budou umístěny budovy a následné vygenerování budov.

Metody přibližně odpovídají etapám generování města popsaným v kapitole 3. Tato sekvence je následně volána po sobě v metodě `generate()`.

```
1  StreetGraph* map = new StreetGraph;
2  OrganicRoadPattern* generator = new OrganicRoadPattern();
3
4  Polygon* area = new Polygon;
5  area->addVertex(Point(1000,1000));
6  area->addVertex(Point(1000,-1000));
7  area->addVertex(Point(-1000,-1000));
8  area->addVertex(Point(-1000,1000));
9
10 generator->setTarget(map);
11 generator->setAreaConstraints(area);
12 generator->setRoadType(Road::PRIMARY_ROAD);
13 generator->setRoadLength(600, 800);
14 generator->setSnapDistance(200);
15
16 generator->setInitialPosition(area->centroid());
17 generator->setInitialDirection(Vector(0,1));
18
19 generator->generate();
```

Zdrojový kód 6.2: Příklad použití generátoru hlavních silnic.

6.3 OgreCity

Tato podkapitola je věnována především použitým vykreslovacím postupům a procedurálnímu generování objektů v OGRE 3D.

6.3.1 Vykreslení prostředí

O vykreslení prostředí se stará třída `EnvironmentRenderer`. Vykreslen je terén, obloha a světelné podmínky. Vykreslení terénu probíhá z výškové mapy a je relativně náročné (využívá `Ogre::TerrainManager`). Z toho důvodu bylo vykreslení terénu přesunuto do zvláštní třídy `TerrainRenderer`. Obloha je vykreslena pomocí sky boxu, který je už součástí knihovny Ogre.

6.3.2 Procedurální geometrie v OGRE

Ještě něž přejdeme k principům vykreslování objektů, je nutné se seznámit s tím, jakým způsobem je v knihovně Ogre 3D možné vytvářet modely přímo v programu. K tomu je určena třída `Ogre::ManualObject` [19], jenž umožňuje vytváření objektů pomocí postupného přidávání vrcholů (metodou `position()`). Na každý vrchol lze voláním metody `textureCoord()` namapovat texturu.

Nakonec je nutné definovat povrch objektu pomocí indexování vrcholů. K tomu slouží metody `index()` či zkratky, kterými rovnou definujeme trojúhelníky (`triangle()`) nebo čtverce (`quad()`). Problém nastává v okamžiku, kdy chceme vykreslit povrch tvořený obecným (konvexním i nekonvexním) polygonem o předem neznámém počtu vrcholů. K tomuto účelu byl použit algoritmus pro rozklad polygonu na trojúhelníky (*polygon triangulation algorithm*) [9].

Vytvořený objekt je pak nutné umístit do scény. To lze v OGRE několika způsoby, podle toho, k čemu je objekt určen. Pokud se jedná o objekt, jímž chceme dále manipulovat (přesouvat, transformovat apod.), vložíme ho do scény jako *movableObject*. Druhou možností

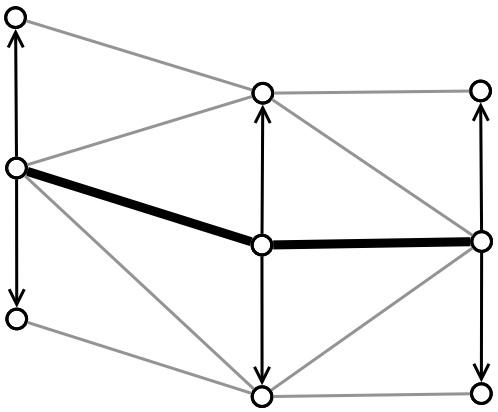
je tvorba tzv. statické geometrie. Takto vytvořené objekty již nelze přesouvat, zabírají více místa v paměti, ale jejich vykreslování je daleko efektivnější a rychlejší.

Pro vykreslování všech částí města (silnic i budov) byla zvolena druhá metoda, protože není nutné (ani žádoucí) je v budoucnu přesouvat.

6.3.3 Vykreslování silniční sítě

Při vykreslování pozemních komunikací máme k dispozici jako vstup instanci objektu `StreetGraph`, tedy mapu podle které bude vykreslování probíhat a dále terén, kam budou silnice umístěny. Pokud je terén rovný je situace jednoduchá, problémy nastávají až, v případě jeho zakřivení.

Informace, jenž zjistíme z grafu silnic jsou odkud vede která komunikace a kde jsou umístěny křižovatky. Úkolem tedy je vytvořit silnici mezi dvěma body tak, aby kopírovala terén. Využijeme zde výše zmíněných manuálních objektů, jenž poskytuje knihovna Ogre 3D pro procedurální tvorbu objektů. `StreetGraph` poskytne tzv. páteř silnice. Při vykreslování musíme vytvořit vrcholy v určité vzdálenosti (dané šířkou silnice) od této páteře tak, aby vznikla plocha, jenž bude tvořit silnice. Tvorba silnice je znázorněna na obrázku 6.3. Kruhy představují řídící body objektu, černé šipky jsou normálové vektory k páteři cesty. Na konci těchto normál jsou přidány další řídící body (je vytvořena plocha silnice). Šedé hrany pak znázorňují, jak je konstruhován povrch objektu pomocí indexované geometrie.



Obrázek 6.3: Vykreslení dvou vzorků silnice.

Dalším problémem je vykreslení křižovatek. Pokud se někde silnice scházejí, nelze je vykreslit jen tak, přes sebe. Při řešení křižovatek je nutné spočítat průsečíky a nastavit společné řídící body dané křižovatky. Dále je nutné přestat vzorkovat silnici ve vhodném okamžiku tak, aby nezasahovala do křižovatky.

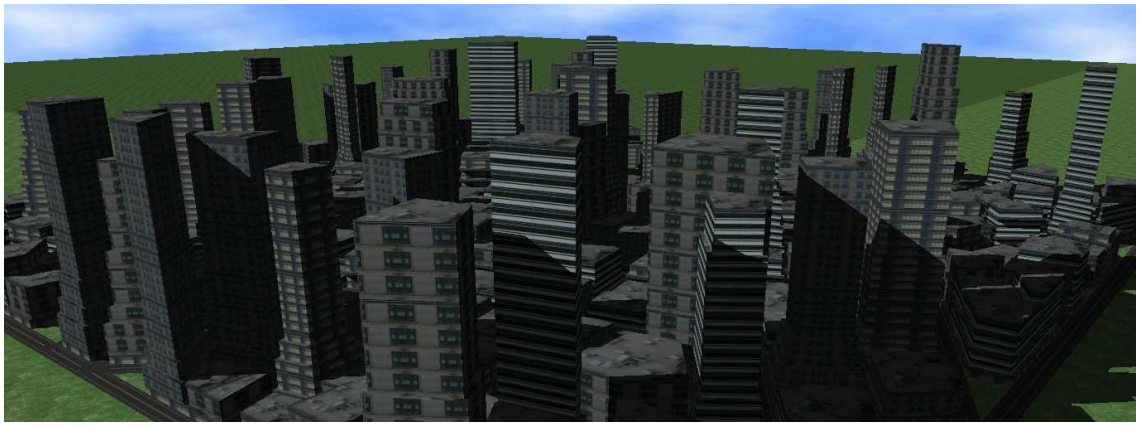
Pro vykreslování křižovatek bylo implementováno několik algoritmů. Nejjednodušší z nich vykresluje pouze dvoucestné křižovatky (tedy spojení silnic). Tento algoritmus byl následně rozšířen pro tři a čtyř-cestné křižovatky. Dále je možné jej zobecnit pro křižovatky s n cestami, ale tento typ křížení pozemních komunikací není příliš běžný, proto byly křižovatky s více než čtyřmi cestami zanedbány.

6.3.4 Budovy

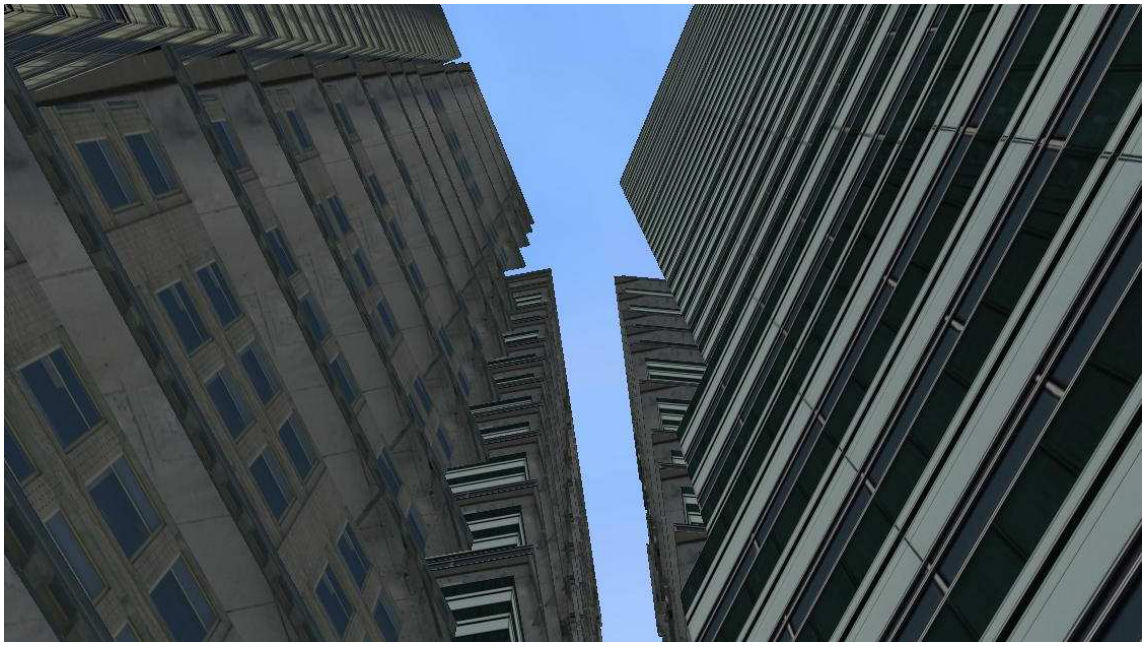
Vykreslování budov je silně individuální a záleží vždy na konkrétním typu budovy, jaké je použito nastavení. Pro budovy je v knihovně libcity dostupná třída `Building` (vychází

z L-systémů). OgreCity definuje odvozenou třídou `OgreBuilding`, která definuje základní symboly do abecedy L-systému jako je vykreslení plochy, patra a podobně. Další aspekty tvorby budov záleží čistě na tvůrci vykreslovací aplikace.

Pro OgreCity bylo vytvořeno několik tříd, jenž vykreslují různé druhy budov. Například třída `SkyScraper` vykresluje věžové budovy a mrakodrapy apod.



Obrázek 6.4: Město vygenerované pomocí OgreCity.



Obrázek 6.5: Pohled na mrakodrapy v OgreCity.

Kapitola 7

Testování

Testování software je důležitou součástí jeho vývoje. Při vytváření funkcionality programu je nutné se ujišťovat, že kód funguje tak, jak bylo zamýšleno a provádí právě to, co se od něj očekává (tedy o nic méně a také nic neočekávaného navíc). Někteří programátoři dokonce považují funkce programu, pro které nejsou testy za neexistující [1].

Testování nabírá na důležitosti s rozsahem projektu. Čím je systém rozsáhlejší, tím více škody může způsobit přidání nesprávně fungujícího kódu. Netestujeme-li vůbec, tak se při každém rozšíření zdrojového kódu o novou funkcionalitu pouštíme na „velmi tenký led“, protože si nikdy nemůžeme být jisti, jestli provedená změna nějakým způsobem nenarušila existující systém.

Tato kapitola popisuje metody testování, jenž byly použity v průběhu tohoto projektu.

7.1 Testování funkcionality

K testování knihovny libcity byla vytvořena sada jednotkových testů. Testy jsou strukturovány podle tříd, jejichž funkcionalitu testují. Testy pro jednu třídu jsou vždy v jednom souboru. K tvorbě testů byl použit framework pro jazyk C++, UnitTest++ (viz 6.1.2).

Celkem bylo implementováno 84 testů v 18 testových modulech. Testováno je jak rozhraní mezi objekty, tak správnost implementace různých algoritmů za různých vstupních podmínek. Aktuální revize zdrojového textu splňuje všechny testy na 100%.

Díky sadě testů si můžeme dovolit i výraznější zásahy do systému a mít jistotu, že nezpůsobíme problémy v ostatních částech, proběhnou-li všechny testy jednotek bez chyby.

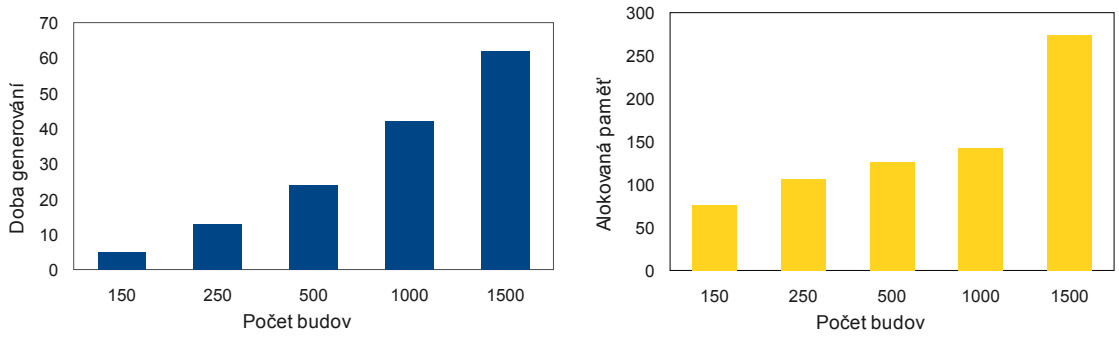
Dalším nástrojem, jenž byl specificky vytvořen a použit pro testování knihovny libcity je aplikace *libcity map*. Jedná se o vykreslovací část (tedy ekvivalent aplikace OgreCity). Knihovna libcity sama o sobě neumožňuje žádné zobrazení výsledků, proto by bylo ladění bez použití nějakého dalšího nástroje velmi zdlouhavé a neefektivní.

Aplikace mapa je napsána s využitím UI frameworku Qt. Po spuštění aplikace vygeneruje podle nastavení síť pozemních komunikací a vykreslí ji jako mapu. Dále je aplikace schopná do mapy vykreslit vytvořené bloky a parcely. Příklad výstupu z této aplikace je na obrázku 6.2.

7.2 Testy výkonnosti

V rámci testování systému bylo prováděno vyhodnocení rychlosti aplikace. Bylo prováděno měření toho, jakým způsobem ovlivní různé konfigurace dobu generování výsledku. Všechna měření byla provedena na počítači s dvoujádrovým procesorem pracujícím na frekvenci 1,80 GHz, 2GB operační paměti a grafickým adaptérem ATI Radeon HD 2600XT. Na počítači byl nainstalován operační systém Ubuntu 9.10 32bit s verzí jádra 2.6.31-22-generic. Byla použita knihovna OGRE 3D ve verzi 1.7.3 (Cthugha).

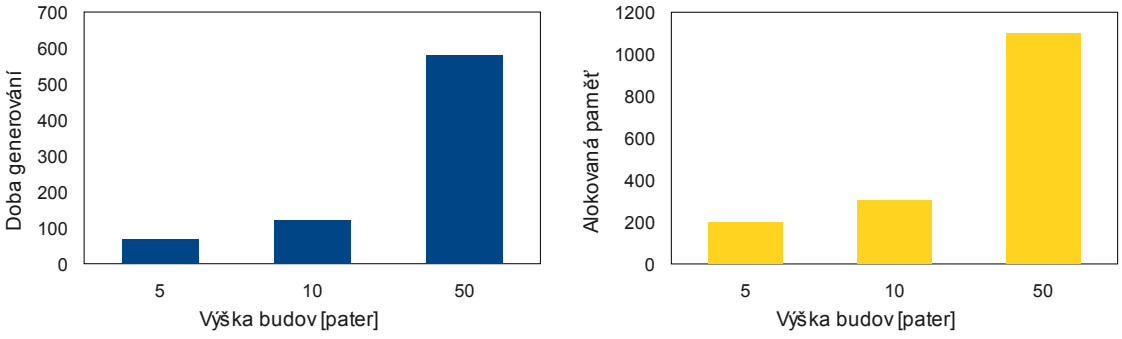
U prvního testu jsme měřili dobu trvání generovacího procesu v závislosti na počtu budov. Na obrázku 7.1 je průběh graficky znázorněn. Pro vygenerování města se 150 budovami stačilo 5 vteřin. Při desetinásobném zvýšení počtu budov generování trvalo 62 vteřin. Závislost vychází téměř lineární.



(a) Doba generování (b) Alokované zdroje

Obrázek 7.1: Znázornění doby generování a alokované paměti programem v závislosti na počtu budov.

Dále jsme měřili jak ovlivňuje výsledek výška budov ve městě. Čím vyšší jsou budovy, tím více generací musí L-systémy provádět. Toto měření ukazuje, jakým způsobem ovlivňuje míra detailu dobu generování města (vyšší detail budovy vyžaduje více generací). Výsledek je znázorněn na obrázku 7.2. Měření bylo prováděno ve městě s 1500 budovami.



(a) Doba generování (b) Alokované zdroje

Obrázek 7.2: Znázornění doby generování a alokované paměti programem v závislosti na výšce budov.

Generování pětipatrových budov trvalo přibližně 70 vteřin, při desetinásobném nárůstu

výšky budov (na 50 pater) se zvedla doba generování přibližně také na desetinásobek. Generovací proces pak trval 10 minut. Závislost je i v tomto případě téměř lineární. Ukazuje se, že úroveň detailu budov má podstatný vliv na dobu generování města. Při generování přílišných geometrických detailů by mohlo dojít k neúnosnému nárůstu doby generování.

Jako poslední jsme měřili dobu generování pouze silniční sítě samotné. Zjištěné hodnoty jsou uvedeny v tabulce 7.1.

Počet segmentů	250	500	2000	20000
Doba generování	2,3s	4s	17s	27min
Alokovaná paměť [MB]	15	15	16	19

Tabulka 7.1: Závislost doby generování sítě pozemních komunikací na počtu silničních segmentů.

Při desetinásobném nárůstu počtu silničních segmentů vzrostla doba generování přibližně stokrát. Takovýto vysoký nárůst je způsoben použitým *snap algoritmem*. Každá nová silnice je porovnávána se všemi již existujícími a jsou hledány průsečíky. K zlepšení tohoto problému by mohla vést optimalizace tohoto algoritmu tak aby nebylo nutné procházet vždy všechny existující segmenty, ale jen určitou podmnožinu segmentů, které jsou novému segmentu nejblíže.

Kapitola 8

Závěr

Cílem bakalářské práce bylo prostudovat techniky a postupy používané v oblasti procedurálního generování měst a na základě těchto poznatků implementovat multiplatformní aplikaci, jenž bude demonstrovat použití těchto algoritmů.

Na začátku jsme studovali existující systémy a to jak pracují. Dále se hlouběji zabývali principy vybraných technik procedurálního generování měst a tvorbou realistického modelu města. Zejména postupů vycházejících z formálních gramatik. Na základě nastudovaných poznatků byl proveden návrh aplikace. V této fázi byl projekt rozdělen do dvou částí. První část tvoří knihovna libcity. Nad ní je vystavěna demonstrační aplikace OgreCity.

Objektově orientovaná knihovna implementuje používané algoritmy pro generování topologie města. Součástí knihovny jsou nástroje pro generování sítě pozemních komunikací v několika vzorech, dále algoritmus dělení území města na bloky a parcely. Je také položen formální základ pro generování budov (L-systémy).

Při navrhování knihovny byl kladen důraz na snadnou rozšiřitelnost a rozsáhlé možnosti konfigurace. Programátor dostává k dispozici určité nástroje a může si vybrat jakým způsobem je použije. Není nucen je využívat přesně tak, jak to zamýšlí autor. S knihovnou nejsou svázané žádné vykreslovací algoritmy, což umožnilo implementaci bez jakýchkoliv závislostí na dalších knihovnách či platformě. Zdrojový text je možno přeložit v jakémkoliv C++ kompilátoru s podporou standardní knihovny.

Nad knihovnou byla dále vystavěna demonstrační aplikace *OgreCity*. Jejím hlavním úkolem je předvedení možností implementovaných algoritmů.

Rozdíl od ostatních, již existujících systémů spočívá v zaměření projektu. Cílovou skupinu tohoto projektu netvoří běžní uživatelé či designeři, ale programátoři. Knihovna je určená k použití v hrách, simulacích či jiných aplikacích, kde vzniká potřeba procedurálního generování města. Zdrojové texty byly zveřejněny pod open-source licencemi (viz kapitola 6).

Možná rozšíření

Procedurální generování měst je poměrně rozsáhlá oblast, a proto existuje spousta možností dalšího rozšíření tohoto projektu a pokračování práce.

V současné době jsou jedinými objekty ve virtuálním OgreCity budovy. Jedním z možných rozšíření je implementace generování i jiných objektů, například náměstí, parků, stromů a podobně. K tomuto rozšíření je již knihovna libcity připravena.

Prostor pro zlepšení poskytuje i algoritmus pro generování budov. Nynější implementace

pomocí L-systémů poskytuje dobré výsledky, avšak vhodnějším nástrojem jsou Chomského sekvenční gramatiky. Mezi ně patří například gramatika CGA shape, která je přímo určená pro generování architektury.

Cílem budoucí práce by také mohla být optimalizace rychlosti a paměťové náročnosti celé aplikace.

Literatura

[1] Beck, K.: *Extrémní programování*. U průhonu 22, 170 00 Praha 7: GRADA Publishing, spol. s r. o., 2002, ISBN 80-247-0300-9.

[2] Décoret, X.; Sillion, F.: Street Generation for City Modelling. In *Architectural and Urban Ambient Environment*, 2002.
URL http://artis.imag.fr/Publications/2002/DS02

[3] Eberly, D.: The Minimal Cycle Basis for a Planar Graph. 2008.
URL http://www.geometrictools.com/Documentation/MinimalCycleBasis.pdf

[4] George Kelly: Interactive Procedural City Generation. [online], Sat, 2008-03-08 20:40.
URL http://citygen.net/

[5] George Kelly, G. M.: A Survey of Procedural Techniques for City Generation. 2006: s. 87–131.
URL http://www.itb.ie/site/researchinnovation/itbjournal/ITB-Journal-December-2006.pdf

[6] George Kelly, G. M.: Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, 2007, s. 8–16.
URL http://citygen.net/files/citygen_gdtw07.pdf

[7] Greuter, S.; Parker, J.; Stewart, N.; aj.: Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-578-5, s. 87–ff.
URL http://doi.acm.org/10.1145/604471.604490

[8] Haegler, S.; Wonka, P.; Arisona, S. M.; aj.: Grammar-Based Encoding of Facades. In *EGSR 2010 (Eurographics Symposium on Rendering, Computer Graphics Forum)*, ročník 29, editace J. Lawrence; M. Stamminger, Wiley, June 2010, s. 1479–1487.

[9] J., R.: Efficient Polygon Triangulation. [online], 2008.
URL
http://www.flipcode.com/archives/Efficient_Polygon_Triangulation.shtml

[10] Junker, G.: *Pro OGRE 3D Programming (Pro)*. Berkely, CA, USA: Apress, 2006, ISBN 1590597109.

[11] Koutný, J.: L-systémy a jejich aplikace. FIT VUT v Brně, 2008.

[12] Llopis, N. and Nicholson, C.: UnitTest++. [online], 2010.
URL http://unittest-cpp.sourceforge.net/

[13] Müeller, P.: Applied procedural modeling. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA: ACM, 2007, s. 112–147.

[14] Müller, P.; Wonka, P.; Haegler, S.; aj.: Procedural Modeling of Buildings. ročník 25, č. 3, 2006: s. 614–623, ISSN 0730-0301.

[15] Parish, Y. I. H.; Müller, P.: Procedural Modeling of Cities. In *Proceedings of ACM SIGGRAPH 2001*, editace E. Fiume, New York, NY, USA: ACM Press, 2001, ISBN 1-58113-374-X, s. 301–308.

[16] Procedural Inc.: 3D Modeling Software for Urban Environments. [online], 2010.
URL http://www.procedural.com/

[17] Stiny, G.: Introduction to shape and shape grammars. *Environment and Planning B*, ročník 7, č. 3, 1980: s. 343–351, doi:10.1068/b070343.
URL http://dx.doi.org/10.1068/b070343

[18] Sun, J.; Yu, X.; Baciu, G.; aj.: Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-530-0, s. 33–40.

[19] Torus Knot Software: OGRE: API Reference Start Page. [online], Fri, Oct 15 2010 11:10:03.
URL http://www.ogre3d.org/docs/api/html/

[20] Whelan Hugh, G. M., George Kelly: Roll your own city. In *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*, DIMEA '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-248-1, s. 534–535.
URL http://doi.acm.org/10.1145/1413634.1413742

[21] Wonka, P.; Wimmer, M.; Sillion, F.; aj.: Instant architecture. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-709-5, s. 669–677.
URL http://doi.acm.org/10.1145/1201775.882324

Obsah CD

- `/libcity/` – zdrojové kódy knihovny libcity
- `/libcity/doc/` – dokumentace zdrojových kódů knihovny libcity
- `/libcity_map/` – ladící nástroj pro knihovnu libcity
- `/ogrecity/` – zdrojové kódy aplikace OgreCity
- `/ogrecity/doc/` – dokumentace zdrojových kódů aplikace OgreCity
- `/preview/` – ukázky vygenerovaných měst
- `/procedurally_generated_city.pdf` – text práce ve formátu pdf
- `/procedurally_generated_city_poster.png` – plakát prezentující práci