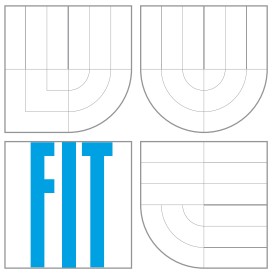
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PROCEDURÁLNÍ ARCHITEKTURA
PROCEDURAL ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
PAVEL ROREČEK

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. TOMÁŠ MILET

BRNO 2016

Zadání bakalářské práce/18018/2015/xrorec00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Roreček Pavel**
Obor: Informační technologie
Téma: **Procedurální architektura**
Procedural Architecture
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte L-systémy a šumy pro procedurální generování grafiky a knihovnu OpenGL.
2. Navrhněte systém pro procedurální generování architektury.
3. Implementujte multiplatformní knihovnu pro procedurální generování architektury. Knihovna umožní export vygenerovaného modelu a bude pracovat s textovým konfiguračním souborem pro popis modelu.
4. Vytvořte jednoduchou vizualizační aplikaci, která využije OpenGL a implementovanou knihovnu. Vytvořte sadu příkladů.
5. Vytvořte video s demonstrací implementované knihovny a práci zveřejněte na internetu pod některou z open-source licencí.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3 a kostra knihovny.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato bakalářská práce popisuje implementaci knihovny pro procedurální generování architektury. V práci je popsán způsob zpracování vstupních souborů obsahujících popis pravidel a symbolů. Vygenerovaný model je zobrazen pomocí vizualizační aplikace využívající knihovnu OpenGL a může být vyexportován do souboru.

Abstract

This Bachelor's thesis describes implementation of library used for procedural generating of architecture. The thesis describes way of handling input files containing rules and symbols. Generated model is shown using visual application that uses OpenGL library and the model can be exported.

Klíčová slova

Procedurální generování, L-systém, architektura

Keywords

Procedural generating, L-system, architecture

Citace

ROREČEK, Pavel. *Procedurální architektura.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Milet Tomáš.

Procedurální architektura

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Roreček
17. května 2016

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Tomáši Miletovi za poskytnutí odborné pomoci, konzultací a cenných rad.

© Pavel Roreček, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce bylo vytvoření multiplatformní knihovny, která načte vstupní soubor obsahující cesty k souborům, ve kterých jsou popsána pravidla popisující způsob, jakým se vygeneruje výstupní model. Syntaxe vstupního souboru je založena na syntaxi jazyka C.

Sestavení pravidel pro generování je sice podstatně složitější než přímé vytvoření modelu v libovolném 3D editoru, ale zato nabízí uživatelům možnost využití náhodnosti při generování výsledných modelů. Náhodnost umožňuje vytvoření šablony, ze které se pokaždé vygeneruje odlišný model.

Knihovna lze uplatnit v mnoha odvětvích, ve kterých je zapotřebí vygenerování většího počtu odlišných modelů stejného typu.

Kapitola 2

Teorie

2.1 Procedurální generování

Procedurální generování je proces vytváření obsahu automaticky za použití algoritmů. Generováním obsahu za běhu aplikace se dramaticky sníží prostorová náročnost aplikace, protože není potřeba uchovávat již vygenerované části v souborech, avšak se zvýší složitost časová.

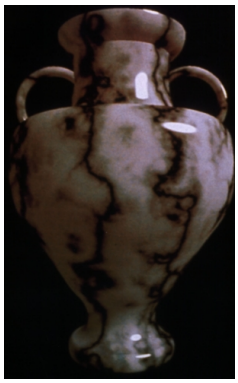
Pro popis algoritmů existuje mnoho způsobů. V této práci bude zmíněno generování pomocí šumů a hlavní důraz bude kladen na L-systémy.

2.1.1 Šum

Šum je nechtěná elektrická nebo elektromagnetická energie, která snižuje kvalitu signálů a dat. Šum se vyskytuje v digitálních a analogových systémech a může ovlivňovat soubory a komunikace všech typů, včetně textu, programů, obrázků, audia a telemetrie [10].

2.1.2 Procedurální šum

Přídavné jméno *procedurální* se používá pro odlišení entit, které nejsou popsány datovými strukturami, ale jsou popsány programovým kódem. Procedurální techniky jsou části kódu nebo algoritmy, které specifikují charakteristiky počítačem generovaných modelů nebo efektů.



Obrázek 2.1: Mramorová váza. (převzato z [4])

Například mramorová textura na obrázku 2.1 používá pro definování barev algoritmy a matematické funkce, nikoliv digitální fotografii. *Procedurální šumovou funkci* definujeme jako procedurální techniku pro simulování a vyhodnocování šumu.

Výhody *procedurálního šumu*:

- Procedurální šumová funkce je extrémně *kompaktní*, většinou vyžaduje poměrně málo kilobytů místa, narozdíl od šumových obrázků a volumetrických dat, které vyžadují několik megabytů.
- Procedurální šumová funkce je *souvislá* a *není založena na diskétních navzorkovaných datech*. Procedurální šumová funkce může vyprodukovat šum v jakémkoli rozlišení.
- Procedurální šumová funkce je *neperiodická*, zcela vyplňující dvou, tří až n-rozměrný prostor. Umožňuje pokrývat velké plochy bez viditelných krajů a opakování.
- Procedurální šumová funkce je parametrizovaná, takže může vytvářet široké spektrum vzorů, není limitována pouze na jeden fixní šumový vzor. Parametry umožňují kontrolovat sílu spektra šumu, která charakterizuje šumový vzor.
- Procedurální šumová funkce poskytuje náhodný přístup. Může být vyhodnocena v konstantním čase, bez ohledu na pozici bodu, pro který se vyhodnocuje. Tato možnost náhodného přístupu činí šumové funkce vhodné pro využití výkonu vícejádrových procesorů.

Tyto výhody jsou pouze potenciální. Nejsou nutně garantovány, avšak měly by být brány jako cíle, kterých by šumové funkce měly dosáhnout [4].

2.2 Formální jazyky

V následující podkapitole budou popsány základy formálních jazyků, systémů a různých druhů L-systémů. Pro popis podmínek použití pravidel je použit deterministický/stochastický parametrický 1L-systém (viz. 2.3.3), který pracuje s pravým kontextem.

2.2.1 Abeceda a jazyk

Abeceda Σ je konečná neprázdná množina, jejíž prvky se nazývají *symboly*. Každá neprázdná podmnožina abecedy Σ je *podabeceda* abecedy Σ. Konečná sekvence symbolů z Σ je řetězec nad Σ; jako ϵ je označován prázdný řetězec - tzn. řetězec skládající se z nula symbolů. Jako Σ^* je označována množina všech řetězců nad Σ; $\Sigma^+ = \Sigma^* - \{\epsilon\}$.

Každá podmnožina $L \subseteq \Sigma^*$ je *formální jazyk* nebo také *jazyk* nad Σ. Pokud L reprezentuje konečnou množinu řetězců označuje se L jako *konečný jazyk* jinak je označován jako *nekonečný jazyk*. Množiny složené z jazyků se nazývají *rodiny jazyků* [5].

2.2.2 Přepisující systém

Přepisující systém je dvojice $M = (\Sigma, R)$, kde

- Σ je abeceda symbolů,
- R je konečná relace nad Σ^*.

Σ se nazývá *úplná abeceda* systému M, nebo jednoduše *abeceda* systému M. Prvek relace R se nazývá pravidlo systému M. R se nazývá *množina pravidel* systému M.

Přepisující relace nad Σ se označuje symbolem $\Rightarrow$ a je definována takto : $\forall u, v \in \Sigma : u \Rightarrow v \in M$, pokud existuje $(x, y) \in R$ a $w, z \in \Sigma^*$ takové, že $u = wxz$ a $v = wyz$. Symbol $\Rightarrow^*$ značí reflexivní a transitivní uzávěr nad $\Rightarrow$[5].

2.2.3 L-systémy

Lindenmayerův systém nebo L-systém je druh přepisovacího systému, ve kterém jsou přepisovací pravidla aplikována tak, aby v každém kroku přepsala co nejvíce symbolů (většinou je přepisován pouze jeden). Tyto systémy byly využívány především pro modelování rostlin, organismů, věcí jako jsou například městské stavby nebo se využívaly při generování rozsáhlé škály fraktálů [6]. L-systémy budou blíže popsány dále v podkapitole 2.3.

2.2.4 Backusova-Naurova forma

Backusova-Naurova forma (dále jen BNF) je notace pro vyjádření gramatiky jakyza ve formě produkčních pravidel. BNF gramatiky se skádají z terminálů, což jsou položky, které se mohou vyskytnout v jazyku (např. +, -, atp.) a neterminálů, které se mohou rozgenerovat na jeden nebo více terminálů nebo neterminálů.

Gramatika se dá vyjádřit jako čtveřice $\{N, T, P, S\}$, kde

- N je množina neterminálů,
- T je množina terminálů,
- P je množina produkčních pravidel, které mapují prvky z N na T,
- S je počáteční symbol z množiny N.

Pokud existuje více pravidel aplikovatelných na symbol z N, jsou možnosti odděleny znakem `"|"` [7].

2.3 L-systémy

Formálně jsou tyto systémy definovány gramatikou, trojicí $G = (\Sigma, \Pi, \alpha)$, kde

- Σ je abeceda symbolů,
- Π je množina přepisovacích pravidel,
- α je počáteční konfigurace (axiom).

Nechť Σ^* je množina všech slov nad Σ a Σ^+ je množina všech neprázdných slov nad Σ. Axiom α patří do množiny Σ^+. Pravidlo $\pi \in \Pi$ je mapování symbolu $s \in \Sigma$ na slovo $s' \in \Sigma^*$, tj. $\pi : \Sigma \mapsto \Sigma^*$. Existuje mnoho rozšíření L-systému, např. kontextové nebo parametrizované L-systémy [6].

2.3.1 DOL-systém

Tato část se bude zabývat nejjednodušší třídou L-systémů nazývanou DOL-systémy. DOL-systémy jsou bezkontextové a deterministické.

Nechť V označuje abecedu, V^* množinu všech slov nad V a V^+ množinu všech neprázdných slov nad V. *OL-systém* je uspořádaná trojice $G = \langle V, \omega, P \rangle$, kde V je *abeceda* systému, $\omega \in V^+$ je neprázdné slovo nazývaný *axiom* a $P \subset V \times V^*$ je konečná množina přepisovacích pravidel. Pravidlo $(a, \chi) \in P$ se zapisuje jako $a \to \chi$. Písmeno a a slovo χ se nazývají *předchůdce* a *následník* tohoto pravidla. Platí, že $\forall a \in V$ existuje právě jedno slovo $\chi \in V^*$, takové, že platí $a \to \chi$, pokud existuje právě alespoň jedno slovo, jedná se o OL-systém, tzn. nederministický [9].

Následující příklad ilustruje operace nad DOL-systémem. Mějme zadaný následující L-systém:

$$\begin{aligned} &\omega : a_r \\ &p_1 : a_r \to a_l b_r \\ &p_2 : a_l \to b_l a_r \\ &p_3 : b_r \to a_r \\ &p_4 : b_l \to a_l \end{aligned} \tag{2.1}$$

Počínaje prvním symbolem a_r (axiomem) je postupně vygenerována následující sekvence slov:

$$\begin{aligned} &a_r \\ &a_l b_r \\ &b_l a_r a_r \\ &a_l a_l b_r a_l b_r \\ &b_l a_r b_l a_r a_r b_l a_r a_r \\ &\dots \end{aligned}$$

2.3.2 Stochastické L-systémy

Stochastický L-systém je uspořádaná čtveřice $G_\pi = \langle V, \omega, P, \pi \rangle$. Abeceda V, axiom ω a množina přepisovacích pravidel P jsou definovány tak jako u OL-systému (viz 2.3.1). Funkce $\pi : P \to (0, 1\rangle$ zvaná *rozdělení pravděpodobnosti* mapuje množinu pravidel na množinu *náhodností pravidel*. Předpokládá se, že pro písmeno $a \in V$ je součet pravděpodobností roven 1 [9].

Příklad stochastického systému, ve kterém bude mít každé pravidlo 25% šanci na použití:

$$\begin{aligned} &\omega : a \\ &p_1 : a \xrightarrow{.25} b \\ &p_2 : a \xrightarrow{.25} c \\ &p_3 : a \xrightarrow{.25} d \\ &p_4 : a \xrightarrow{.25} e \end{aligned} \tag{2.2}$$

2.3.3 Kontextové L-systémy

Kontextové L-systémy jsou podobné DOL-systémům, s tím rozdílem, že mohou vyžadovat, aby symbolu v pravidle předcházel určitý řetězec, nebo aby naopak určitý řetězec následoval [9].

1L-systémy

1L-systémy mají jednostranný kontext a pravidla se zapisují ve tvaru $a_l < a \rightarrow \chi$ nebo $a > a_r \rightarrow \chi$, při použití prvního tvaru se vyžaduje, aby symbolu a předcházel symbol a_l a ve druhém tvaru, aby naopak následoval symbol a_r [9].

Následující příklad ilustuje propagaci symbolu řetězcem:

$$\begin{aligned} \omega &: baaaaaa \\ p_1 &: b < a \rightarrow b \\ p_2 &: b \rightarrow a \end{aligned} \tag{2.3}$$

Ukázka několika řetězců vygenerovaných tímto systémem:

$$\begin{aligned} &baaaaaa \\ &abaaaaa \\ &aabaaaa \\ &aaabaaa \\ &\dots \end{aligned}$$

2.3.4 Parametrické L-systémy

Parametrické L-systémy pracují s *parametrickýma slovama*, což jsou řetězce *modulů* skládajících se z *písmen* s přiřazenými *parametry*. Písmena patří do abecedy V a parametry patří do množiny *reálných čísel* $\Re$. Modul s písmenem $A \in V$ a parametry $a_1, a_2, \dots, a_n \in \Re$ je zapsán jako $A(a_1, a_2, \dots, a_n)$. Každý modul patří do množiny $M = V \times \Re^*$, kde $\Re^*$ je množina všech konečných sekvencí parametrů. Množina všech řetězců modulů a množina všech neprázdných řetězců se zapisují jako $M^* = (V \times \Re^*)^*$ resp. $M^+ = (V \times \Re^*)^+$.

Parametrický 0L-systém je definován jako uspořádaná čtveřice $G = \langle V, \Sigma, \omega, P \rangle$, kde

- V je abeceda systému,
- Σ je množina parametrů,
- $\omega \in (V \times \Re^*)^+$ je neprázdná množina parametrických slov nazývaná *axiom*,
- $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \times (V \times \mathcal{E}(\Sigma)^*)^*$ je konečná množina produkčních pravidel. $\mathcal{C}(\Sigma)$ označuje logický výraz s parametry z množiny Σ. $\mathcal{E}(\Sigma)$ označuje aritmetický výraz s parametry z množiny Σ.

Symboly : a → jsou použité pro oddělení 3 komponent pravidla: *předchůdce*, *podmínky* a *následovníka*. Pravidlo má tedy tvar

předchůdce : podmínka → následovník

Příklad pravidla s předchůdcem $A(t)$, podmínkou $t > 5$ a následovníkem $B(t+1)CD(t \wedge 0.5, t-2)$ se zapíše jako

$$A(t) : t > 5 \rightarrow B(t+1)CD(t \wedge 0.5, t-2) \tag{2.4}$$

[8]

2.4 Výstavba překladače

V této podkapitole budou popsány části překladače, které je potřeba implementovat pro vytvoření překladače zdrojových souborů.

2.4.1 Lexikální analýza

Lexikání analyzátor načítá řetězec znaků tvořící vstupní soubor a tvoří z něj tzv. lexémy. Každý z vytvořených lexému je reprezentován tokenem, což je dvojice tvořena jménem tokenu a hodnotou. Jméno tokenu je využíváno při syntaktické analýze a hodnota je využívána při kontrole správnosti sémantiky [2].

2.4.2 Syntaktická analýza

Syntaktický analyzátor využivá první složku tokenu k vybudování stromové reprezentace gramatické struktury řetežce tokenů. Typickou reprezentací je tzv. syntaktický strom, ve kterém uzel reprezentuje operaci a potomci reprezentují argumenty operace [2].

2.4.3 Sémantická analýza

Sémantický analyzátor využívá syntaktický strom a informace v něm uložené ke kontrole sémantické konzistence s definicí jazyka.

Důležitou části sémantické analýzy je typová kontrola, která kontroluje, že každý operátor má příslušné operandy např. mnoho programovacích jazyků vyžaduje, aby indexem do pole bylo celé číslo - kompilátor musí vypsat chybovou hlášku, pokud je jako index použito desetinné číslo [2].

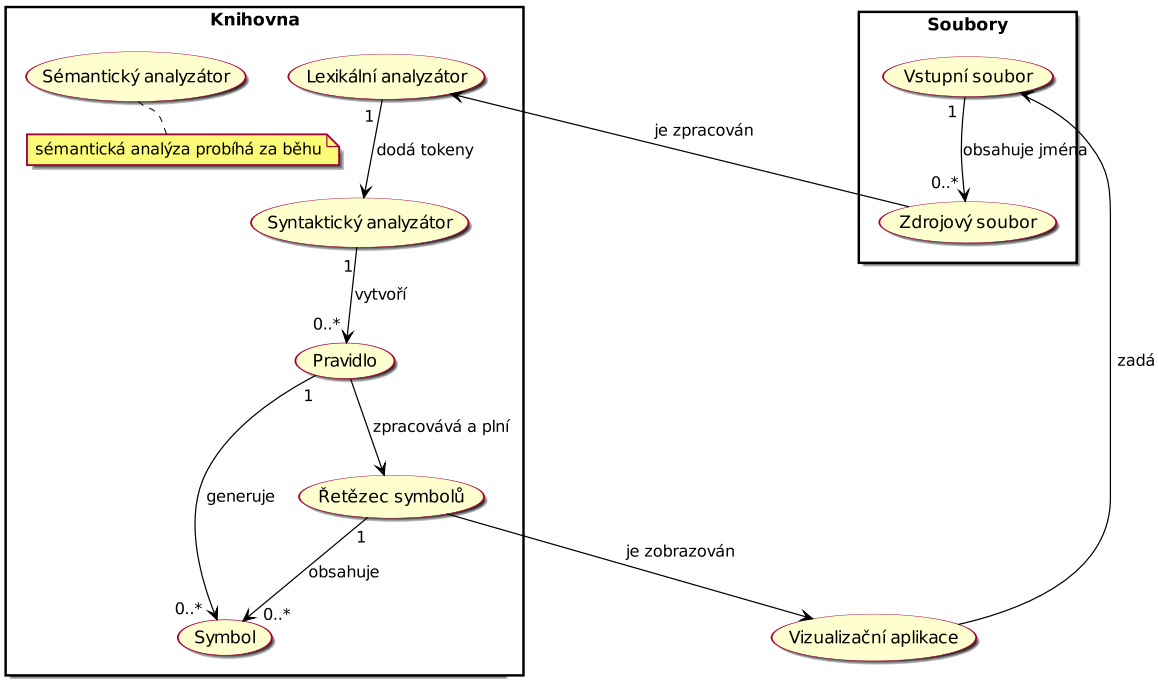
Kapitola 3

Návrh systému pro procedurální generování

Tato kapitola se bude zabývat návrhem vstupního souboru a souborů zdrojových, požadavky na knihovnu a způsobem zpracování souborů.

Syntaxe souborů popisující pravidla bude založena na jazyku C. Kompletní syntaxi zapsánu v Backus-Naurově formě lze nalézt v příloze (viz B).

3.1 Návrh struktury programu



Obrázek 3.1: Návrh struktury

3.2 Vstupní soubor

Jak již bylo zmíněno, vstupní soubor bude tvořen výčtem jednotlivých zdrojových souborů. Každý zdrojový soubor bude dále zpracován lexikálním, syntaktickým a sémantickým analyzátorem. Možnost zadávat soubory samostatně vede k lepší dekompozici a větší kontrole nad soubory.

3.3 Zdrojový soubor

Ve zdrojových souborech bude definováno nastavení generování společně s definicí symbolů a pravidel, které s těmito symboly budou pracovat.

3.3.1 Nastavení

Součástí zdrojových souborů bude také možnost nastavit informace důležité pro vygenerování modelu. Pro vývojáře bude zpřístupněna možnost zapnutí debugovacích výpisů sloužících pro případné zjištění příčin chybného běhu aplikace, např. pokud bude aplikace předčasně ukončena, bude možno dohledat, ve které funkci došlo k chybě. Pro uživatele bude určena možnost povolení výpisu informativních hlášek, např. které pravidlo se právě aplikuje, kolikátá iterace právě probíhá a další. Samozřejmě nebude chybět možnost povolit export modelu.

Vzhledem ke stylu přepisovacích pravidel je potřeba brát v potaz, že by mohlo dojít k nekonečnému aplikování pravidel, proto se doporučuje, zadat v nastavení maximální počet iterací, aby se zacyklení předešlo.

Pokud se při vygenerovávání modelu bude využívat náhodnost, je možné zadat tzv. seed hodnotu. Touto hodnotou se bude inicializovat náhodný generátor čísel. Pokud se bude seed hodnota rovnat nule, bude se generátor náhodných čísel inicializovat hodnotou vypočítanou ze současného systémového času, takže bude velmi malá pravděpodobnost, že při opětovném spuštění generování modelu bude vygenerována stejná posloupnost náhodných čísel, jako při předchozím vygenerování. Pokud je však seed nastaven na určitou hodnotu, bude sekvence náhodných čísel vygenerována pokaždé totožná, toto je výhodné, pokud se například při vygenerování vyskytne chyba a je potřeba tuto chybu opravit.

3.3.2 Symboly

Jelikož je u symbolů potřeba uchovávat jak parametry, tak styl, jakým se z těchto parametrů vygeneruje množina polygonů, které budou tvořit část modelu, nebude stačit pouze ekvivalent struktury z jazyku c, resp. prostý výčet proměnných/struktur.

Pro definici symbolu bude potřeba nejdříve nadefinovat jeho typ, poté bude následovat blok, který bude obsahovat deklarace a definice interních proměnných a vnořených symbolů/struktur. K těmto interním prvkům bude možno přistupovat. Další blok u definice symbolu bude určovat, jakým způsobem budou interní prvky zpracovány, při vygenerovávání výsledného modelu z řetězce symbolů. V tomto bloku budou definovány vrcholy a způsob, jakým budou propojeny - vytvoření polygonů. Ke každému symbolu je vytvořen objekt obsahující vygenerované polygony, tzv. MESH.

3.3.3 Pravidla

U každého pravidla bude definována tzv. délka určující, k jakémů počtu symbolů ze vstupního řetězce bude mít pravidlo přístup. Následuje blok, po jehož provedení bude možno zjistit, zda lze dané pravidlo aplikovat na řetězec symbolů nebo ne. Ve druhém bloku bude definována akce, která se má při aplikování pravidla provést, např. přidání dalších symbolů do řetězce symbolů.

3.4 Návrh knihovny

Jak již bylo zmíněno, cílem je vytvořit knihovnu, která ze vstupního souboru vygeneruje model. Jako vstupní parametr bude knihovna požadovat název vstupního souboru, po předání jména souboru knihovna vstupní soubor a zdrojové soubory zpracuje a poskytne uživateli možnost, zažádat si o seznam Meshů, které obsahují polygony.

3.5 Vizualizační aplikace

Vizualizační aplikace bude tvořena pouze jedním oknem, ve kterém bude zobrazen model vygenerovaný pomocí knihovny pro procedurální generování architektury. Po zpracování vstupního souboru knihovnou bude ze získaných polygonů vytvořen a zobrazen model uživateli.

Kapitola 4

Implementace

Práce je implementována v jazyce C++. Pro lexikální a syntaktickou analýzu jsou použité knihovny Flex a Bison. Pro vizuální aplikaci je využito knihoven SDL2 a OpenGL. Knihovna vznikala v operačním systému Ubuntu a později byla upravena tak, aby fungovala i v systému Windows 7. Dokumentace je vytvořena v sázecím jazyce LaTeX. Při vývoji byl využíván verzovací nástroj Git poskytovaný webovou službou GitHub.

Pro kompletní funkcionalitu knihovny je potřeba zajistit korektní načtení vstupního souboru a jeho zpracování. Jelikož je pro vstupní soubor vytvořena speciální syntaxe, vycházející z jazyka C, je potřeba vymyslet způsob, jak tento soubor načíst a interpretovat. Využitím lexikálního analyzátoru (scanneru), syntaktického (parser) a sémantického analyzátoru se v jazyce C++ vytvoří příslušné struktury, reprezentující data ze zdrojového souboru a tyto struktury je možno dále zpracovávat.

V následující části bude popsán způsob implementace návrhu celého systému a knihovny, které byly při implementaci využity. Je nutno poznamenat, že pro překlad knihovny je potřeba mít nainstalovány knihovny Flex a Bison.

4.1 Lexikální analýza

Při lexikální analýze se využívá knihovna Flex (fast lexical analyzer), která při nalezení řetězce znaků odpovídajících zadaným pravidlům, vyvolá libovolnou akci (navrácení tokenu - i s případnou hodnotou, vypsání chybové hlášky, atp.). Všechna pravidla jsou zahrnuta v příloze (viz A).

Pravidla, podle kterých se tokeny generují, dodržují následující syntaxi:

regulární výraz { sekvence příkazů }

při nalezení sekvence znaků, které vyhovují danému regulárnímu výrazu, dojde k vyvolání sekvence příkazů. Příklad konkrétního pravidla:

[0-9]+.[0-9]+ { yylval->fval=atof(yytext); return FLOAT; }

pokud je řetězec složen z 1 a více čísel následovaných tečkou a 1 a více čísly, dojde k vytvoření tokenu typu desetinné číslo a zároveň je v proměnné fval uchována jeho hodnota.

4.2 Syntaktická analýza

Pro syntaktickou analýzu je využito knihovny Bison. Bison je generátor syntaktického analyzátoru, který převádí bezkontextovou gramatiku v Backus-Naurově formě 2.2.4 na LR nebo obecný LR syntaktický analyzátor [3, p. 1].

Při kontrole syntaxe zdrojového souboru se vytváří stromová struktura tvořena instancemi specifických tříd (viz. 4.4).

Příklad pravidla zapsaného v Backus-Naurově formě:

<symbol> :== výraz | výraz ...

Místo slova „výraz“ se může vyskytovat jeden nebo více terminálních nebo neterminálních symbolů. Na levé straně výrazu je vždy neterminál. Pokud lze pravidlo aplikovat na více výrazů, je potřeba tyto výrazy oddělit znakem „|“.

Přiklad pravidla ze vstupního souboru pro knihovnu bison:

```
variable: type IDENTIFIER SEMICOLON
    {
        // Vytvoreni instance tridy Variable
        // S typem $1 a jmenem $2
    }
    | type IDENTIFIER EQUAL expr SEMICOLON
    {
        // Vytvoreni instance tridy Variable
        // S typem $1 a jmenem $2 a hodnotou $4
    }
    ...

type
    : INT    {$$ = "int";    }
    | FLOAT  {$$ = "float";  }
    | STRING {$$ = "string";}
```

Pokud je možno aplikovat pravidlo na přicházející tokeny je umožněno vyvolání určité akce, která je zapsána jako sekvence přikazů v bloku ihned za pravidem. Je možno aplikování klasických funkcí z jazyku C++, ale je možno využít speciálních konstrukcí. Například zapsáním hodnoty do proměnné $$ je nadřazenému pravidlu umožněn přístup k hodnotě uložené v této proměnné . Přístup k této hodnotě z nadřazeného pravidla je umožněn přes proměnnou $i, kde i je index terminálu/neterminálu ve výrazu na pravé straně pravidla. V příkladu je z pravidla „type“ navrácen datový typ proměnné, dle kterého je v pravidle „variable“ vytvořena proměnná příslušného datového typu.

4.3 Sémantická analýza

Jak již bylo zmíněno jazyk popisující zdrojové soubory je velmi inspirován jazykem C. Jedná se tedy o silně typovaný jazyk, ve kterém tudíž nelze do proměnné jednoho datového typu přiřadit datový typ jiný, např. do celočíselného typu nelze zapsat desetinné číslo. Při nalezení sémantické chyby dojde k vypsání chybové hlášky a ukončení programu. Odlišností od jazyka C je fakt, že sémantická analýza je prováděna za běhu aplikace.

4.4 Interpret

V následující části budou popsány třídy, ze kterých jsou vytvářeny instance během syntaktické analýzy a tyto instance jsou potřebné ke správnému vygenerování modelu.

4.4.1 Variable

Variable označuje třídu, která reprezentuje proměnnou. Stejně tak, jako třída Symbol si uchovává jméno a typ. Jakožto třída dědící od třídy Function (viz 4.4.3) se třída Variable zavazuje k implementaci operátoru (), při jehož vyvolání dojde k vytvoření proměnné, která je přidána do příslušícího bloku.

4.4.2 Symbol

Symbol je ve své podstatě objekt, ve kterém mohou být uchovány nejen proměnné, ale i další symboly.

Hlavní využití nalezá třída symbol při definování prvků v řetězci. Při iteracích jsou na symboly, nebo pouze jeden symbol, v řetězci aplikována pravidla, která mohou tento řetězec modifikovat. Pokud již není možná aplikace jakéhokoliv pravidla, dojde k interpretaci symbolů v řetězci (viz 4.4.4).

Velmi důležitá je funkce `makeCopy`, která vytváří identické kopie symbolu. Tohoto je využito při vytváření instancí určitého symbolu z jeho šablony, která obsahuje deklarace a definice interních symbolů a proměnných.

4.4.3 Function

Třída Function je *funktor*, tzn. jedná se o třídu, která má nadefinován operátor `()`. Při aplikaci tohoto operátoru na instanci objektu této třídy, dojde k vykonání určité akce. Funkce jsou rozděleny do několika kategorií podle typu vykonané akce.

Function slouží výhradně jako šablona pro vytváření specifičtějších tříd, definuje funkce, které se potomci této třídy zavazují implementovat a také obsahuje enumeraci obsahující názvy všech dceřiných tříd, kterou využívají konstruktory dceřiných tříd k uchování názvu třídy.

Basic

Do této kategorie patří základní funkce. Velmi důležité jsou funkce vyjadřující cykly (while, for) a větvení (if). Základem těchto funkcí je podmínka, která je vyhodnocena a podle jejího výsledku je provedena příslušná akce. Akce je definována jako instance třídy Block a je provedena, pokud je podmínka kladná. Třída If obsahuje může obsahovat 2 akce, jedna se provede, pokud je podmínka pravdivá a druhá, pokud je nepravdivá.

Funkce Print slouží k vypsání hodnot do konzole. Umožňuje vypsání přímo zadaných hodnot, hodnot proměnných a strukturovaný výpis interních proměnných a symbolů v symbolu.

Return ukončuje vyhodnocování funkcí v bloku a také celému blocku nastavuje návratovou hodnotu (pokud se jedná o vnořené bloky, je návratová hodnota propagována do vyšších vrstev).

Advanced

Do této kategorie patří pokročilé funkce.

V současném stádiu se zde vyskytuje jediná funkce a to ROTATE, která umožňuje rotaci vektoru, kolem vektoru jiného a určitý úhel. Funkce lze uplatnit například, pokud je potřeba vygenerovat model sloupu. Stačí nadefinovat jedinou stranu a otočit ji několikrát kolem bodu.

VarSym

V této kategorii se vystytují nejen třídy popisující proměnné a symboly, ale i funkce pro jejich vytváření.

Manipulation

Zde patří funkce, které určitým způsobem modifikují nebo zprostředkovávají přístup k proměnným a symbolům.

Mesh

Vždy, když je ze symbolu vytvářena část výsledného modelu, je pro tento symbol vytvořena instance třídy MESH a právě funkce z této kategorie do této instance přidávají vrcholy (MESHADDVERTEX) a definují polygony, z těchto vrcholů složené (MESHADDFACE). Z instance třídy MESH je poté možno tyto polygony získat a zobrazit.

Output

Funkce, které určitým způsoben pracují s řetězcem symbolů se vyskytují v této kategoii.

Funkce pro přidávání symbolů do řetězce symbolů určeného pro zpracování v další iteraci nebo konečného řetězce se nazývá OUTPUTADD.

Pro zjištění, zda je současně zpracovávaný řetězec symbolů prázdný slouží funkce OUTPUTEMPTY. Tímto lze vytvořit pravidlo, které je možno aplikovat při prvnotním spuštění zpracovávání řetězce symbolů (řetězec symbolů je prázdný).

4.4.4 Block

Hlavní úlohou BLOCKu je sdružení funkcí do vektoru funkcí. Na každou funci v tomto vektoru je aplikován operarátor `()` čímž dojde k provedení příslušné akce popsané v implementaci funkce.

Ke každému BLOCKu je navíc přidružen speciální symbol (viz 4.4.2), který v sobě uchovává proměnné a další symboly. K obsahu symbolu (proměnným a vnořeným symbolům) mají přístup funkce uložené v BLOCKu, ke kterému symbol přísluší a funkce z bloků vnořených.

Interpretation

INTERPRETATION je podtřída třídy BLOCK, která obsahuje instrukce, které slouží k vygenerování modelu z informací (proměnných a symbolů) obsažených ve zdrojovém symbolu. Instance této třídy je vytvořena vždy, když se při definování symbolu vyskytuje 2. blok.

4.4.5 Rule

Ukázka defice pravidla ve zdrojovém souboru:

```
rule pravidlo[length] {
        // ConditionBlock
}{
        // ActionBlock
}
```

Rule je třída obsahující 2 instance třídy Block - tzv. conditionBlock a actionBlock. Dále obsahuje tzv. délku (length). Délka určuje, kolik symbolů ze symbolu řetězců bude přístupných v blocích. Pokud nelze symboly zpřístupnit (délka je větší než celkový počet symbolů v řetězci nebo by bylo potřeba přistupovat k symbolům mimo řetězec symbolů) je pravidlo neaplikovatelné. Při zpřístupňování symbolů se vždy používá pravý kontext, tzn. vždy jsou zpřístupněny prvky n až $n + l - 1$ z řetězce symbolů, kde n udává pozici právě testovaného symbolu a l udává délku.

Při testování, zda lze pravidlo aplikovat nebo ne, je potřeba na Block conditionBlock použít operátor (), tímto dojde ke zpracování blocku. Jelikož má Block návratovou hodnotu, je možné si o ni zažádat a podle její hodnoty lze určit, zda pravidlo lze aplikovat.

Pokud je návratová hodnota Blocku conditionBlock `true`, je vyhodnocen actionBlock, který může modifikovat řetězec symbolů. Vždy je z původního řetězce odebrán určitý počet symbolů, který je dán hodnotou proměnné délka. Pokud actionBlock vytváří symboly nové, jsou tyto symboly přidány do řetězce symbolů určeného pro další iteraci nebo do výsledného řetězce symbolů, pokud se jedná o iteraci konečnou.

4.4.6 Interpretace

Při začátku interpretace jsou vytvořeny 2 řetězce symbolů. Jeden je určen jako vstupní řetězec a druhý jako vstupní řetězec pro následující iteraci.

Pro každé pravidlo je vyhodnocena část určující, zda může být tělo pravidla aplikováno na řetězec. Pokud je možných pravidel více, je určitým způsobem vybráno jedno z nich a jeho tělo je aplikováno na vstupní řetězec. Pokud pravidlo vytváří nové symboly, tak jsou tyto symboly přídány do řetězce určeného pro další iteraci.

Když se aplikuje pravidlo je posunut ukazatel do řetězce symbolů o délku pravidla (počet symbolů, které jsou symbolu poskytnuty), pokud by pozice ukazatele byla větší než je délka řetězce, pokračuje se další iterací. Další iterací se pokračuje v případě, kdy již nelze aplikovat další pravidla nebo pokud byl zpracován celý řetězec.

Před další iterací je jako vstupní řetězec nastaven výstupní řetězec z iterace minulé a je vytvořen nový výstupní řetězec pro současnou iteraci.

Po skončení všech iterací se přejde ke zpracování symbolů ve výstupním řetězci. Nad každým symbolem je vyvolána část, která z informací v tomto symbolu obsažených vytvoří instanci třídy Mesh.

4.5 Knihovna

Knihovna je implementována jako knihovna statická. Aby mohla být použita, je potřeba ji nejdříve přeložit pomocí příkazu make. Následně je ve složce *libs* vytvořen soubor *libargen.a*.

Pro samotné využití knihovny je zapotřebí ve zdrojovém souboru pomocí direktivy `#include` zahrnout hlavičkový soubor knihovny argen.h nacházející se ve složce *src*. Aby

šel zdrojový kód využívající knihovnu přeložit je potřeba, aby v jeho souboru *Makefile* byla knihovna staticky linkována, např. takto `-Wl,-Bstatic -L./argen/libs -largen`.

4.6 Vizualizační aplikace

Vizualizační aplikace není součástí knihovny. Jedná se o samostatnou aplikaci využívající implementovanou knihovnu. Při spouštění z příkazové řádky je potřeba aplikaci zadat cestu k souboru, který obsahuje soubory potřebné k vygenerování modelu. Pro samotné vygenerování modelu je potřeba při vytváření instance třídy Argen předat cestu ke vstupnímu souboru. Po zpracování vstupu knihovnou je z této knihovny získám seznam Meshů a z nich jsou získány polygony, které se pomocí openGL zobrazí v okně uživateli.

4.6.1 SDL2

SDL (Simple DirectMedia Layer) je multiplatformní knihovna poskytující nízkoúrovňový přístup k audiu, klávesnici, myší, joysticku a grafickému hardwaru skrze OpenGL a Direct3D. Je používána video přehrávači, emulátory a počítačovými hrami.

SDL podporuje operační systémy Windows, Mac OS X, Linux, iOS, a Android. Knihovna je napsána v jazyce C a nativně funguje v jazyce C++. Lze používat i s jazyky C# a Python [1].

V této práce je SDL využito pro vytvoření okna aplikace a poskytnutí OpenGL kontextu, na který lze pomocí OpenGL vykreslovat.

4.6.2 OpenGL

OpenGL (Open Graphics Library) je API (Application Programming Interface - rozhraní pro programování aplikací) ke grafickému hardwaru. API se skládá ze stovek procedur a funkcí, které vývojáři umožňují specifikovat *shader* programy, objekty a operace potřebné k produkování trojdimenzionálních objektů vysoké kvality.

Většína OpenGL funkcí vyžaduje, aby grafický hardware obsahoval *framebuffer*, na jehož vlastnostech závisí způsob vykreslování bodů, čar a polygonů [11].

Kapitola 5

Export

Jako výstupní formát exportovaného modelu byl zvolen Wavefront obj. *Obj* soubory definují geometrii a další vlastnosti objektu. Tyto soubory mohou být v ASCII formátu nebo ve formátu binárním [12].

Při vytváření výstupního souboru je potřeba postupně zpracovat všechny instance třídy Mesh. Nejprve je do obj souboru přidán řádek se jménem, které indentifikuje konkrétní část modelu, jméno je rovno názvu instance symbolu. V dalším kroku jsou do souboru přidány informace o všech vrcholech, které jsou v instanci Mesh popsány a nakonec jsou přidány informace o polygonech, které určují, jak jsou vrcholy propojeny.

V následujícím příkladu budou demonstrovány kontrukce obj souboru, které jsou využity při exportu modelu. Mějme symbol a jeho interpretaci definovánu následovně:

```
symbol Square {
        float width = 1.0;
        float depth = 1.0;
}{
        vec3 p0 = {0.0, 0.0, 0.0};
        vec3 p1 = {width, 0.0, 0.0};
        vec3 p2 = {width, depth, 0.0};
        vec3 p3 = {0.0, depth, 0.0};

        _mesh.addVertex(p0, p1, p2, p3);
        _mesh.addFace(0, 1, 2, 3);
}
```

Pokud bude ve výstupním řetězci symbolů vytvořena instance symbolu Square umístěná v počátku souřadného systému a pojmenovaná jako „square1“, budou se výstupním souboru vyskytovat tyto řádky:

```
g square1
v 0.000000 0.000000 0.000000
v 1.000000 0.000000 0.000000
v 1.000000 1.000000 0.000000
v 0.000000 1.000000 0.000000
f 1 2 3
f 1 3 4
```

Při importování exportovaného modelu do 3D editoru dojde k vytvoření objektu se jménem „square1“, který obsahuje 2 polygony tvořené 4-mi vrcholy.

Kapitola 6

Závěr

V rámci bakalářské práce byla vytvořena knihovna pro procedurální generování architektury, které ze vstupních souborů získá potřebné informace o symbolech, jejich interpretaci a pravidlech pro modifikaci řetězce těchto symbolů. Vizualizační aplikace finální model zobrazí uživateli. Knihovna poskytuje možnost exportování modelu do *obj* souboru, který lze importovat do 3D editoru a dále s modelem pracovat.

Jelikož se jedná o poměrně složitý projekt, lze jej v mnoha ohledech zdokonalit a rozšířit např. o procedurální generování textur a jejich aplikaci na model nebo o možnost importování již hotových modelů, které by šly využít pro generování složitějších modelů. Vizuální aplikace by jistě šla také zdokonalit, například o možnost volného pohybu kolem modelu.

K této práci bylo vytvořeno video zobrazující postup vytváření modelu panelového domu postupným přidáváním symbolů a detailů částem modelu ze symbolů generovaných. Ve videu je využito 11 modelů, které zobrazují jednotlivá stádia modelu od nejjednoduššího (pouhého kvádru) až po výsledný panelový dům. Pro každý z těchto modelů bylo vytvořeno 10s video obsahující model, kolem kterého je otáčena kamera. Pro renderování modelů byl využit 3D editor Blender. Pro vytvoření finálního videa byl využit jednoduchý volně dostupný nástroj pro editaci videa OpenShot Video Editor.

Open-source licence, pod kterou je knihovna vydána a zdrojové soubory knihovny jsou veřejně přístupné na adrese `https://github.com/xrorec00/procedural-architecture`.

Literatura

[1] Introduction to SDL 2.0. `https://wiki.libsdl.org/Introduction`.

[2] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006, iSBN 0321486811.

[3] Donnelly, C.; Stallman, R.: *Bison*. Free Software Foundation, 2015, iSBN 1-882114-44-2.

[4] Lagae, A.; Lefebvre, S.; Cook, R.; aj.: A Survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*, ročník 29, č. 8, 2010: s. 2579–2600.

[5] Meduna, A.: *Formal languages and computation : models and their applications*. Boca Raton : CRC Press, 2014, iSBN 978-1-4665-1345-7.

[6] Nivoliers; C.Gérot; V.Ostromoukhov; aj.: L-system specification of knot-insertion rules for non-uniform B-spline subdivision. *Computer Aided Geometric Design*, ročník 29, č. 2, 2012: s. 150–161.

[7] O'Neill, M.; Ryan, C.: Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*, ročník 5, č. 4, 2001: str. 350.

[8] Prusinkiewicz, P.; Hammel, M.; Hanan, J.; aj.: L-Systems: From The Theory To Visual Models Of Plants. 1996.

[9] Prusinkiewicz, P.; Lindenmayer, A.; Hanan, J. S.; aj.: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990, iSBN 0-387-972-978.

[10] Rouse, M.: Noise. `http://whatis.techtarget.com/definition/noise`.

[11] Segal, M.; Akeley, K.: The OpenGL Graphics System : A Specification. `https://www.opengl.org/registry/doc/glspec45.core.pdf`.

[12] WWW stránky: Obj specifikace. `http://www.martinreddy.net/gfx/3d/OBJ.spec`.

Přílohy

Seznam příloh

Příloha A

Seznam tokenů

Regular expression	Token
`[ \t]`	(pass)
`"//".*`	(pass)
`\n`	(pass)
`rule`	`RULE`
`return`	`RETURN`
`true`	`BOOL`
`false`	`BOOL`
`print`	`PRINT`
`if`	`IF`
`else`	`ELSE`
`for`	`FOR`
`while`	`WHILE`
`symbol`	`SYMBOL`
`int`	`INT`
`float`	`FLOAT`
`string`	`STRING`
`"&&"`	`AND`
`"||"`	`OR`
`"_output.empty"`	`OUTPUTEMPTY`
`"_output.add"`	`OUTPUTADD`
`"_output.get"`	`OUTPUTGET`
`randomInt`	`RANDOMINT`
`randomFloat`	`RANDOMFLOAT`
`rotate`	`ROTATE`
`"_mesh.addFace"`	`MESHADDFACE`
`"_mesh.addVertex"`	`MESHADDVERTEX`
`output`	`OUTPUT`
`settings`	`SETTINGS`
`[0-9]+\.[0-9]+`	`FLOAT`
`[0-9]+`	`INT`
`[$]?[a-zA-Z0-9_]+`	`IDENTIFIER`
`\".+\"`	`STRING`

`"("`	`LPAR`
`")"`	`RPAR`
`"["`	`LSQBRAC`
`"]"`	`RSQBRAC`
`"{"`	`LBRAC`
`"}"`	`RBRAC`
`";"`	`SEMICOLON`
`":"`	`COLON`
`","`	`COMMA`
`"++"`	`PLUSPLUS`
`"+"`	`PLUS`
`"-"`	`MINUS`
`"/"`	`DIVIDE`
`"*"`	`MULTIPLY`
`"=="`	`EQUALITY`
`"!="`	`NEQUALITY`
`"<"`	`LESS`
`"<="`	`LESSEQUAL`
`">"`	`GREATER`
`">="`	`GREATEREQUAL`
`"="`	`EQUAL`
`"."`	`DOT`
`.`	(invalid token)

Příloha B

Syntaxe jazyku

⟨*main*⟩ ::= ⟨*settings*⟩ ⟨*main*⟩
| ⟨*symbol*⟩ ⟨*main*⟩
| ⟨*rule*⟩ ⟨*main*⟩
| ε

⟨*settings*⟩ ::= SETTINGS LBRAC ⟨*settingsItemsOrEmpty*⟩ RBRAC

⟨*settingsItemsOrEmpty*⟩ ::= ⟨*settingsItems*⟩
| ε

⟨*settingsItems*⟩ ::= ⟨*settingsItems*⟩ ⟨*settingsItem*⟩
| ⟨*settingsItem*⟩

⟨*settingsItem*⟩ ::= IDENTIFIER EQUAL BOOL SEMICOLON
| IDENTIFIER EQUAL INT SEMICOLON

⟨*symbol*⟩ ::= SYMBOL ⟨*createSymbol*⟩ ⟨*setSymbolType*⟩ LBRAC ⟨*symVariablesOrEmpty*⟩ RBRAC ⟨*symbolInterpretation*⟩
| SYMBOL ⟨*createSymbol*⟩ ⟨*setSymbolType*⟩ SEMICOLON

⟨*setSymbolType*⟩ ::= IDENTIFIER

⟨*createSymbol*⟩ ::= ε

⟨*symVariablesOrEmpty*⟩ ::= ⟨*symVariables*⟩
| ε

⟨*symVariables*⟩ ::= ⟨*symVariables*⟩ ⟨*variable*⟩
| ⟨*variable*⟩

⟨*symbolInterpretation*⟩ ::= ⟨*createInterpretation*⟩ LBRAC ⟨*codeLinesOrEmpty*⟩ RBRAC
| ε

⟨*createInterpretation*⟩ ::= ε

⟨*rule*⟩ ::= RULE IDENTIFIER ⟨*ruleLength*⟩ LBRAC ⟨*createBlock*⟩ ⟨*codeLinesOrEmpty*⟩ RBRAC LBRAC ⟨*createBlock*⟩ ⟨*codeLinesOrEmpty*⟩ RBRAC

⟨*ruleLength*⟩ ::= ε
  | LSQBRAC INT RSQBRAC

⟨*createBlock*⟩ ::= ε

⟨*codeLinesOrEmpty*⟩ ::= ⟨*codeLines*⟩
  | ε

⟨*codeLines*⟩ ::= ⟨*codeLine*⟩
  | ⟨*codeLine*⟩ ⟨*codeLines*⟩

⟨*codeLine*⟩ ::= ⟨*variable*⟩
  | ⟨*if*⟩
  | ⟨*for*⟩
  | ⟨*while*⟩
  | ⟨*print*⟩
  | ⟨*return*⟩
  | ⟨*output*⟩
  | ⟨*mesh*⟩
  | ⟨*increment*⟩ SEMICOLON

⟨*variable*⟩ ::= ⟨*type*⟩ IDENTIFIER SEMICOLON
  | ⟨*type*⟩ IDENTIFIER EQUAL ⟨*expr*⟩ SEMICOLON
  | ⟨*multilevelIdentifier*⟩ EQUAL ⟨*expr*⟩ SEMICOLON
  | IDENTIFIER IDENTIFIER SEMICOLON
  | IDENTIFIER IDENTIFIER EQUAL ⟨*initializerList*⟩ SEMICOLON
  | OUTPUTGET LPAR INT RPAR

⟨*type*⟩ ::= INT
  | FLOAT
  | STRING

⟨*if*⟩ ::= IF LPAR ⟨*expr*⟩ RPAR LBRAC ⟨*createBlock*⟩ ⟨*codeLinesOrEmpty*⟩ RBRAC ⟨*else*⟩

⟨*else*⟩ ::= ELSE LBRAC ⟨*createBlock*⟩ ⟨*codeLinesOrEmpty*⟩ RBRAC
  | ε

⟨*for*⟩ ::= FOR ⟨*createBlock*⟩ LPAR ⟨*variable*⟩ ⟨*expr*⟩ SEMICOLON ⟨*increment*⟩ RPAR LBRAC ⟨*codeLinesOrEmpty*⟩ RBRAC

⟨*while*⟩ ::= WHILE ⟨*createBlock*⟩ LPAR ⟨*expr*⟩ RPAR LBRAC ⟨*codeLinesOrEmpty*⟩ RBRAC

⟨*print*⟩ ::= PRINT LPAR ⟨*expr*⟩ RPAR SEMICOLON

⟨*return*⟩ ::= RETURN ⟨*expr*⟩ SEMICOLON

⟨*output*⟩ ::= OUTPUTADD LPAR IDENTIFIER RPAR SEMICOLON

⟨*mesh*⟩ ::= MESHADDFACE LPAR ⟨*expr*⟩ COMMA ⟨*expr*⟩ COMMA ⟨*expr*⟩ RPAR SEMICOLON
  | MESHADDFACE LPAR ⟨*expr*⟩ COMMA ⟨*expr*⟩ COMMA ⟨*expr*⟩ COMMA ⟨*expr*⟩

RPAR SEMICOLON
| MESHADDVERTEX LPAR ⟨*multipleFunctions*⟩ RPAR SEMICOLON

⟨*multipleFunctions*⟩ ::= ⟨*multilevelIdentifier*⟩ COMMA ⟨*multipleFunctions*⟩
| ⟨*multilevelIdentifier*⟩

⟨*expr*⟩ ::= ⟨*multilevelIdentifier*⟩
| ⟨*genericNullary*⟩
| OUTPUT
| OUTPUTEMPTY LPAR RPAR
| ⟨*rotate*⟩
| ⟨*comparison*⟩
| ⟨*expr*⟩ DIVIDE ⟨*expr*⟩
| ⟨*expr*⟩ MULTIPLY ⟨*expr*⟩
| ⟨*expr*⟩ PLUS ⟨*expr*⟩
| ⟨*expr*⟩ MINUS ⟨*expr*⟩
| ⟨*expr*⟩ AND ⟨*expr*⟩
| ⟨*expr*⟩ OR ⟨*expr*⟩
| LPAR ⟨*expr*⟩ RPAR
| ⟨*increment*⟩
| RANDOMFLOAT LPAR FLOAT RPAR
| RANDOMINT LPAR INT RPAR

⟨*rotate*⟩ ::= ROTATE LPAR ⟨*multilevelIdentifier*⟩ COMMA ⟨*multilevelIdentifier*⟩ COMMA ⟨*multilevelIdentifier*⟩ RPAR

⟨*increment*⟩ ::= ⟨*multilevelIdentifier*⟩ PLUSPLUS

⟨*genericNullary*⟩ ::= INT
| FLOAT
| STRING
| BOOL

⟨*multilevelIdentifier*⟩ ::= ⟨*multilevelIdentifierMaker*⟩

⟨*multilevelIdentifierMaker*⟩ ::= IDENTIFIER
| IDENTIFIER DOT ⟨*multilevelIdentifierMaker*⟩

⟨*comparison*⟩ ::= ⟨*comparisonIdentifierOrNullary*⟩ EQUALITY ⟨*comparisonIdentifierOrNullary*⟩
| ⟨*comparisonIdentifierOrNullary*⟩ NEQUALITY ⟨*comparisonIdentifierOrNullary*⟩
| ⟨*comparisonIdentifierOrNullary*⟩ LESS ⟨*comparisonIdentifierOrNullary*⟩
| ⟨*comparisonIdentifierOrNullary*⟩ LESSEQUAL ⟨*comparisonIdentifierOrNullary*⟩
| ⟨*comparisonIdentifierOrNullary*⟩ GREATER ⟨*comparisonIdentifierOrNullary*⟩
| ⟨*comparisonIdentifierOrNullary*⟩ GREATEREQUAL ⟨*comparisonIdentifierOrNullary*⟩

⟨*comparisonIdentifierOrNullary*⟩ ::= ⟨*multilevelIdentifier*⟩
| ⟨*genericNullary*⟩

⟨*initializerList*⟩ ::= LBRAC ⟨*expr*⟩ COMMA ⟨*expr*⟩ COMMA ⟨*expr*⟩ RBRAC

Příloha C

Praktické ukázky

C.1 Vytvoření jednoduché části střechy

V tomto příkladu bude demonstrováno vytváření pravidel a symbolů, přístupy k interním proměnných symbolů a způsob vytváření modelů z bodů.

C.1.1 Vytvoření symbolu reprezentující kvádr

```
symbol Brick {
        //Interni promenne s pocatecnimi hodnotami
        float height = 1.0;
        float width = 1.0;
        float depth = 1.0;
}{
        //Body
        vec3 p0 = {0.0, 0.0, 0.0};
        vec3 p1 = {0.0, 0.0, height};
        vec3 p2 = {width, 0.0, height};
        vec3 p3 = {width, 0.0, 0.0};

        vec3 p4 = {0.0, depth, 0.0};
        vec3 p5 = {0.0, depth, height};
        vec3 p6 = {width, depth, height};
        vec3 p7 = {width, depth, 0.0};

        //Steny
        _mesh.add(p0, p1, p2, p3);
        _mesh.add(p7, p3, p2, p6);
        _mesh.add(p7, p6, p5, p4);
        _mesh.add(p4, p5, p1, p0);
        _mesh.add(p4, p0, p3, p7);
        _mesh.add(p1, p5, p6, p2);
}
```

První blok při deklaraci symbolu je vyhrazen pro deklaraci interních proměnných a symbolů. Druhý je určen pro popsání způsobu, jakým bude symbol interpretován.

Deklarace symbolu Brick, ve kterém jsou uchovány, kromě pozice v souřadném systému a rotace, údaje o výšce, šířce a hloubce. Při interpretaci dojde k vytvoření 8 vektorů, které reprezentují vrcholy kvádru (resp. krychle). Nakonec je z těchto bodů vygenerováno 6 stěn.

C.1.2 Počáteční pravidlo, které vygeneruje 4 sloupy

```
rule initial {
        //Podminka pro aplikaci pravidla
        if (_output.empty()) {
                return true;
        }
        else {
                return false;
        }
}{
        //Akce pri aplikaci pravidla
        Brick brick1;

        Brick brick2;
        brick2._position.x = 5.0;
        brick2._position.y = 0.0;

        Brick brick3;
        brick3._position.x = 5.0;
        brick3._position.y = 10.0;
        brick3.height = 10.0;

        Brick brick4;
        brick4._position.x = 0.0;
        brick4._position.y = 10.0;
        brick4.height = 10.0;

        _output.add(brick1);
        _output.add(brick2);
        _output.add(brick2);
        _output.add(brick3);
        _output.add(brick3);
        _output.add(brick4);
        _output.add(brick4);
        _output.add(brick1);
}
```

Pravidlo INITIAL zajistí, že pokud je splněna podmínka pro aplikaci, tzn. řetězec symbolů je prázdný - zatím nebyly vygenerovány žádné symboly, dojde k aplikaci pravidla. Vygenerují se 4 sloupy a tyto sloupy jsou umístěny do řetězce symbolů, který bude zpracován v následující iteraci. Do řetězce symbolů je potřeba vkládat sloupy vždy dvakrát, protože pravidlo, které mezi těmito sloupy vytvoří zeď ze vstupního řetězce odstraní 2 instance sloupu.

Pokud se jedná o poslední iteraci, je řetězec symbolů interpretován. Výsledek je zobrazen na obrázku C.1.

C.1.3 Symbol reprezentující zeď

```
symbol Wall {
        vec3 startTop;
        vec3 startBottom;

        vec3 endTop;
        vec3 endBottom;
}{
        _mesh.add(startBottom, startTop, endTop, endBottom);
}
```

Symbol WALL, při interpretaci vytvoří plochu, resp. stěnu mezi 4 body.

C.1.4 Pravidlo generující zeď

```
rule wallRule[2] {
        if ($0._type == "Brick" && $1._type == "Brick") {
                return true;
        else{
                return false;
        }
}{
        Wall wall;

        wall.startBottom = $0._position;
        wall.startTop = $0._position;
        wall.startTop.z = wall.startTop.z + $0.height;

        wall.endBottom = $1._position;
        wall.endTop = $1._position;
        wall.endTop.z = wall.endTop.z + $1.height;

        _output.add(wall);
}
```

Pokud se v řetězci symbolů nacházejí 2 symboly typu BRICK vedle sebe, je místo nich do řetězce vložena zeď. Výsledek iterace je na obrázku C.2. V současné verzi nefunguje korektně operátor &&, je tedy potřeba struktury IF vnořovat.

C.1.5 Symbol Roof - střecha

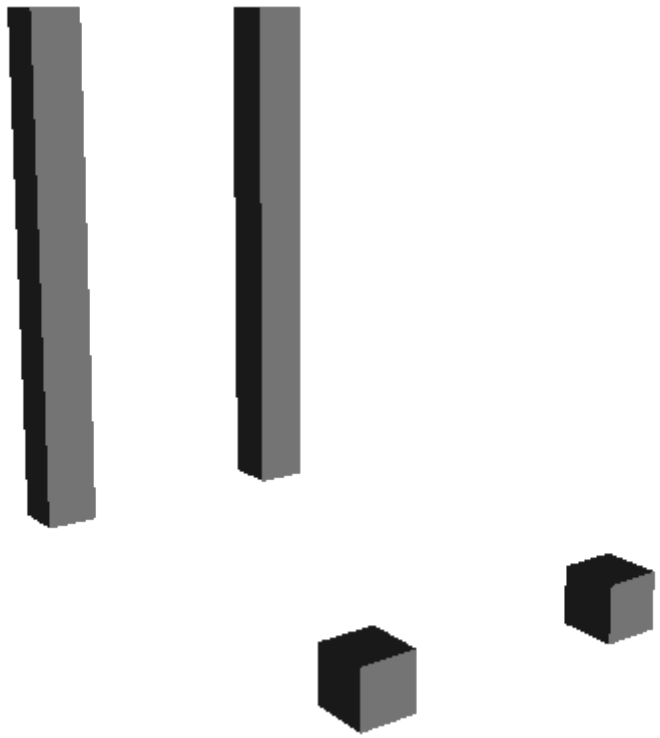
```
symbol Roof {
        vec3 point1;
        vec3 point2;
        vec3 point3;
        vec3 point4;
}{
        _mesh.add(point4, point3, point2, point1);
}
```

Jedná se o jednoduchý symbol, který pouze vytvoří plochu propojením 4 bodů.

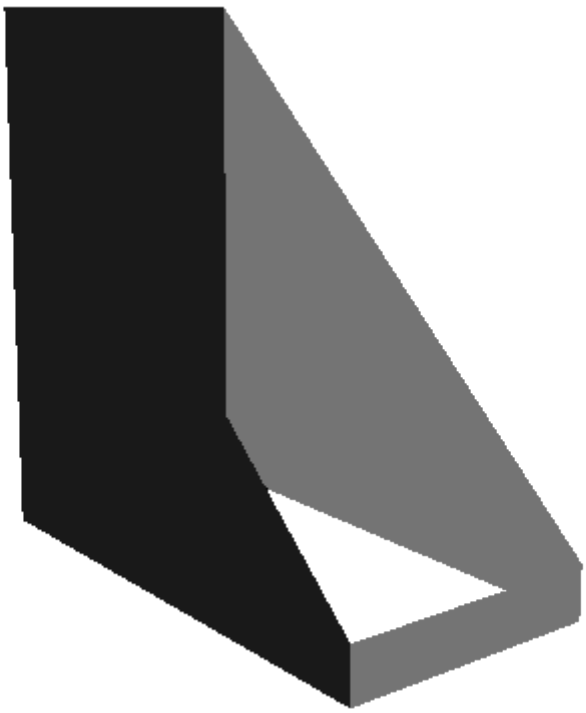
C.1.6 Pravidlo generující střechu

```
rule roof[4] {
        if ($0._type == "Wall" && $1._type == "Wall" &&
        $2._type == "Wall" && $3._type == "Wall") {
                return true;
        else{
                return false;
        }
}{
        Roof roof;

        roof.point0 = $0.startTop;
        roof.point1 = $1.startTop;
        roof.point2 = $2.startTop;
        roof.point3 = $3.startTop;

        _output.add($0);

        _output.add(roof);

        _output.add($1);
        _output.add($2);
        _output.add($3);
}
```

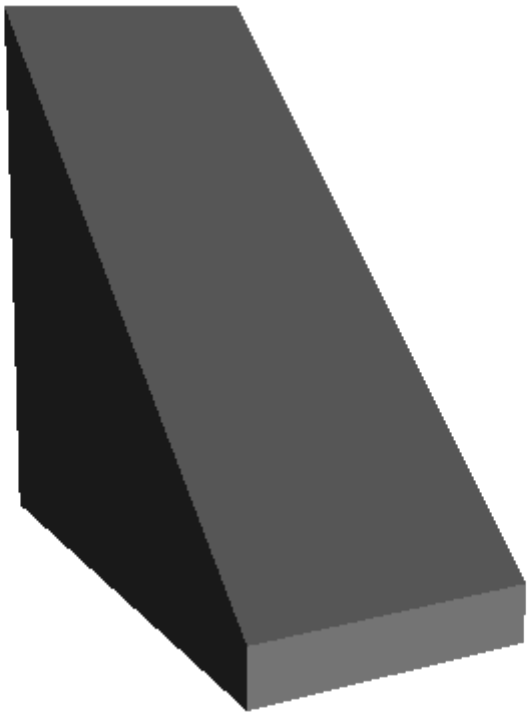
Pravidlo se aplikuje, pokud jsou v řetězci 4 symboly WALL. Při aplikaci je požadováno, aby nedošlo k odebrání symbolů Wall z řetězce a proto je potřeba tyto symboly do řetězce opět vrátit (resp. přidat). Avšak je nutno přerušit sekvenci 4 po sobě následujících symbolů typu Wall, jinak by došlo k zacyklení programu, protože by neustále docházelo k aplikaci pravidla ROOF. Finální výsledek je na obrázku C.3



Obrázek C.1: Výsledek 1. iterace



Obrázek C.2: Výsledek 2. iterace



Obrázek C.3: Výsledek 3. iterace

Příloha D

Obsah CD

Obsah přiloženého CD:

- Zdrojové soubory knihovny
- Zdrojové soubory vizualizační aplikace
- Zdrojové soubory dokumentace
- Příklady
- Video s ukázkou generování panelového domu