UNIVERZITA PALACKÉHO V OLOMOUCI
KATEDRA MATEMATICKÉ INFORMATIKY

ROČNÍKOVÝ PROJEKT

Želví geometrie na krychli

Červen 1999

Aleš Keprt
Informatika, III. ročník

Abstrakt

Kromě ActiveX, které je základním a neotřesitelným zadáním tohoto projektu, je tématem také implementace želví geometrie na krychli. Program nabízí interpret jazyka Logo, který má syntaxi kompatibilní se známou knihou Turtle Geometry. K bohaté množině více než šedesáti základních příkazů pro ovládání želvy, které jsou implementovány podle de facto standardu Comenius Logo, je možno libovolně vytvářet další procedury. Plně podporována je také rekurze, takže program lze s výhodou využít pro modelování čárových fraktálů a dalších složitějších útvarů. Grafický výstup na obrazovku nebo souboru (EPS, BMP) využívá rychlý 3D engine na bázi baricentrických souřadnic a umožňuje interaktivní pohyb po scéně pomocí myši. Jazyk, na rozdíl od ostatních projektů, implementuje všechny obvyklé příkazy želvího Loga, navíc příkazy pro nastavení velikosti krychle a také RGB barvy. K dispozici je také velká sada příkladů - programů v Logu, které demonstrují zajímavé aspekty želví geometrie na ploše a na krychli. Pro vyzkoušení ActiveX serveru je přiložen také testovací ActiveX klient.

Obsah

Seznam obrázků

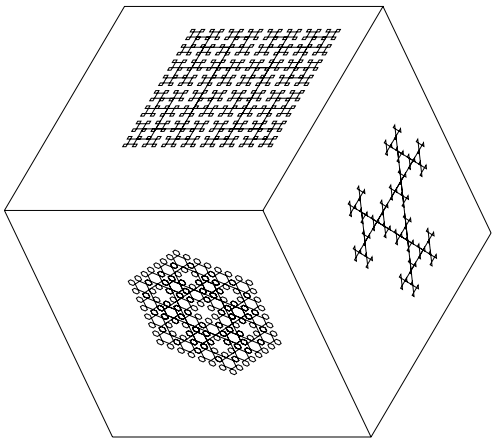
1 Popis zadání a analýza projektu

1.1 Zadání projektu

Vedoucí projektu RNDr.Sklenář mnohokrát zdůraznil, že zadáním projektu je ActiveX. Toto je prvek nový, který v předchozích letech nebyl vidět. Samotná implementace želví geometrie na krychli vychází již více ze starších projektů, avšak především z mých představ, které se poněkud liší od představ mých předchůdců.

Implementoval jsem OLE Automation objekt, který podporuje všechny požadované metody. Jelikož před dokončením projektu jsme nedostali zadání v písemné podobě, implementoval jsem vlastní interface, který je kompatibilní s klientem [5], který je také přiložen.

Smyslem projektu bylo implementovat želví geometrii na krychli. Pro želví geometrii byl požadován jazyk Logo s doporučením programu [2] a knihy [1]. Zvolil jsem tedy syntaxi z [1], která je více „lidská“. Z [1] jsem pak převzal názvy a významy systémových funkcí. Syntaxe je tedy trošku podobná C++, jelikož jazyk má daleko silnější matematický aparát než původní Logo. Pro člověka znalého Logu je práce snadná, jelikož systémové příkazy pro ovládání želvy jsou implementovány všechny a mají standardní význam. Tím se taky tento projekt výrazně liší od projektů z minulých let, které byly až provokativně skoupé na dodržování základních pravidel Loga.



Obrázek 1: Čárové fraktály patří k nejčastějším nasazením Loga

Mezi více než šedesáti systémovými příkazy jsou také příkazy pro nastavení velikosti krychle a podpora RGB barev, včetně předdefinovaných konstant (red, brown, yellow, ...).

Jedním ze základních požadavků také bylo, aby implementovaný jazyk plně podporoval rekurzi. To jsem splnil, takže program lze s výhodou využít pro modelování čárových fraktálů a dalších složitějších útvarů. Grafický výstup na obrazovku nebo souboru (EPS, BMP) využívá rychlý 3D engine na bázi baricentrických souřadnic a umožňuje interaktivní pohyb po scéně pomocí myši. Engine na rychlém počítači stíhá i tisíce čar bez zřetelného zadrhávání. Blikání obrazu jsem zamezil použitím speciálního algoritmu, který je sice triviální, ale když ho člověk nezná, tak má smůlu. Program neobsahuje lištu ikon, což trochu překvapivě přimělo k rychlejšímu vykreslování scény.

K dispozici je také velká sada příkladů - programů v Logu, které demonstrují zajímavé aspekty želví geometrie na ploše a na krychli. Pokud se nestala nepředvídaná událost, několik obrázků z mého programu byste měli vidět vložených na různých místech tohoto

textu.

Pro vyzkoušení ActiveX serveru je přiložen také testovací ActiveX klient [5], jeho autorem je Libor Brodský.

Bližší informace o programu je možno získat v uživatelské dokumentaci (kapitola 2).

1.2 Analýza starších projektů

1.2.1 Jitka Skalická : Želva na krychli, KMI 1996

Jako první jsem se podíval na nejstarší z projektů dostupných na serveru. Ten program mě docela zarazil už při spuštění. Po chvíli zkoušení jsem z toho všeho byl naprosto vedle. Myslím, že horší projekt jsem už viděl jen jeden (editor not).

Po zapnutí programu se objeví divné tmavě šedé maximalizované okno, takže první, co člověka napadne, je zmenšit okno na vhodnou velikost. A ejhle! Programu je velikost okna lhostejná a klidně si kreslí dál. Pro nechápavé: program kreslí mimo okno ve Windows!!! No to je teda něco. A aby to nebylo málo, grafika se podivuhodně překresluje vždy na jiné místo, takže po chvíli máte pět kreseb přes sebe (tzv. želví guláš). V takové situaci člověku ani nevadí, že ta krychle se vlastně krychli vůbec nepodobá. Ano, ona to vlastně ani není krychle. Možná má geometricky vlastnosti krychle, ale jako krychle určitě nevypadá.

Takže, co tu máme: želva stojí v rohu jedné stěny krychle a čeká na příkazy.

Nejprve zadávám **dopredu 10**. Želva popolezla o malý kousek. Asi by to chtělo větší číslo, dávám tedy **dopredu 1000**. Uáááá! Želva vypálila tak rychle kupředu, až z toho opustila krychli a vlezla kamsi do prostoru vedle krychle. Potom se teleportovala na jiné místo mimo krychle a udělala ještě jednu čáru. Ano, to je velice zajímavá verze želvy „na“ krychli. Hned musím podotknout, že funkce na chození želvou mimo krychli jsem v mém programu neimplementoval.

Připomíná mi to úsloví: „Neváhej a toč!“, které, ačkoliv původně názvem vtipného televizního pořadu, nyní coby vtipný počítačový program. A tak neváhám a točím. Při otáčení krychle získávám tři zajímavé poznatky:

1. Želva se skutečně teleportovala kamsi mimo krychli, při otáčení krychle je to dobře vidět.

2. Program dál vesele kreslí mimo okno Windows, jelikož krychle je až v rohu okna a želva při putování mimo krychle občas navštíví menu, lištu nebo programy okolo (commander apod.). Skutečně k popukání!

3. Krychle při otáčení podivně pulzuje, tzn. mění svou velikost v závislosti na úhlu otočení. Vzhledem k tomu, že drogy neberu, obávám se, že ten program skutečně není v pořádku. I když, vzato z druhé strany to je vlastně taková úplně nová projektivní transformace (nebo možná nový superalgoritmus pro násobení matic, kdo ví?).

Na závěr ještě pár slov k dokumentaci. přeskakuji nedůležité kapitoly (jako cituji: „Želví geometrie je program...“) a ...a jsem na konci. Celých pět stran dlouhá dokumentace bakalářského projektu, to je skutečně něco! Nemluvě o skutečně nezvyklém smyslu pro český *pa*jazyk. Nebudu raději citovat.

Toto že někdo obhájil? Obávám se, že na tomto „projektu“ není co analyzovat.

1.2.2 Petr Horáček : Želva na krychli, KMI 1998

Druhý a poslední projekt stejného zadání je napsán ve **Visual Basicu**. Toho jsem si všiml ihned po spuštění programu, když se na mě vyvalila plocha okna úchylně posetá samými

tlačítky (tlačítkový telefon i s faxem). Nechápu, proč mají všichni programátoři ve Visual Basicu stejnou úchylku.

Prostředí připomíná svou složitostí spíše řídící centrum raketoplánu než výukový program („pro děti“ se ani neodvažuji dodávat). Asi tak půl minuty marně zkouším udělat něco - cokoliv - ale marně. Potom mě napadne, že bez myši to asi nepůjde a kliknu zkusmo na první tlačítko. Ha! Chyba „Illegal function call“ a program se ukončil. Ten projekt mi něco velmi připomíná...

Po znovuspuštění zkouším jiný postup. Na tlačítkovém telefonu „vyklikám“ číslo 3-0-0 a potom kliknu na 'krok'. Kupodivu to funguje. îkoda, že krychle je zobrazena z divného úhlu. Snažím se ji tedy otočit, abych taky něco viděl. Dle Murphyho zákonů jsou všechny čáry na odvrácené straně krychle, proto zapínám drátěný model, abych viděl i dozadu a ...áááá... Želva (asi po vzoru z předchozího programu) prchá kamsi hodně daleko mimo krychli. No co to? Ten projekt mi fakt něco připomíná...

Zajímavé prostředí by dokázal pochopit možná MacGyver, ale já nikoliv.

Na závěr čtu dokumentaci. Už zase mi to něco připomíná. Co mi to tak asi připomíná? Dokumentace nemá tentokrát neuvěřitelných 5 stran, ale ještě o celé půl strany víc. Aha, už to mám, vždyť ta dokumentace je stejná, jako ta předchozí. Tedy, abych autora neurazil, nejméně jeden odstavec jsem objevil jako nově přidaný. Teď už vím, odkud to znám. Opět tedy není co analyzovat.

Výsledek analýzy těchto dvou projektů: Teď už věřím, že obhájit se dá skutečně cokoliv.

1.2.3 Lenka Bednaříková : Želva v prostoru, KMI 1996

Po neslavné analýze starších projektů jsem naštěstí našel jeden projekt, který vypadá normálně. Program jsem opět podrobil krátkému testu. V obou předchozích programech jsem chybu objevil již po asi 5 sekundách (prostě po zadání prvního příkazu), zde jsem se s žádnými neduhy nesetkal. Navíc vidím, že to dělala opět nějaká dáma, takže moje předchozí černé vize se naštěstí nepotvrdily.

Dokumentace je, ze zkušenosti už trochu překvapivě, napsána česky bez častých chyb. Skutečně. A má daleko víc než 5 stran. Dokonce víc než deset...

Z tohoto projektu jsem tedy načerpal základní poznatky pro projekt můj.

1.2.4 Miroslav Ambros : Želva v prostoru, KMI 1998

Tento projekt z neznámých důvodů vůbec nebyl na síti ke stažení, takže jsem ho nemohl při analýze projektu pořádně nastudovat. Nyní je již k dispozici, a tak jsem provedl alespoň vzevrubné prozkoumání.

Program je velmi nepřehledný a složitý, možná z toho důvodu, že obsahuje nepoměrně větší množinu funkcí než ostatní projekty. Zajímavá je podpora více želv, ačkoliv celkově mi to připadá spíše nesmyslně složité (opět mám z toho pocit, jakoby to byl raketoplán). Pro výuku dětí naprosto nevhodné. Bohužel taky grafické podání není na úrovni doby, což je velká škoda.

1.3 Teoretická příprava

Teoretickou přípravou na tento projekt bylo především přečtení knihy [1]. Tato kniha velmi podrobně popisuje všechny detaily jak obecné želví geometrie, tak její implementace na krychli. Kniha nepopisuje vlastní jazyk Logo, protože konkrétní podoba jazyka není pro výuku podstatná. Kniha je psána **přesně** ve stylu MIT Press, čili je nesmyslně dlouhá, těžko se čte, základní věci popisuje velmi podrobně, zatímco nejsložitější problémy nechává

nevysvětlené s pobídkou, aby na to přišel čtenář sám. Musím s politováním konstatovat, že zatím všechny americké vědecké knihy, které jsem viděl, byly psány tímto podivným stylem.

Druhou knihou, která mi dala spíše morální podporu, než konkrétní vědomosti, je [4]. Citacím z této knihy jsem věnoval samostatnou kapitolu na konci textu, protože ji považuji za základní kámen pro realizaci (a také obhajobu) jakýchkoliv softwarových projektů.

1.4 Shrnutí analýzy, OpenGL

Za analýzu považuji především důkladné prozkoumání starších projektů. To mi dalo velmi přesnou představu, jak bude můj program vypadat. Na tomto místě bych chtěl zdůraznit pouze grafické podání, které jako bylo odsunuto na vedlejší kolej ve všech dosavadních projektech. Já nechci dělat nějaké nové senzační vynálezy, ale je mým prvořadým cílem, aby to málo, co udělám, bylo stoprocentní. A mám na mysli právě kvalitu, rychlost a interaktivitu grafiky. Především interaktivita v předchozích projektech velmi scházela, ačkoliv je známo, že pouhé otáčení krychle myší **výrazně** podporuje prostorovou představivost. **U takového programu, jehož potenciálními uživateli jsou i děti, je interaktivní podání prostorového světa prvořadým úkolem**.

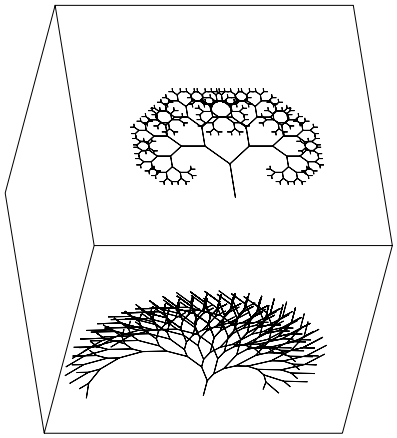
Podle původního plánu měla být grafika realizována pomocí OpenGL. To nakonec padlo, jelikož jsem dal přednost systému baricentrických souřadnic, který by měl být výrazně rychlejší. OpenGL tento koncept bohužel nepodporuje, navíc se mi nepodařilo sehnat driver pro hardwarovou akceleraci OpenGL pro žádný dostupný systém. Čili použití OpenGL by nepřineslo žádné očekávané pozitivní výsledky, ba naopak by mohlo přinést některé neočekávané nepříjemnosti.

2 Uživatelská dokumentace

2.1 Úvod

Program CubeTurtle je interpret jazyka Logo v prostředí povrchu krychle. Program nabízí funkce typické pro běžný interpret Loga, ovšem s důrazem na želví geometrii a její interaktivní provedení na úrovní současné techniky.

Po spuštění programu se objeví hlavní okno, ve kterém se odehrává téměř celý „život“ programu. Tradičními prvky okna je hlavní menu (nahoře), ve kterém jsou všechny potřebné příkazy, které se netýkají vlastního jazyka, a status bar (dole), ve kterém se zobrazuje stručná nápověda k jednotlivým položkám menu apod.



Obrázek 2: Stromeček lbranch(40,23,8) a květák branch(140,8)

2.2 Konzola

Ve spodní části okna je konzola. Je to obdoba klasického příkazového řádku s tím rozdílem, že zde se jedná o obrazovkový editor, čili je to okno textového editoru, kde můžete libovolně jezdit kurzorem a psát kamkoliv text. Od obyčejného textového editoru se tento liší tím, že po stisku klávesy **Enter** se vezme obsah řádku nalevo od kurzoru a vykoná se pomocí interpretu jazyka Loga. Proto se tomuto prvku říká konzola.

Během vykonávání příkazu (lze zadat libovolný platný příkaz) je na konzole vypsán text <**exec**>, který zmizí po dokončení vykonávání příkazu zmizí. Kromě toho se při vykonávání příkazu objeví malé okno s tlačítkem **Abort program** po jehož stisknutí se vykonávání programu násilně přeruší. Při přerušení programu nebo při výskytu chyby se vypíše chybové hlášení informující, k jaké chybě došlo, ve které proceduře a ukáže se přesně místo programu, kde došlo k chybě.

Chcete-li vykonat více příkazů najednou, napište je oddělené středníky na jeden řádek. V konzole nejde zakládat nové procedury, k tomu slouží editor, který je popsán v následující kapitole.

2.3 Editor

Ačkoliv konzola umí vykonat jakýkoliv příkaz, nelze v ní zakládat nové procedury, ani proměnné (to z toho důvodu, že proměnné existují pouze lokálně uvnitř konkrétní procedury).

A právě k zakládání nových procedur slouží editor, který je v pravé části hlavního okna nad konzolou. Editor se chová stejně jako konzola, akorátže při stisku klávesy **Enter** nedojde k vykonání příkazu. Čili je to normální textový editor. Způsob zakládání nových procedur je podrobně popsán v kapitole 3. V editoru i na konzole můžete používat standardní klávesové zkratky pro blokové operace s textem.

2.4 Obrázek krychle

Zbytek okna (nalevo od editoru) je vyhrazen pro obrázek krychle. Po zapnutí CubeTurtle se objeví průhledná krychle v rovnoběžném promítání. Pomocí myši si můžete zobrazení libovolně upravovat. K dispozici máte následující funkce.

- Při stisku a držení tlačítka se pohybem myši krychle otáčí kolem os **x** a **y**.
- Při stisku a držení Ctrl a tlačítka myši se pohybem myši krychle otáčí kolem os **x** a **z**.
- Při stisku a držení Alt a tlačítka myši se horizontálním pohybem myši mění velikost krychle.
- Při stisku a držení Shift a tlačítka myši se pohybem myši posouvá výřez okna. Je tak možno po zvětšení krychle podívat se na její libovolnou část.

Všechna nastavení jsou relativní, čili při změně velikosti okna se změní ve stejném poměru i obraz uvnitř okna. Je pouze třeba dávat pozor na to, že zobrazovací oblast by ideálně měla mít tvar čtverce, v opačném případě se velikost obrazu řídí velikostí menší ze stran obdélníka. Čili jednoduše řečeno je nepraktické nastavovat velikost zobrazovací oblasti moc širokou, anebo naopak moc placatou. V takovém případě je velká část plochy nevyužitá - nic se tam nekreslí. Jak nastavovat velikost jednotlivých částí okna se dočtete v následující kapitole.

2.5 Nastavení okna

Je možno libovolně měnit velikost a tvar okna. Ideální je zřejmě okno maximalizovat na celou obrazovku, ale není to v žádném případě nutné. Nastavení se navíc ukládá do konfiguračního souboru, takže při dalším spuštění programu okno vypadá tak, jak jste si nastavili.

Mezi konzolou, editorem a obrázkem krychle jsou oddělovače (splittery), pomocí kterých můžete libovolně měnit velikost jednotlivých částí okna. Jednoduše kliknete myší např. mezi konzolu a editor (prostě někam mezi jednotlivé části okna), držíte tlačítko a tahem myši měníte velikost jednotlivých částí okna.

2.6 Hlavní menu

Program obsahuje hlavní menu, tak jako každý jiný program pro Windows. V menu jsou k dispozici příkazy, které se netýkají jazyka. Příkazy pro želvu (v jazyce Logo) se zadávají na konzole. Všechny příkazy menu zobrazují nápovědu dole ve stavovém řádku (status bar - nejspodnější část okna, šedý řádek pod konzolou).

2.6.1 File - soubor

New (Ctrl+N) založí nový soubor (v případě potřeby se zeptá, zda nejprve uložit existující soubor)

Open (Ctrl+O) otevře existující soubor (v případě potřeby se zeptá, zda nejprve uložit existující soubor)

Save (Ctrl+S) uloží editovaný soubor, pokud soubor nemá jméno, vykoná se Save As

Save As uloží editovaný soubor pod novým názvem

Export Image (Ctrl+E) exportuje obrázek krychle v aktuální podobě a libovolné velikosti do souboru, na výběr je formát EPS nebo BMP

Exit (Alt+F4) Ukončí aplikaci CubeTurtle

2.6.2 Edit - úpravy

Cut (Ctrl+X, Shift+Delete) vyjme označenou část textu a vloží ji do clipboardu

Copy (Ctrl+C, Ctrl+Delete) označenou část textu vloží do clipboardu

Paste (Ctrl+V, Shift+Insert) vloží text z clipboardu na místo kurzoru

Find (Ctrl+F) zobrazí dialog hledání řetězce

Find Next (F3) najde další výskyt řetězce od místa kurzoru

Copy Image (Ctrl+I) okopíruje obrázek krychle v aktuální podobě do clipboardu (pro použití v jiných programech)

2.6.3 Libraries - knihovny

Load Library (Ctrl+L) načte a zaregistruje novou knihovnu funkcí želvího Loga

Libraries (Alt+L) zobrazí seznam knihoven a jejich procedur (viz níže)

2.6.4 Options - nastavení

Console Font zobrazí dialog, kde si můžete libovolně změnit font písma konzoly

Editor Font zobrazí dialog, kde si můžete libovolně změnit font písma editoru

Transparent Cube (Ctrl+T) přepínač průhlednosti krychle (u průhledné krychle jsou zadní stěny vidět čárkovaně, u neprůhledné nejsou vidět vůbec)

Show Faces (Ctrl+W) přepínač prostorového a plošného zobrazení krychle (plošné zobrazení ukazuje krychli rozloženou na 6 stěn)

Visible Turtle (Ctrl+R) přepínač viditelnosti želvy, dá se také ovlivnit příkazy showturtle a hideturlte Loga (je to jediný příkaz menu, který přímo koresponduje s Logem)

Paint Never (Ctrl+1) určuje, že během provádění příkazů se nebude překreslovat krychle

Paint Sometimes (Ctrl+2) určuje, že během provádění příkazů se bude krychle překreslovat po každých 10 nových linkách

Paint Everytime (Ctrl+3) určuje, že během provádění příkazů se bude krychle překreslovat po každé nové lince

2.6.5 Help - nápověda

Contents (F1) zobrazí obsah nápovědy (elektronická obdoba této dokumentace)

Show Hints (Shift+F1) přepínač zobrazování okamžité nápovědy (v malých žlutých obdélnících)

About zobrazí informace o aplikaci CubeTurtle a jejím autorovi

2.7 Seznam knihoven a jejich procedur

Stiskem Alt+F lze zobrazit seznam knihoven a jejich procedur. Je to okno se dvěma listboxy, kde levý ukazuje abecední seznam všech knihoven, které jsou v paměti, a pravý ukazuje vždy procedury ve zvolené knihovně. U každé procedury je také vidět seznam parametrů. Poklepáním na knihovnu nebo proceduru lze zobrazit její kód v editoru. Procedury z knihoven se zobrazují ve speciálním okně a pochopitelně je nelze editovat.

Jelikož je tento dialog snadno přístupný přes klávesovou zkratku Alt+L, lze ho s výhodou použít při vyhledávání konkrétní procedury v delším textu.

Okno, ve kterém se zobrazí kód knihovny je nemodální, čili může zůstat otevřeno i při další práci v CubeTurtle.

2.8 Export obrázků

CubeTurtle umožňuje export obrázků dvěma způsoby: do clipboardu nebo do souboru. Při vložení obrázku do clipboardu získáte vlastně přesnou kopii obrázku, jak jej vidíte na obrazovce. Při exportu do souboru máte možnost vybrat si mezi formáty **Encapsulated PostScript (EPS)** a **Windows Bitmap (BMP)** a navíc si můžete nastavit velikost exportovaného obrázku.

Ať už obrázek vyexportujete jakýmkoliv způsobem, vždy dostanete 100

2.9 ActiveX

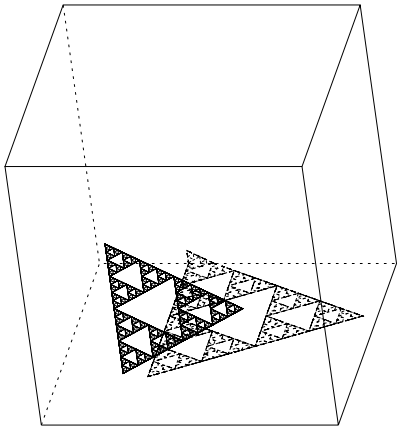
CubeTurtle podporuje také komunikaci přes ActiveX rozhraní. Pro běžného uživatele není tato věc důležitá a je podrobně popsána v programátorské dokumentaci. Pro vyzkoušení ActiveX rozhraní je přibalen testovací program **OLETest** (viz [5]). CubeTurtle přitom podporuje outproc i inproc server a funguje dobře i přes dispatch rozhraní (které ostatně používá i OLETest).

2.10 Jazyk Logo

Želví Logo implementované v CubeTurtle se od ostatních liší především svým založením na fulltextovém editoru a silnou podporou matematických výrazů.

Fulltextové založení znamená, že všechny uživatelské procedury se píší velmi pohodlně do jednoho textového souboru v jednom editačním okně. Interpret pak pročítá tento soubor při každé změně a zapamatuje si všechno, co ho zajímá. Systém je navržen tak, aby byl dostatečně rychlý. Na soudobých počítačích se čas spotřebovaný na automatické procházení textem pohybuje někde v řádu milisekund, takže to nijak neznepříjemňuje práci. Moje implementace Loga tedy více využívá potenciálu soudobé techniky a je daleko příjemnější pro uživatele, nehledě na to, že více připomíná práci v běžných programovacích jazycích (Basic, Pascal, C++).

Matematická orientace mého Loga se projevuje v tom, že téměř všechny příkazy se vyhodnocují na číselné hodnoty a mohou být součástí složitějších příkazů. Tato vlastnost



Obrázek 3: Trojúhelník vrhá stín

je dána konstrukcí interpretu. Podrobný popis je uveden v kapitole 3, zde jeden příklad za všechny: **fd 34+local a=3*sin fd rt 90**
Tento příkaz nejprve otočí želvu o 90 stupňů doprava, potom popojde o hodnotu, kterou vrátí procedura **rt**, následně vypočítá sinus návratové hodnoty z **fd** a trojnásobek této hodnoty uloží do nově založené proměnné a. Nakonec se ještě posune o dalších 34+a jednotek dopředu.

2.11 Rozdíly od Comenius Loga

Programátoři znalí Loga zřejmě považují za standard implementaci **Comenius Logo** [2]. Proto tato kapitola pojednává o rozdílech mé implementace oproti Comenius Loga.

2.11.1 Zakládání procedur

Procedury se v CubeTurtle zapisují do editoru. Deklarují se stejně jako v Comenius Logu, akorátže se parametry oddělují čárkami. K ukončení procedury je klasický příkaz **end**, ten ale v CubeTurtle má jen formální význam, v podstatě jako příkaz **stop** s kontrolou syntaxe (nelze zadat end uvnitř bloku if nebo repeat).

2.11.2 Zakládání proměnných

Proměnné jsou v CubeTurtle pouze lokální. Základními proměnnými jsou vstupní parametry procedury, další se zadávají příkazem **local**, např. **local a** nebo **local a=40**. Zde je základní odlišnost od Comenius Loga, které používá velmi zvláštní syntaxi (tzv. standardní Logo). Naopak CubeTurtle má v podstatě syntaxi Basicu. Proměnné lze také zakládat skupinově, v tom případě může být inicializována jen poslední proměnná v řadě, např. **local a,b,c=1**. Neinicializované proměnné mají nulovou hodnotu.

2.11.3 Oddělování příkazů

Na rozdíl od Comenius Loga, CubeTurtle je platným ukončením příkazu pouze konec bloku, středník nebo nový řádek. Tzn. např. příkaz **fd 10 rt 90** je chybný, správně musí být uprostřed středník.

2.11.4 Množina systémových příkazů

CubeTurtle podporuje jen příkazy pro ovládání želvy, další obecné příkazy Loga nejsou podporovány. Navíc jsou k dispozici některé nové příkazy, ať už matematické nebo jiné. Pro přesnější informace je doporučeno nahlédnout do seznamu knihoven (Alt+L) a neznámé procedury vyhledat v referenčním manuálu.

2.11.5 Barvy

Zatímco Comenius Logo je nějak nepochopitelně omezeno jen na 16 barev, CubeTurtle nabízí milion barev včetně 21 základních barevných konstant. Základní barvy nereprezentují obvyklé RGB rozložení, nýbrž barvy „člověku milé“ (orange, brown, pink apod.). Konstantu **light** lze přičíst pro získání světlejšího odstínu. To ale nefunguje u několika nejsvětlejších barev (yellow, white apod.).

RGB barvy se zadávají v procentním formátu, např. číslo 235099 znamená: červená 23%, modrá 50%, zelená 99%. Hodnota 000000 je černá barva, naopak 999999 je nejsvětlejší bílá.

3 Referenční manuál želvího Loga

3.1 Základní konstrukty jazyka

3.1.1 Zapisování příkazů

Příkazy se zapisují na samostatné řádky, anebo se oddělují středníkem. Přebytečné středníky mezi příkazy se ignorují. Některé konstrukty používají bloky příkazů uzavřené v hranatých závorkách, které rovněž fungují jako oddělovače příkazů.

V následujícím textu jsou do <lomených závorek> uzavřeny povinné položky a do [hranatých závorek] položky nepovinné.

3.1.2 Definice procedury - to

V CubeTurtle se definují pouze začátky procedur. Syntaxe je následující:

```
to název <parametry>
```

Parametry se zapisují jako seznam jmen oddělených čárkami. Parametry jsou nepovinné, jejich počet je libovolný.

3.1.3 Konec procedury - end

Ukončení procedury má jen formální význam. Rozdíl mezi příkazem **stop** a **end** je v tom, že stop umí ukončit proceduru z libovolného místa (vnořené if apod.), zatímco end nesmí být vnořen do žádných bloků.

3.1.4 Zakládání proměnných - local

Proměnné jsou vždy lokální, číselné a zakládají následujícím způsobem:

```
local [seznam] <název> [ = hodnota]
```

Při nezadání hodnoty se proměnná založí s nulovou hodnotou. Seznam může obsahovat libovolný počet dalších proměnných oddělených čárkami, které se také založí vždy z nulovou hodnotou.

3.1.5 Podmíněný příkaz - if

Podmíněný příkaz má vždy stejný tvar:

```
if <výraz> <[>blok1<]> <[> blok2<]>
```

Nejprve se vyhodnotí výraz za klíčovým slovem if. Pokud je jeho hodnota nenulová, vykoná se blok1 (uzavřený v hranatých závorkách), v opačném případě se vykoná blok2 (rovněž v hranatých závorkách). Syntaxe bloku, který se nevykoná, se nekontroluje. Uvnitř bloku mohou být libovolné příkazy.

3.1.6 Cyklus (opakování) - repeat

Cyklus má jen jeden možný tvar.

```
repeat <výraz> <[>blok<]>
```

Před prvním průchodem se vyhodnotí výraz a podle jeho hodnoty se vykoná příkaz n-krát, kde n je největší celé číslo menší nebo rovno hodnotě výrazu.

3.1.7 Volání procedury

Procedura se volá udáním jména a seznamem parametrů. Parametry jsou v závorkách (všechny dohromady) a jsou odděleny čárkami. Jednoparametrové procedury je možno volat i bez závorky, jelikož fungují jako unární operátory. Je však třeba si dát velký pozor na příkazy typu **fd 10+2**, jelikož unární operátory mají vyšší prioritu než binární, čili v uvedeném příkladu se vykoná nejprve **fd 10** a až potom se k tomu přičte 2. Navíc se neobjeví žádné chybové hlášení, jelikož příkaz může být matematickým výrazem. A to i v tomto případě, kdy příkaz **10+2** nemá žádný efekt.

3.1.8 Návratová hodnota procedury - output

Pokud není explicitně uvedeno jinak, jednoparametrové příkazy vracejí hodnotu vstupního parametru (např. **forward**). Příkazy bez parametru žádnou hodnotu nevracejí (např. **penup**). Toto se týká pouze systémových procedur.

Uživatelské procedury vracejí standardně nulu, příkazem **output** <**hodnota**> lze vracet libovolné číslo. Tyto procedury tedy vždy nějaké číslo vracejí.

3.1.9 Logické operace

V logických operacích se CubeTurtle chová stejně jako Céčko, čili jakákoliv nenulová hodnota je **true** a nula je **false**. Systémové funkce vždy vracejí buď nulu nebo jedničku (myšleno funkce provádějící logické operace).

3.2 Číselné operace

3.2.1 Infixové operátory

CubeTurtle podporuje standardní infixové operátory Loga, navíc několik dalších je odkoukáno z Céčka.

\+ součet

- rozdíl

***** součin

/ podíl

= přiřazení

% zbytek po celočíselném dělení

(šipka nahoru) reálná mocnina

(1 nebo 2 svislé čára) or (logický součet)

& nebo && and (logický součin)

! negace (logický zápor)

< menší (vyhodnocuje se na false=0 nebo true=1)

<= menší nebo rovno (vyhodnocuje se na false=0 nebo true=1)

> větší (vyhodnocuje se na false=0 nebo true=1)

>= větší nebo rovno (vyhodnocuje se na false=0 nebo true=1)

== porovnání dvou čísel na rovnost (vyhodnocuje se na false=0 nebo true=1)

! = **nebo** <> nerovná se (vyhodnocuje se na false=0 nebo true=1)

3.2.2 abs <číslo>

Vrací absolutní hodnotu parametru, čili vzdálenost parametru od nuly na reálné číselné ose.

3.2.3 and <číslo1>,<číslo2>

Vrací logický součin parametrů.

3.2.4 arctan <číslo>

Vrací arkus tangens parametru, úhel je v radiánech.

3.2.5 cos <úhel>

Vrací kosinus parametru, úhel je v radiánech.

3.2.6 difference <číslo1>,<číslo2>

Vrací rozdíl číslo1-číslo2.

3.2.7 div <číslo1>,<číslo2>

Vrací celočíselný podíl číslo1/číslo2. Umí dělit libovolná reálná čísla (kromě nuly), vrací vždy celé číslo.

3.2.8 equal? <číslo1>,<číslo2>

Test na rovnost dvou čísel, vrací false=0 nebo true=1.

3.2.9 exp <číslo>

Vrací přirozený exponent parametru.

3.2.10 false

Konstanta false=0.

3.2.11 greater? <číslo1>,<číslo2>

Test, zda je první číslo větší než druhé, vrací false=0 nebo true=1.

3.2.12 int <číslo>

Vrací celočíselnou část čísla (ořezání na číslo bližší k nule).

3.2.13 less? <číslo1>,<číslo2>

Test, zda je první číslo menší než druhé, vrací false=0 nebo true=1.

3.2.14 ln <kladné číslo>

Vrací přirozený logaritmus parametru.

3.2.15 mod <číslo1>,<číslo2>

Vrací zbytek po celočíselném dělení číslo1/číslo2. Umí dělit libovolná reálná čísla (kromě nuly), vrací vždy celé číslo. Tato procedura je totožná s procedurou **remainder**.

3.2.16 not <číslo1>,<číslo2>

Vrací logický součin parametrů..

3.2.17 or <číslo1>,<číslo2>

Vrací logický součin parametrů..

3.2.18 product <číslo1>,<číslo2>

Vrací součin dvou čísel.

3.2.19 quotient <číslo1>,<číslo2>

Vrací podíl číslo1/číslo2.

3.2.20 random <kladné číslo>

Vrací náhodné celé číslo z intervalu <0;parametr-1>.

3.2.21 randomize

Nastaví semínko vyhledávacího algoritmu na náhodnou hodnotu, podle hodin reálného času. Nevrací hodnotu.

3.2.22 round <číslo>

Vrací parametr zaokrouhlený na celá čísla (nejbližší celé číslo k parametru).

3.2.23 sin <úhel>

Vrací sinus parametru, úhel je v radiánech.

3.2.24 sqrt <nezáporné číslo>

Vrací druhou odmocninu parametru.

3.2.25 remainder <číslo1>,<číslo2>

Vrací zbytek po celočíselném dělení číslo1/číslo2. Umí dělit libovolná reálná čísla (kromě nuly), vrací vždy celé číslo. Tato procedura je totožná s procedurou **mod**.

3.2.26 sum <číslo1>,<číslo2>

Vrací součet dvou čísel.

3.2.27 tan <úhel>

Vrací tangens parametru, úhel je v radiánech.

3.2.28 true

Konstanta true=1.

3.3 Želví grafika

3.3.1 back <vzdálenost>

Posune želvu dozadu o danou vzdálenost. Totožné s procedurou **bk**.

3.3.2 bk <vzdálenost>

Posune želvu dozadu o danou vzdálenost. Totožné s procedurou **back**.

3.3.3 clean

Smaže všechny čáry. Ostatní věci nemění. Nevrací hodnotu.

3.3.4 clearscreen

Smaže všechny čáry a vrátí želvu do původní pozice a směru. Ostatní nastavení nemění. Nevrací hodnotu.

3.3.5 draw

Smaže všechny čáry a resetuje želvu (vrátí ji do stavu při spuštění CubeTurtle). Nevrací hodnotu.

3.3.6 fd <vzdálenost>

Posune želvu dopředu o danou vzdálenost. Totožné s procedurou **forward**.

3.3.7 forward <vzdálenost>

Posune želvu dopředu o danou vzdálenost. Totožné s procedurou **fd**.

3.3.8 heading

Vrací úhel natočení želvy, hodnota je v intervalu <0;360).

3.3.9 hideturtle

Schová želvu. Želva není vidět, ale jinak to nemá na nic vliv. Nevrací hodnotu. Totožné s procedurou **ht**.

3.3.10 home

Přesune želvu do výchozího místa, ale nemění ostatní věci. Nevrací hodnotu.

3.3.11 ht

Schová želvu. Želva není vidět, ale jinak to nemá na nic vliv. Nevrací hodnotu. Totožné s procedurou **hideturtle**.

3.3.12 left <úhel>

Otočí želvu doleva, hodnota je ve stupních. Totožné s procedurou **lt**.

3.3.13 lt <úhel>

Otočí želvu doleva, hodnota je ve stupních. Totožné s procedurou **left**.

3.3.14 pd

Položí pero, želva začne kreslit kudy chodí. Nevrací hodnotu. Totožné s procedurou **pendown**.

3.3.15 pc

Vrací barvu pera. Totožné s procedurou **pencolor**.

3.3.16 pencolor

Vrací barvu pera. Totožné s procedurou **pc**.

3.3.17 pendown

Položí pero, želva začne kreslit kudy chodí. Nevrací hodnotu. Totožné s procedurou **pd**.

3.3.18 penup

Zvedne pero, želva přestane kreslit kudy chodí. Nevrací hodnotu. Totožné s procedurou **pu**.

3.3.19 pu

Zvedne pero, želva přestane kreslit kudy chodí. Nevrací hodnotu. Totožné s procedurou **penup**.

3.3.20 right <úhel>

Otočí želvu doprava, hodnota je ve stupních. Totožné s procedurou **rt**.

3.3.21 rt <úhel>

Otočí želvu doprava, hodnota je ve stupních. Totožné s procedurou **right**.

3.3.22 setpc <číslo>

Nastaví barvu čáry. Totožné s procedurou **setpencolor**.

3.3.23 setpencolor <číslo>

Nastaví barvu čáry. Totožné s procedurou **setpc**.

3.3.24 seth <číslo>

Nastaví úhel natočení želvy, hodnota je v intervalu <0;360). Totožné s procedurou **setheading**.

3.3.25 setheading <číslo>

Nastaví úhel natočení želvy, hodnota je v intervalu <0;360). Totožné s procedurou **seth**.

3.3.26 setx <číslo>

Nastaví x-ovou souřadnici želvy. Souřadnice se vztahuje vždy k jedné stěně.

3.3.27 sety <číslo>

Nastaví y-ovou souřadnici želvy. Souřadnice se vztahuje vždy k jedné stěně.

3.3.28 shown?

Dotaz, zda je želva vidět. Vrací false=0 nebo true=1.

3.3.29 showturtle

Ukáže želvu. Želva je vidět, ale jinak to nemá na nic vliv. Nevrací hodnotu. Totožné s procedurou **st**.

3.3.30 st

Ukáže želvu. Želva je vidět, ale jinak to nemá na nic vliv. Nevrací hodnotu. Totožné s procedurou **showturtle**.

3.3.31 xcor

Vrací aktuální x-ovou souřadnici želvy. Souřadnice se vztahuje vždy k jedné stěně.

3.3.32 ycor

Vrací aktuální y-ovou souřadnici želvy. Souřadnice se vztahuje vždy k jedné stěně.

3.3.33 towards <x>,<y>

Vrací úhel na který by se musela želva otočit, aby se dívala do bodu [x,y]. Pro natočení želvy je třeba použít **setheading**.

3.4 Práce s krychlí

Tato sekce popisuje příkazy želví grafiky, které se vztahují ke krychli.

3.4.1 cubesize

Vrací aktuální velikost hrany krychle, standardně je to 1000.

3.4.2 face

Vrací číslo stěny krychle, na které stojí želva. Stěny jsou očíslovány 0 až 5.

3.4.3 sethome <celé číslo>

Přesune želvu do středu zadané stěny. Stěny jsou číslovány 0 až 5.

3.4.4 setcubesize <kladné číslo>

Nastaví velikost hrany krychle. Poloha želvy ani existující čáry na krychli se nezmění.

3.4.5 setface <celé číslo>

Přesune želvu na danou stěnu, ale nemění žádné další nastavení. Stěny jsou číslovány 0 až 5.

3.5 Barvy

3.5.1 Procentní RGB systém

CubeTurtle nabízí milión barev včetně 21 základních barevných konstant. Základní barvy nereprezentují obvyklé RGB rozložení, nýbrž barvy „člověku milé“ (orange, brown, pink apod.).

RGB barvy se zadávají v procentním formátu, např. číslo 235099 znamená: červená 23%, modrá 50

3.5.2 Předdefinované konstanty

Yellow	757500	Black	000000
Cyan	007575	Grey	505050
Blue	000075	Gray	505050
Violet	750075	Orange	884400
Red	750000	White	999999
Brown	381909	Pink	996375
Green	007500	Light	242424

Konstantu **light** lze přičíst pro získání světlejšího odstínu. To ale nefunguje u tří nejsvětlejších barev (orange, white, pink). Základní barvy můžete také libovolně míchat, jelikož se jedná o RGB, např. součtem **red+blue** vznikne fialová barva.

Konstanty **gray** a **grey** mají stejnou hodnotu (americká resp. Britská angličtina). Celkem je tedy k dispozici 21 předdefinovaných barevných konstant.

3.6 Obecné příkazy

3.6.1 end

Ukončí provádění procedury. Má stejný význam jako **output 0**, ale nesmí být uvnitř žádného bloku [ ].

3.6.2 case <celé číslo>

Nastaví case sensitivitu interpretu. Je-li parametr **true**, interpret bude rozlišovat malá a velká písmena. Je-li parametr **false**, interpret velká a malá písmena nerozlišuje. Standardní hodnota je **false**. Ukončí provádění procedury. Má stejný význam jako **output 0**, ale nesmí být uvnitř

3.6.3 stop

Ukončí provádění procedury. Má stejný význam jako **output 0**.

3.6.4 output <číslo>

Ukončí provádění procedury a vrací dané číslo jako návratovou hodnotu procedury.

3.7 Priority operátorů

Následující tabulka ukazuje priority operátorů.

( )	16
unární operátory, procedury s 1 parametrem	11
!	10
mocnina	9
/ %	8
+ -	7
< <= > >= == ! = <>	5
and, or	4
=	2
,	1

Unární operátory, funkce s jedním parametrem a přiřazování se asociují zprava, ostatní operátory se asociují zleva. Závorky jsou ternárním operátorem.

4 Interpret jazyka Logo

V této následujících kapitolách bude popsána struktura programu (tzv. programátorská dokumentace). Z důvodu poměrné rozsáhlosti programu jsem popis rozdělil do několika samostatných částí. Přitom se zaměřím na popis struktury a metod jednotlivých tříd, zatímco celkové chování systému je popsáno v uživatelské dokumentaci. V této kapitole bude tedy popsána nejrozsáhlejší a nejdůležitější část projektu, interpret jazyka Logo.

4.1 Deklarační makra

Pro ulehčení života programátora jsem hojně používal následující makra, která zajistí deklaraci privátní proměnné a veřejných přístupových metod nebo properties.

Každé makro má dva parametry: **typ proměnné** a **název proměnné**.

makro	popis
fvar	proměnná + metody **get** a **set** bez těla
vvar	proměnná + metody **get** a **set** které přímo čtou/zapisují proměnnou
rvar	proměnná + metoda **get** která přímo vrací její hodnotu
vprop	proměnná + property, která přímo zpřístupňuje proměnnou
rprop	proměnná + read-only property, která přímo zpřístupňuje proměnnou

4.2 enum ParserError

Interpret při výskytu chyby vrací konstantu typu **ParserError**. Seznam možných hodnot tohoto výčtového typu je v následující tabulce.

konstanta	význam
PE_None=0	bez chyby
PE_IdentExpected	očekáván identifikátor
PE_InvalidCharacter	neplatný znak
PE_Syntax	chybná syntaxe
PE_UnexpectedEnd	neočekávaný konec
PE_UnexpectedOperator	neočekávaný operátor
PE_MissingBracket	chybějící pravá závorka
PE_StackOverflow	zásobník přetekl
PE_OperatorExpected	očekáván operátor
PE_DivideByZero	dělení nulou
PE_VarExists	proměnná již existuje
PE_UnmatchedEnd	přebývající 'end'
PE_UnknownIdent	neznámé jméno
PE_BlockExpected	očekáván blok [ ]
PE_UnmatchedEndBlock	přebývající]
PE_InvalidPower	neplatná (záporná) mocnina
PE_Unsupported	nepodporovaná vlastnost
PE_BracketExpected	očekávána závorka (
PE_StopRequest	požadováno násilné přerušení interpretu

4.3 Třída Interpret - interpret želvího Loga

Třída Interpret je jakousi špičkou ledovce. Je to vstupní bod do interpretu želvího Loga. Metody této třídy provádějí za pomoci ostatních metod tohoto balíku vykonávání kódu. Jazyk je interpretován pomocí automatu, který je realizován systémem příkazů switch-case a if-else. Nejsem si jistý, zda je interpret dobře napsán, jelikož interpretaci či překlad

programovacího jazyka alespoň zatím nebyl součaástí výuky ve škole. Proto jsem musel hodně improvizovat.

4.3.1 Metody execute...

Veřejná metoda **execute** slouží jako vstupní bod do interpretu.

```
typedef void (*FarProc)();
static ParserError execute(const char *text,int &index,float &result,
  FarProc callback=0);
```

parametr	význam
text	pointer na program v Logu, který se má vykonat
index	vrací se v něm číslo znaku, kde došlo k chybě
result	vrací se v něm návratová hodnota
callback	nepovinný pointer na funkci, která se volá po vykonání každého příkazu

Tato veřejná metoda inicializuje potřebné věci a volá privátní metodu **execute**. Lokální execute se volá i při volání uživatelských procedur z jiných procedur.

```
static ParserError execute(float &result);
```

Tato metoda pouze zavolá execute2 a následně najde místo textu, kde došlo k chybě.

```
static ParserError execute2(float &result);
```

Teprve metoda execute2 provádí vlastní vykonání kódu.

4.3.2 Metody call

Metody **call** slouží pro volání procedur. První varianta se volá pro provedení procedury bez parametru nebo s jedním parametrem (je to rychlejší).

```
static ParserError call(Proc *proc,float &result,int params);
```

parametr	význam
proc	pointer na volanou proceduru
result	hodnota vstupního parametru a současně návratová hodnota procedury
params	počet vstupních parametrů (vždy 0 nebo 1)

Pro vkastní vykonání se použije druhá verze metody **call**, která již akceptuje všechny existující procedury, včetně systémových.

```
static ParserError call(CalledProc *called,float &result);
```

parametr	význam
called	pointer na volanou proceduru
result	hodnota vstupního parametru a současně návratová hodnota procedury

Jak je vidět, základní rozdíl je v prvním parametru. Zatímco objekt **Proc** existuje pro každou proceduru právě jeden a předává se pouze pointer na něj, objekt **Called** se vytváří nový při každém volání procedury. Třída Called má schopnost střádat a přenášet vstupní parametry, zatímco třída Proc má uchovány pouze jejich názvy.

Systémové procedury se vykonají přímo v této metodě, pro vykonání uživatelských se volá **execute** (viz výše).

4.3.3 Metoda init

```
static void init(); //inicializuje interpret
```

Tato metoda se volá při spuštění aplikace a slouží k jeho inicializaci. Jelikož objekt intepretu je statický, tato metoda může být v něm. V opačném případě by tato metoda byla globání funkcí. Metoda nemá žádné parametry ani nevrací žádnou hodnotu.

4.3.4 Metoda regprocs

```
static const char* regprocs(const char*,int libid=LIB_User);
```

Toto je velmi důležitá metoda. Volá se velmi často z venku a slouží pro registraci uživatelských procedur po té, co uživatel změnil text programu v editačním okně. Metoda pomocí lexikálního analyzátoru prochází text a hledá klíčová slova **to**. Pokud je zápis syntakticky správný, registrují se všechny nalezené procedury.

4.3.5 Metoda geterror

Tato metoda slouží pro zjištění přesných informací o chybě, která nastala při vykonávání programu.

```
void geterror(const char *&text,int &size,int &pos);
```

parametr	význam (všechny parametry jsou výstupní)
text	pointer na text řádku s chybou
size	délka řádku s chybou
pos	pozice chyby na řádku (číslo znaku)

Hodnoty vracené touto metodou se počítají v metodě **execute** ihned po návratu z metody **execute2**.

4.3.6 Další metody

```
static void setcase(int s); //nastaví case sensitivitu interpretu
static const char* errname(ParserError); //vrací jméno chyby podle čísla
Proc* getlastcalled(); //vrací pointer na posledně volanou proceduru
void abort(); //vynutí si násilné ukončení interpretu
```

4.4 Seznam operátorů

Operátory jsou definovány v lexikálním analyzátoru (subor **semantic.cpp**).

konstanta	řetězec	význam
OP_Note	//	komentář
OP_And	&&	and
OP_Or	||	or
OP_NotEqual	<>	nerovná se
OP_NotEqual	!=	nerovná se
OP_LoEqual	<=	menší nebo rovno
OP_HiEqual	>=	větší nebo rovno
OP_Equal	==	rovno
OP_Plus	+	součet
OP_Minus	–	rozdíl
OP_Mul	*	součin
OP_Div	/	podíl
OP_Mod	%	zbytek po celočíselném dělení
OP_Pow	↑	mocnina
OP_And	&	and
OP_Or	|	or
OP_LtBracket	(	levá závorka v matematických výrazech
OP_RtBracket	)	pravá závorka v matematických výrazech
OP_Let	=	přiřazení
OP_Not	!	negace
OP_Lower	<	menší
OP_Higher	>	větší
OP_Comma	,	čárka
OP_BegList	[	začátek bloku příkazů
OP_EndList	]	konec bloku příkazů

4.5 Tempalte stack - šablona zásobníku

4.5.1 Popis šablony

Intepret želvího Loga používá skutečně hodně dynamických datových struktur. Většina z nich je postavena na této šabloně, která reprezentuje klasický zásobník. Na zásobník se ukládají přímo objekty, tedy nikoli poitnery na obejty. Je to tak rychlejší a lépe se s tím pracuje. Vyžaduje to pouze, aby buď každá třída dědila společného „zásobníkového“ předka nebo aspoň deklarovala členskou proměnnou **prev**, která ukazuje na předchozí prvek na zásobníku. Je rovněž potřeba v konstruktoru tuto proměnnou vynulovat.

Odměnou je nám rychlá a elegantní implementace zásobníku, jak na třídy, tak na datové struktury.

4.5.2 Metody šablony

```
Stack() {top=0;items=0;}
virtual ~Stack() {empty();}

int getitems()const {return items;} //vrací počet prvků na zásobníku
Item* gettop()const {return top;}   //vrací prvek z vrcholu zásobníku

void push(Item *item); //přidá prvek na vrchol zásobníku
void pop(); //odstraní prvek z vrcholu zásobníku
void empty(); //vyprázdní zásobník
```

4.5.3 Procházení zásobníkem

Procházení zásobníkem je velmi snadné. Nejprve voláním metody **gettop()** získáme pointer na vrchol zásobníku a potom vždy proměnná **prev** obsahuje pointer na předchozí prvek. U tříd je samozřejmě nutné realizovat přístupovou metodu pro čtení **prev**. Ta se ve všech mých třídách jmenuje **getprev()**.

4.6 Tempalte queue - šablona fronty

4.6.1 Popis šablony

Druhou a poslední použitým dynamickým abstraktním datovým typem je fornta **queue**. Je velice podobná zásobníku **stack**. Je to v podstatě totéž, pouze se jedná o FIFO sytém namísto tamního LIFO.

Fronta je použita jen na jendom místě programu - pro pamatování čar na krychli. Je zajímavé, jak je při realizace interpretu zásobník daleko cennější než fronta. Je třeba také říci, že v situacích, kde jsem měl na výběr mezi frontou a zásobníkem (bylo to jedno, prostě bylo potřeba nějaký dynamický seznam), vybral jsem si už ze setrvačnosti zásobník. Práci programu to mnohdy vůbec neovlivnilo.

4.6.2 Metody šablony

```
Queue() {first=last=0;items=0;}
virtual ~Queue() {empty();}

int getitems()const;   //vrací počet prvků ve frontě
Item* getfirst()const; //vrací první prvek ve frontě

void put(Item *item); //přidá prvek na konec fronty
void get(); //odstraní první prvek z fronty
void empty(); //vyprázdní frontu
```

4.7 Třída Turtle - želva

Tato třída reprezentuje želvu. Dostává z interpretu povely v téměř Logovské tvaru a převádí je na sekvenci plošných posunů a zápisů čar na krychli. Jelikož krychle (viz níže, třída Cube) funguje pouze „pamatovač“ čar, želva dělá potřebné transformace. Tato třída tedy obsahuje důležité geometrické algoritmy pro detekci přechodu přes hranu krychle.

Želva existuje jedna v celé aplikaci, ale nic nebrání tomu, aby bylo vytvořeno i více instancí.

4.7.1 Metoda forward

Metoda forward je vůbec jedinou složitější metodou této třídy. Na druhou stranu je třeba dodat, že jde o jednu z nejdůležitějších metod celého programu. Provádí totiž pohyb želvy, přičemž překládá želví systém do kartézského souřadného systému, přitom také testuje přechody přes hrany krychle, kdy je třeba přesunout želvu na sousední stěnu.

Algoritmus testování kolizí želvy s hranami krychle využívá testování průsečíku dvou úseček. Algortimus je velmi optimalizován, aby bylo dosaženo maximální rychlosti (ačkoliv mám pocit, že na současných počítačích by byl jakýkoliv algoritmus dostatečně rychlý).

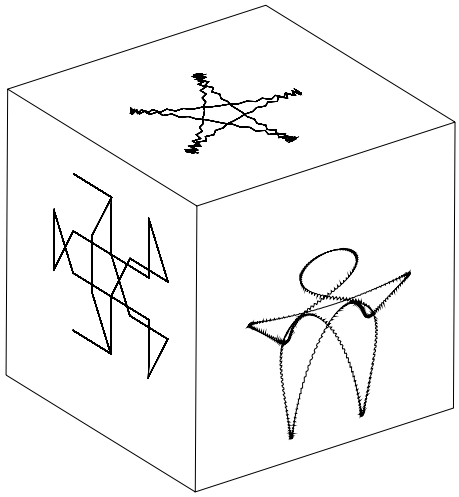
Samotný posun želvy na sousední stěnu je založen na algoritmu z knihy [1]. Přitom samotná krychle je postavena tak, aby každá hrana patřila právě jedné stěně, čili želva

nikdy nestojí na hraně, ani se nestane, že by se želva nemohla rozhodnout, ke které stěně má právě patřit.

Přechod na sousední stěnu se může odehrát několikrát během jednoho příkazu. Proto algoritmus používá rekurze, kdy se forward znovu zavolá. Jelikož však jde o koncovou rekurzi, nahradil jsem ji s výhodou cyklem **while**.

4.7.2 Další metody

```
void back(float l) {forward(-l);}
void right(float a) {setheading(heading+a);}
void left(float a)  {setheading(heading-a);}
void home(); //vrátí želvu do výchozí pozice
void reset(); //resetuje úplně všechno
```



Obrázek 4: Tajemné bytosti z cizích světů?

4.8 Struktura LCell - lexikální buňka

Struktura Lcell představuje lexikální buňku, čili prvek na výstupu lexikálního analyzátoru (viz níže, třída Lexical).

```
struct LCell {
 LID id;       //typ buňky
 union {
  int size;      //délka identifikátoru
  int key;       //číslo operátoru (ascii) nebo číslo keywordu
  float num;     //číslo
 };

 LCell() {}
 LCell(LID ID) {id=ID;}
 LCell(LID ID,int KEY) {id=ID;key=KEY;}
 LCell(LID ID,float NUM) {id=ID;num=NUM;}
};
```

4.9 enum LID - typy lexikálních buňek

Typy lexikálních buňěk jsou použity ve struktuře LCell (viz výše).

konstanta	význam
SID_Number	číslo
SID_Ident	identifikátor (nklíčové slovo)
SID_Operator	operátor (klíčová slova a řetězce)
SID_End	konec textu
SID_Invalid	neplatný znak na vstupu
SID_EOL	konec řádku (také středník)
SID_Var	proměnná - pomocná hodnota
SID_Proc	procedura - pomocná hodnota

4.10 Třída Lexical - lexikální analyzátor

Toto je zřejmě druhá nejdůležitější třída interpretu. Funguje jako vstupní stream pro třídu Interpret (viz výše). Čte vstupní text a překládá ho na lexiální buňky (tokeny?), které umí zpracovat interpret.

4.10.1 Konstruktor

Konstruktor se volá s parametrem poitneru na text, který se má zpracovat. Při volání užitelských procedur se vytvoří vždy nová instance konstruktoru vždy s textem dané procedury.

```
Lexical(const char *TEXT);
```

4.10.2 Metoda getcell

Tato metoda má v podstatě výsnam **input**. Čili je cyklicky volána interpretem a vždy načte další buňku z textu a tu vrátí.

```
LCell getcell();
```

4.10.3 Další metody

```
void setpos(int POS) {pos=POS;eol_oper=0;} //nastaví pozici
int getpos()const {return pos;} //vrací aktuální pozici
const char *gettext() {return text;}
int getcur()const {return cur;} //vrací pozici poslední buňky
void nextline(); //přeskočí v textu na následující řádek
```

4.10.4 Globální funkce lexical_init

Tuto funkci je třeba zavolat jednou pro inicializaci všech tabulek. Je volána z metody **Interpret::init**

```
void lexical_init();
```

4.11 Třída Proc - registrovaná procedura

Tato třída reprezentuje registrovanou proceduru. Registrovaná procedura je každá, kterou lze zavolat (vykonat její kód). Je to v podstatě jen zapouzdřená datová struktura, obsahující informace o názvu proceduty, typu (uživatelská nebo systémová), počtu a názvech jednotlivých parametrů.

4.11.1 Deklarace třídy

```
 IdentName name;  //jméno funkce
 vvar(const char*,prog);//pointer na tělo funkce
 vvar(int,id);    //číslo systémové funkce
 rvar(int,params);//počet paramterů
 rvar(Proc*,prev);//předchozí prvek na zásobníku
 rvar(IdentName*,pname);//jména parametrů
 vprop(int,lib);  //číslo knihovny, ve které je funkce
 char* def;       //definice procedury (intepret nepotřebuje)

public:

 Proc(const char *NAME); //vezme jméno jako char*
 Proc(int size,const char *NAME); //vezme jméno jako část jiného textu
 ~Proc();

 const char* getname() {return name;}

 void setdef(const char *text,int size); //uloží definici jako výřez textu
 void setdef(const char *text); //uloží definici jako jméno+parametry

 const char* getdef();

 void setparams(int PARAMS,IdentName *names);
```

Definice je atribut použitý pouze k výpisu procedur v seznamu, aby uživatel viděl počet a názvy parametrů. Systémové procedury pochopitelně ve skutečnosti nemají názvy parametrů, ale přesto mají zde uvedenou textovou definici.

4.12 Třída ProcStack - zásobník registrovaných procedur

Třídy **...Stack** představují zásobník objektů jiného typu. V tomto případě jde o zásobník objektů typu **Proc**. Zásobník je realizován pomocí template **stack**.

Kromě klasických metod nabízí tento zásobník i několik metod zpříjemňujících práci se zásobníkem. Jsou to metody, které by bylo možno nahradit přímou prací s prvky zásobníku.

4.12.1 Metoda getproc

Tato metoda vyhledá v zásobníku metodu daného jména a vrátí na ni pointer.

```
Proc *getproc(const char *name,int size)const; //vrací pointer na proceduru
```

4.12.2 Metoda markhard

Tato metoda zlouží pro označení všech existujících procedur za **pevné**. Pevné jsou všechny systémové procedury a procedury z knihoven. Důvodem tohoto značení je odlišit od sebe procedury neměnné (pevné) a ty, které se editují a tudíž často mění.

```
void markhard() {hardprocs=items;}
```

4.12.3 Metoda markhard

Tato metoda smaže všechny procedury, které nejsou pevné.

```
void clear();
```

4.13 Struktura SysProc - registrace systémových procedur

Systémové procedury jsou vykonávány zváštním kódem (na rozdíl od uživatelských), ale i tak je nutné mít tyto procedury v seznmu procedur. A k tomuto účelu slouží tato třída, jednoduše přiřazuje lexikálním konstantám (SID_Operator) názvy funkcí a textovou podobu seznamu parametrů (pro výpis v seznamu knihoven).

4.13.1 Deklarace struktury

```
const struct SysProc {
 int params;      //počet vstupních parametrů
 SysProcID id;    //číslo SPID
 char *name;      //jméno procedury
 char *paramnames;//jména parametrů - není potřeba pro interpret
} sysproc[];
```

Obsah struktury je definován v souboru **ProcStack.cpp**.

4.14 Struktura Var - proměnná

Tato struktura představuje proměnnou želvího Loga. Proměnné jsou pouze číselné.

4.14.1 Deklarace struktury

```
 IdentName name;
 float val;
 Var *prev; //ukazatel na předchozí proměnnou na zásobníku

 Var(const char *NAME,float v=0);
 Var(int size,const char *n,float v=0); //bere jméno jako výřez textu
```

4.15 Třída VarStack - zásobník proměnných

Třídy **...Stack** představují zásobník objektů jiného typu. V tomto případě jde o zásobník objektů typu **Var**. Zásobník je realizován pomocí template **stack**.

Kromě klasických metod nabízí tento zásobník i několik metod zpříjemňujících práci se zásobníkem. Jsou to metody, které by bylo možno nahradit přímou prací s prvky zásobníku.

4.15.1 Metoda create

Tato metoda přidá nový prvek na zásobník. Přitom kontroluje, zda proměnná stejného jména již neexistuje. Pokud ano, nevytvoří novou, ale vrací pointer na existující.

```
Var* create(const Var&);
```

4.15.2 Metoda getvar

Tato metoda vrací pointer na danou proměnnou v zásobníku (vyhledá ji podle jména a vrátí pointer nebo NULL, když nic nenajde).

```
Var *getvar(const char *name,int size)const; //vrací pointer na proměnnou
```

4.15.3 Metoda setvalue

Tato metoda nastaví hodnotu proměnné (opět podle názvu proměnné).

```
int setvalue(const char *name,int size,float val);
```

4.16 Třída Library - knihovna procedur

Tato třída není nutná pro běh interpretu, pouze slouží pro uchovávání informací o knihovnách procedur. Pamatuje si jméno knihovny, id-číslo a pointer na obsah knihovny (text).

4.16.1 Deklarace atributů

```
friend Stack<Library>;

static nextid;    //další id-číslo

IdentName name;   //jméno knihovny
rvar(const int,id);    //id-číslo knihovny
rvar(Library*,prev);   //předchozí prvek na zásobníku
rvar(const char*,text);//obsah knihovny
```

4.16.2 Atribut nextid

Tento statický atribut se inkrementuje při založení nového objektu této třídy. Určuje vždy id-číslo nové knihovny.

4.17 Třída LibStack - zásobník knihoven procedur

Třídy **...Stack** představují zásobník objektů jiného typu. V tomto případě jde o zásobník objektů typu **Library**. Zásobník je realizován pomocí template **stack**.

Kromě klasických metod nabízí tento zásobník metodu zpříjemňující práci se zásobníkem. Je to metoda, kterou by bylo možno nahradit přímou prací s prvky zásobníku.

4.17.1 metoda getlib

Tato metoda vrací pointer na knihovnu daného id-čísla.

```
Library* getlib(int id); //vrací knihovnu podle id-čísla
```

4.18 Struktura MathItem - matematická buňka

Interpret vykonává příkazy pomocí dvou zásobníku. Tato třída reprezentuje buňku operačního zásobníku. (Interpret nepoužívá stromové struktury, ale pouze zásobníky.)

4.18.1 Deklarace atributů

```
union {
 float num; //číslo
 void *ptr; //pointer
 int pos;   //pozice ve vstupním textu
};
int oper;      //následující operátor
int prior;     //priorita operace
MathItem *prev;//předcházející prvek na zásobníku
```

Každý objekt této třídy představuje dvojici číslo + operace.

4.19 Třída MathStack - zásobník matematických operací

Třídy **...Stack** představují zásobník objektů jiného typu. V tomto případě jde o zásobník objektů typu **MathItem**, který představuje operační zásobník interpretu. Zásobník je realizován pomocí template **stack**, interpret přitom pracuje výhradně s vrchoem zásobníku.

Kromě klasických metod nabízí tento zásobník metody zpříjemňující práci se zásobníkem. Jsou to metody, které by bylo možno nahradit přímou prací s prvky zásobníku, jde především o čtení atributů prvku na vrcholu zásobníku, jelikož interpret pracuje výhradně s vrcholem zásobníku.

4.19.1 Deklarace třídy

```
class MathStack:public Stack<MathItem> {

public:

 MathStack() {newframe();} //vyrobí základní rámec

 void push(float num,int oper,int prior);
 void push(void *ptr,int oper,int prior);
 void push(int pos,int oper,int prior);

 //vytvoří nový rámec na zásobníku
 void newframe() {push(0,-1,-1);}

 float getnum()const {return top->num;}
 Var *getvar()const {return (Var*)top->ptr;}
 Proc *getproc()const {return (Proc*)top->ptr;}
 int getpos()const {return top->pos;}
 int getprior()const {return top->prior;}
 int getoper()const {return top->oper;}

 void setpos(int pos) {top->pos=pos;}
 void setprior(int PRIOR) {top->prior=PRIOR;}
};
```

4.19.2 Rámce na zásobníku

Operační zásobník podporuje vytváření tzv.rámců. Rámce se používají pro zajištění přednostního vykonání kódu v závorkách. Metoda **newframe** založí nový rámec. Jelikož rámec není nic jiného než zarážka na zásobníku, zrušení rámce se provede klasickou metodou **get** (zarážka rámce musí být na vrcholu zásobníku, to je však splněno vždy).

4.20 Třída CalledProc - volaná procedura

Tato třída představuje proceduru v okamžiku volání. K objektu **Proc** tedy navíc obsahuje informace o vstupních parametrech a jejich hodnotách. Třída CalledProc však třídu Proc nedědí, nýbrž obsahuje.

Objekt této třídy se vytvoří v okamžiku nalezení jména procedury ve vstupním textu. Následuje postupné vyhodnocování vstupních parametrů a jejich přidávání do tohoto objektu. Po přidání posledního parametru (pozná se to podle syntaxe) je volána metoda **Interpret::Call**, která provede vlastní vykonání procedury.

4.20.1 Deklarace třídy

```
class CalledProc {
 friend Stack<CalledProc>;

 rprop(Proc*const,proc);//pointer na existující proceduru
 rprop(const,brackets); //úroveň uzávorkování
 CalledProc *prev;      //předchozí prvek na zásobníku

public:

 VarStack vars;

 CalledProc(Proc *PROC,int BRACKETS=0)
  :_brackets(BRACKETS),_proc(PROC) {}

 void addvar(float val);

 int getparams() {return vars.getitems();} //počet parametrů
```

4.21 Třída CallStack - zásobník volaných procedur

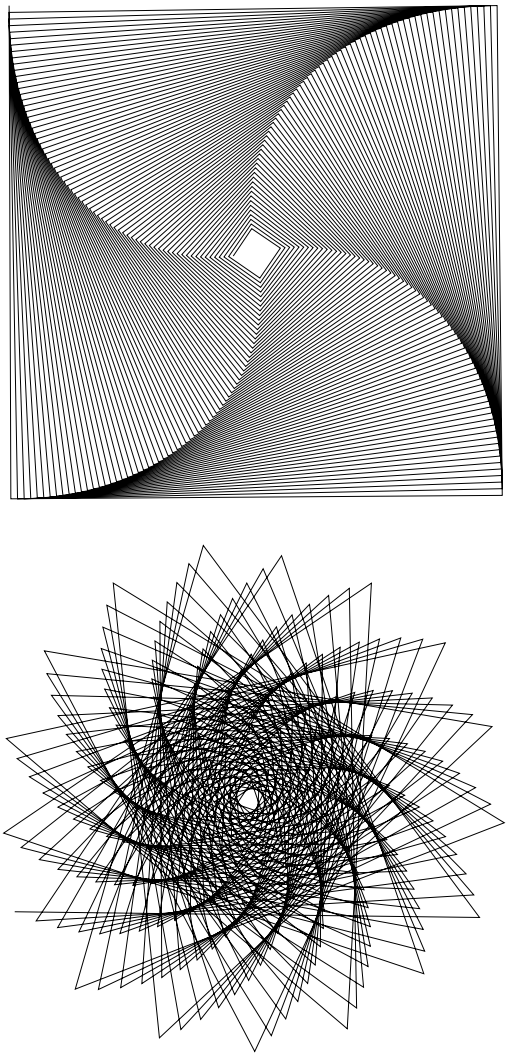
Třídy **...Stack** představují zásobník objektů jiného typu. V tomto případě jde o zásobník objektů typu **Called**, který představuje pomocný zásobník interpretu, používaný v okamžiku volání procedur (když je klasický operační zásobník nedostatečný). Zásobník je realizován pomocí template **stack**, interpret přitom pracuje výhradně s vrchoem zásobníku.

4.22 Struktura CubeItem - prvek na stěně krychle

Tato struktura představuje jednu čáru na krychli, přesněji řečeno jeden příkaz. Čáry na krychli jsou posloupností příkazů **setcolor**, **moveto** a **lineto**, které mají zřejmý význam.

4.22.1 Deklarace struktury

```
struct CubeItem {
```



Obrázek 5: Spirály kreslené procedurou polyspi

```
 const TOP op;
 union {
  struct {int rgb,color;}; //rgb a procentní barvy
  struct {float x,y;};     //stěnové souřadnice
 };
 CubeItem *next; //pointer na další prvek ve frontě

 //vytvoření buňky jako moveto/lineto
 CubeItem(TOP OP,float X,float Y):op(OP),x(X),y(Y) {next=0;}

 //vytvoření buňky jako setcolor
 CubeItem(int RGB,int COLOR):op(TOP_Setcolor)
   ,rgb(RGB),color(COLOR) {next=0;}
};
```

4.23 Třída CubeFace - stěna krychle

CubeFace je stěna krychle. Stěna krychle obsahuje frontu příkazů (čar na krychli). Navíc obsahuje metody pro přidávání dalších čar. Želva tedy nepracuje s krychlí jako celkem, ale s každou stěnou zvlášť. Je to důsledek detailně rozpracované datové abstrakce, kterou jsem použil.

4.23.1 Deklarace třídy

```
class CubeFace {

 Queue<CubeItem> lines; //seznam čar
 int rgb,color;         //rgb a procentní barvy
 float x,y;             //stěnové souřadnice

public:

 CubeFace() {clear();}

 void clear(); //smaže všechny čáry

 void addline(int color,float x1,float y1,float x2,float y2); //přidá linku

 CubeItem* getfirst() {return lines.getfirst();}
};
```

4.24 Třída Cube - krychle

Tato třída existuje v jedné instanci a představuje krychli (mohlo by jich však být i více). Skládá se ze šesti stěn a obsahuje také unikátní číslo, které se inkrementuje při každé nové čáře na krychli. Inkrementaci zajišťují metody **CubeFace**, čili jde o efektivní a elegantní implementaci.

4.24.1 Čáry na krychli

Čáry na krychli jsou uloženy jednoduše v poli šesti stěn.

```
private:
CubeFace face[6];
public:
CubeFace& getface(int i) {return face[i];}
void clear(); //smaže čáry, ale nemění ostatní věci
```

4.24.2 Unikátní číslo

Toto číslo slouží při vykreslování krychle k optimalizaci rychlosti. Umožní nám vykreslovat krychli jen, když je třeba.

```
rvar(int,unique); //unikátní číslo reprezentující počet čar
void resetunique() {unique=0;}
void incunique() {unique++;}
```

5 Geometrické operace

Tato kapitola popisuje tu část programu, která se týká geometrických operací, transformací a zobrazování grafiky. Grafický engine používá téměř výhradně vektorový počet, který je upřednostněn především pro jeho stručné zápisy všech algoritmů. Kromě toho se používají transformační matice a pro vlastní kreslení 2D čar je navržen systém tříd se společným čistě abstraktním předkem. Převážná část geometrických operací je převzata z mého letošního zápočtového programu do geometrie.

5.1 Třída GVector2D - dvojrozměrný vektor

Tato třída představuje plošný (dvousložkový) vektor. Třída implementuje všechny obvyklé vektorové operace, aby bylo možno nasadit vektorovou aritmetiku.

5.1.1 Deklarace třídy

```
struct GVector2D {

 real x,y;

 GVector() {}
 GVector(real X,real Y) {x=X,y=Y;}
 GVector(GVector &v) {x=v.x,y=v.y;}

 int operator==(GVector v)const;

 GVector operator*(real m)const;
 GVector operator/(real m)const;

 GVector operator*(GVector v)const;
 GVector operator/(GVector v)const;
 GVector operator+(GVector v)const;
 GVector operator-(GVector v)const;

 void operator+=(GVector v) {x+=v.x,y+=v.y;}
 void operator-=(GVector v) {x-=v.x,y-=v.y;}
 void operator*=(real m) {x*=m,y*=m;}
 void operator/=(real m) {x/=m,y/=m;}

 void operator=(GVector3D &v) {x=v.x,y=v.y;}
 void operator=(GVector3D v) {x=v.x,y=v.y;}

 GVector operator-()const {return GVector(-x,-y);}
};
```

5.2 Třída GVector3D - třírozměrný vektor

Tato třída implementuje prostorový (třísložkový) vektor. Její význam je stejný jako u plošného vektoru **Vector2D**.

5.2.1 Deklarace třídy

```
struct GVector3D {

 real x,y,z;

 GVector() {}
 GVector(real X,real Y,real Z) {x=X,y=Y,z=Z;}
 GVector(const GVector &v) {x=v.x,y=v.y,z=v.z;}

 int operator==(const GVector &v)const;

 GVector operator*(real m)const {return GVector(x*m,y*m,z*m);}
 GVector operator/(real m)const {return GVector(x/m,y/m,z/m);}

 GVector operator+(const GVector &v)const;
 GVector operator-(const GVector &v)const;
 GVector operator*(const GVector &v)const;
 GVector operator/(const GVector &v)const;

 void operator+=(GVector &v) {x+=v.x,y+=v.y,z+=v.z;}
 void operator-=(GVector &v) {x-=v.x,y-=v.y,z-=v.z;}
 void operator*=(real m) {x*=m,y*=m,z*=m;}
 void operator/=(real m) {x/=m,y/=m,z/=m;}

 GVector operator-()const {return GVector(-x,-y,-z);}
};
```

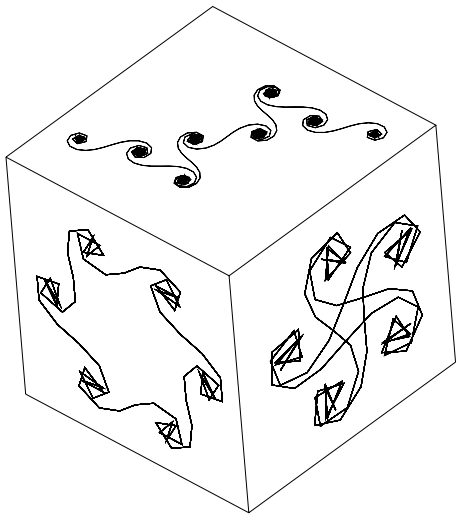
5.3 Třída GMatrix - čtyřrozměrná transformační matice

Tato třída reprezentuje klasickou čtyřrozměrnou transformační matici. Ačkoliv při použití baricentrických souřadnic není nasazení matic tak velké, úplně bez nich by se program neobešel. Třída implementuje metody potřebné pro maticové transformace a vektorovou aritmetiku.

5.3.1 Deklarace třídy

```
struct GMatrix {

 real a[4][4];

 GMatrix() {Unit();}
 GMatrix(GMatrix &m) {*this=m;}

 GMatrix& operator+=(GMatrix &m);
 GMatrix& operator*=(GMatrix &m);
 GMatrix& operator=(GMatrix &m);

 GMatrix& Null(); //nulová matice
 GMatrix& Unit(); //jednotková matice
```

```
 GMatrix& Shift(const GVector3D &v); //přidá posunutí
 GMatrix& Shift(real x,real y,real z); //přidá posunutí

 GMatrix& Scale(const GVector3D &v); //přidá změnu měřítka
 GMatrix& Scale(real s) {return Scale(GVector3D(s,s,s));}

 GMatrix& RotateX(real a); //přidá otočení kolem osy X
 GMatrix& RotateY(real a); //přidá otočení kolem osy Y
 GMatrix& RotateZ(real a); //přidá otočení kolem osy Z

 GVector3D operator*(GVector3D v); //transformace bodu
};

inline GVector3D operator*(GVector3D v,GMatrix &m) {return m*v;}

inline GVector3D& operator*=(GVector3D &v,GMatrix &m);
```



Obrázek 6: Fraktály kreslené procedurou inspi

5.4 Třída GStream - stream grafických operací

Tato abstraktní třída slouží jako výstupní stream při kreslení. Umožňuje provádět operace **setcolor**, **moveto** a **lineto**, přičemž vlastní implementace těchto operací je ponechána až na potomcích.

5.4.1 Deklarace třídy

```
class GStream {
public:
 GStream();
 virtual ~GStream() {}

 virtual void close()=0; //ukončí výstup
```

```
 virtual void setcolor(int RGB,int COLOR)=0; //nastaví barvu

 virtual void moveto(int x,int y)=0; //nastaví aktuální pozici

 virtual void lineto(int x,int y)=0; //čára z aktuální pozice do bodu

 virtual void line(int x1,int y1,int x2,int y2); //čára z bodu do bodu

 virtual void setpen(int PS)=0; //nastaví styl čáry

 int getwidth()const {return width;}
 int getheight()const{return height;}
};
```

5.5 Třída GStreamVCL - obrazovkový stream grafických operací

Tento potomek třídy **GStream** implementuje kreslení na canvas pomocí VCL. VCL je Win32 nadstavba C++ Builderu.

5.5.1 Konstruktor

V konstruktoru je třeba nastavit velikost okna, do kterého se bude kreslit. Okno může být obdélníkové, ačkoliv pro kreslení krychle nemá obdélníkové okno smysl. Třída **TCanvas** je implementována ve VCL.

```
GStreamVCL(TCanvas *canvas,int width,int height);
```

5.6 Třída GStreamPS - souborový stream grafických operací

Tento potomek třídy **GStream** implementuje kreslení v podobě výstupu do souboru EPS. EPS soubory jsou odladěny tak, aby je bylo možno vkládat do Texových dokumentů a aby tam vypadaly dobře (což o EPS souborech z jiných studentských programů nelze říci).

5.6.1 Metoda open

Jelikož při zakládání výstupního souboru může dojít k chybě, nelze soubor otevírat již v konstruktoru. Proto je zde speciální metoda open. Druhý parametr určuje velikost obrázku. Obrázek je vždy čtvercový (pochopitelně).

```
int open(const char *fname,int size);
```

5.7 Třída GView - zobrazovací engine

GView je vlastní kreslící engine. Jeho úloha spočívá v načtení obsahu krychle, jeho zpracování a vykreslení pomocí výstupního streamu (viz výše, třída GStream).

5.7.1 Metoda assign

Tato metoda nasměruje výstup do výstupního streamu. Existují dvě přetížené verze, jedna z nich inicializuje výstup na obrazovku, druhá do souboru EPS, přitom dojde k vytvoření objektu typu **GStream...**.

```
void assign(TCanvas*,int width,int height);//výstup na obrazovku
int assign(const char *fname,int size);    //výstup do EPS
```

5.7.2 Metoda close

Tato metoda uzavře výstup (volá se po kreslení). Je také automaticky volána v destruktoru.

```
void close();
```

5.7.3 Metoda paint

Tato nejdůležitější metoda celé třídy provede vlastní vykreslení prostorové krychle. Na vstupu jsou parametry zobrazení.

```
void paint(GVector3D &angle,float scale,GVector2D shift);
```

parametr	význam
angle	prostorový úhel natočení krychle
scale	velikost krychle vzhledem k velikosti okna
shift	vektor posunutí obrazu (zlomek velikosti okna)

Natočení krychle je provedeno postupným otočením kolem os x,y,z. Následuje zvětšení, kdy velikost 1 znamená „tak akorát“, což je o maličko víc než druhá odmocnina ze tří (délka prostorové úhlopříčky krychle). Nakonec se provede posunutí, hodnoty jsou udávány ve zlomcích velikosti okna, u obdélníkového okna se bere kratší z obou stran.

5.7.4 Metoda paintfaces

Tato metoda provede vykreslení krychle v plošné podobě, čili jak se říká „plášť krychle“.

```
void paintfaces(); //nakreslí rozvinutou krychli
```

6 Uživatelské rozhraní

Uživatelské rozhraní je postaveno na knihovně VCL, což je základní nástroj C++ Builderu. Každá třída v této kapitole představuje jedno okno (v originále **form**).

6.1 Třída TMainForm

Toto je hlavní okno aplikace. Třída obsahuje velké množství vizuálních komponent a dalších věcí, které souvisejí jen s vzhledem okna. Kromě toho jsem doplnil několik metod, které jsou použity při běhu aplikace.

6.1.1 Metody třídy

```
private:

int SaveFile();       //uloží editovaný soubor
int ShouldSaveFile(); //měl by se uložit editovaný soubor
void updatecaption(); //updatuje caption okna podle jména souboru
void PaintMenuClear();//smaže zaškrtnutí v menu
void FindNextString();//najde další string podle FindDialog

//otevře a načte soubor
const char* LoadFile(const char *caption,const char *fname=0);
//druhý parametr používá jen OpenFile při volání z OLE

public:

 TMainForm(TComponent* Owner);
 void DisplayHint(TObject *Sender);

 void regprocs(); //zaregistruje uživatelské procedury

 void OpenFile(const char *fname=0); //otevře soubor
 void NewFile(const char *fname);    //založí nový soubor

protected:

 void WMEraseBkgnd(TWMEraseBkgnd&);
```

Poslední metoda funguje jako handler zprávy **WM_EraseBackground**, který nic nedělá a tak jednak zamezuje blikání a současně zrychluje překreslování okna.

6.2 Třída TAboutForm

Toto je okno dialogu „About“.

6.3 Třída TAbortForm

Toto je okno, které je vidět při vykonávání programu v Logu. V **OnPaint** handleru se provádí vykonávání Loga a pomocí funkce „ProcessMessages“ je umožněno současně zjistit

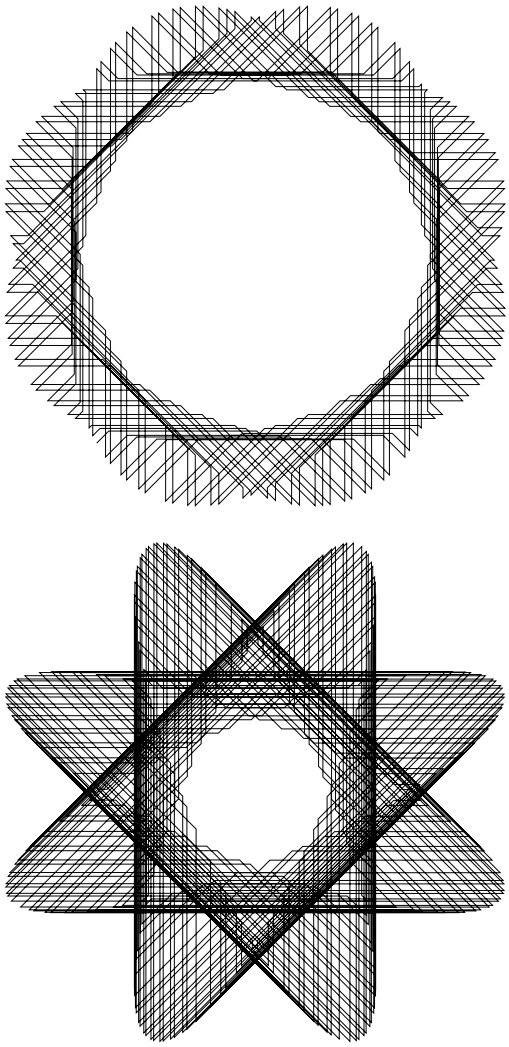
stisk tlačítka „Abort“. V tom okamžiku se nastaví přepínač v interpretu a ten se automaticky přeruší. V podstatě nejde o nic jiného, než o kooperativní multitasking. Standardní preemptivní multitasking by totiž způsobil víc škody než užitku.

6.4 Třída TLibForm

Toto je okno, ve kterém se vypisuje seznam knihoven a jejich procedur. Hlavním úkolem tohoto okna je především umožnit uživateli snadno nalézt kód libovolné procedury.

6.5 Třída TLibShowForm

Toto je nemodální okno, ve kterém se ukazuje kód knihovny.



Obrázek 7: Fraktály kreslené procedurou shrink2

6.6 Třída ITexImpl

Tato třída implementuje ActiveX rozhraní. Pro navázání spojení je nutno nejprve alespoň jednou spustit CubeTurtle (tím dojde k registraci serveru). Potom se na něj lze odkazovat jménem **CubeTurtle.Tex**.

Metody ActiveX rozhraní jsou napsány podle požadavků. Přibalen je i testovací klient, pomocí něhož lze funkce vyzkoušet. Server funguje jako inproc i outproc (snad).

6.7 Konfigurační soubor

Program CubeTurtle ukládá všechna nastavení do konfiguračního souboru **CubeTurtle.ini**. Jde o klasický textový soubor, který má stejnou strukturu, jako ve všech ostatních mých programech, čili první slovo na řádku je identifikátor atributu, zbytek řádku (za mezerou) je hodnota tohoto atributu.

Pro načítání konfiguračního souboru slouží globální funkce **loadini()** a pro jeho ukládání rovněž globální funkce **saveini()**.

7 Softwarové inženýrství

Tuto kapitolu jsem věnoval důležitým poznatkům z knihy [4]. Ačkoliv tato kniha nemá nic společného s želví geometrií, její obsah byl pro mou práci velmi podstatný. Důkazy dvanácti zde uvedených tvrzení nechť čtenář hledá v uvedené knize.

1. Algoritmus, který by dával 100% přesné řešení, obvykle neexistuje.
2. Programátor průměrně vyprodukuje 3000 řádků kódu za rok.
3. Větší produkty (překladače, apod.) se tvoří obvykle 4-6 let, po polovině této doby obvykle „celkem chodí“.
4. Dobu práce nelze libovolně zkracovat.
5. Programy psané ve spěchu jsou delší a je v nich více chyb.
6. 82% chyb programů vzniká při definici požadavků na projekt, jen 1% chyb vzniká při programování.
7. U softwaru se obvykle nedodrží smluvené termíny.
8. Pravidlo úspěchu zní: Spěchej pomalu!
9. Tvorba dokumentace je často pracnější a obtížnější než vlastní programování, ale má to svůj efekt.
10. Schopnost programovat nelze jednoduše otestovat.
11. Nejnáročnější částí celého systému je detekce a řešení chybových stavů.
12. Princip maximální možné lenosti: Většina požadavků na projekt je naprosto zbytečná, ale stojí až 90% práce.

Reference

[1] H. Abelson, A. diSessa: *Turtle geometry*. The MIT Press.

[2] A.Blaho, I.Kalaš, P.Tomcsányi: *Comenius Logo*. Projekt Comenius, Praha 1994.

[3] Kolektiv autorů: *MSDN Library*. Microsoft Corporation, USA 1998.

[4] Jaroslav Král, Jiří Demer: *Softwarové inženýrství*. Academia Praha, Praha 1991.

[5] Libor Brodský: *OLETest*. KMI UP, Olomouc 1999.