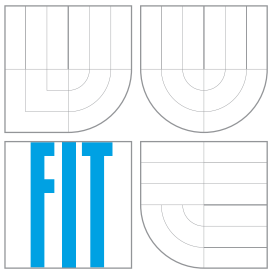
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÝ ENGINE S PODPOROU SKRIPTOVÁNÍ

GRAPHIC ENGINE WITH SCRIPTING SUPPORT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LEOŠ POL
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL VYSKOČIL
SUPERVISOR

BRNO 2008

Abstrakt

Cílem této bakalářské práce je navrhnout řešení pro propojení zvoleného grafického enginu a skriptovacího jazyka. Tento návrh pak realizovat v demonstrační aplikaci a uvést další pokračování projektu.

Klíčová slova

engine, skriptování, spojení, OGRE, Python

Abstract

The objective of this bachelor's thesis is to propose solution for a cross connection between specific graphic engine and a skripting language. Implementation this proposal into a demo application and present further continuing of this project.

Keywords

engine, skripting, connection, OGRE, Python

Citace

Leoš Pol: Grafický engine s podporou skriptování, bakalářská práce, Brno, FIT VUT v Brně, 2008

Grafický engine s podporou skriptování

Prohlášení

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

.....
Leoš Pol
14. května 2008

Poděkování

Touto cestou bych chtěl poděkovat vedoucímu práce Ing. Michalovi Vyskočilovi za odborný dohled a volnost při vytváření této práce.

Obsah

Kapitola 1

Úvod

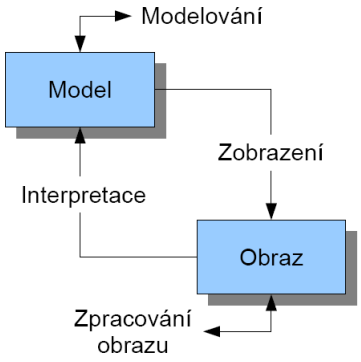
Tato kapitola uvádí do problematiky, která je obsahem této práce, teoreticky rozebírá dvě důležité logické části a to jsou:

- počítačová grafika a
- skriptovací jazyky.

Dále pak má za účel vytvořit seznam dostupných alternativ pro obě tyto části, které pak budou sloužit jako základ pro implementační analyzu a tedy i řešení samotného zadání této práce.

1.1 Počítačová grafika[2]

Počítacová grafika je součástí oboru Informatika, která se zabývá analýzou nebo tvorbou grafické obrazové informace (viz. obr. 1.1). Máme-li již k dispozici nějakou obrazovou informaci, získanou snímáním nebo měřením (kamerou atd.), potřebujeme ji většinou dále zpracovat (upravit, vylepšit) nebo analyzovat (interpretovat a rozpoznat objekty). Jedná se o aplikace typu *Počítacové vidění*. Máme-li naopak k dispozici virtuální počítacový (matematický) popis (model) objektu, potřebujeme ho převést na grafickou obrazovou informaci (syntéza) a zobrazit.



Obrázek 1.1: Schéma počítačové grafiky

Počítačovou grafiku si mužeme rozdělit z různých hledisek:

- podle dimenze zpracovávané informace například na 2D a 3D,
- podle oblasti použití například na zpracování fotek, videa, malování nebo CAD, vizualizaci, animaci atd., a
- z hlediska uživatele grafických programů nebo z hlediska jejich programátora.

Se zrodem pracovních stanic jako LISP machine, Quantel paintbox a Silicon Graphics přišla 3D počítacová grafika založená na vektorové grafice. Namísto toho, aby počítac ukládal informace o bodech, čarách a křivkách na dvojrozměrné ploše, ukládá počítac pozici bodu, úseček, a ploch v trojrozměrném prostoru.

Polygony v trojrozměrném kartézském souřadnicovém systému jsou základní prvky téměř všech 3D systému (zvaných engine). Každé zobrazované těleso v takovém systému se skládá z polygonu. Proto většina z nich ukládá body (což jsou souřadnice v 3D prostoru), úsečky (které tyto body propojují), plošky mezi těmito úsečkami a sekvence plošek, které dohromady tvoří 3D polygon. Dále se pro zobrazení tyto tvary stínují, texturují a rasterizují.

1.1.1 Zástupci

Tato sekce slouží jako přehled dostupných open source 3D grafických enginů, které jsou nejlépe hodnocené vývojáři nejrůznějších grafických aplikací a her po celém světe. Jako podklad byla použita databaze [1].

OGRE

OGRE (Object-Oriented Graphics Rendering Engine) je scénově orientovaný, flexibilní 3D engine napsaný v C++ a navržený, aby byl jednoduchý a intuitivní pro vývojáře, kteří produkují hry využívající 3D hardware. Abstraktní knihovna tříd překrývá všechny detaily v používání knihoven nízké úrovně jako je Direct3D a OpenGL, a poskytuje rozhraní založené na objektech ve scéně a další intuitivní třídy.

Irrlicht

Irrlicht Engine je multiplatformní vysoce výkonný real-time 3D engine napsaný v C++. Je to silné vysokoúrovňové API pro vytváření kompletně 3D a 2D aplikací jako jsou hry nebo vědecké vizualizace. Integruje všechny novodobé techniky pro vizualní prezentaci jako jsou stíny, částicové systémy, animaci postav, kolizní detekci a další.

Crystal Space

Crystal Space je volně šířitelný multiplatformní SDK pro real-time 3D grafiku ve specifických hrách.

Panda3D

Panda3D je výkonný renderovací engine pro SGI, Linux, Sun, a Windows. Jádro enginu je napsané v C++. Panda3D poskytuje skriptovací interface pro Python.

Blender Game Engine

Blender je open source 3D modelovací, renderovací, animační a real-time 3D systém pro hry a simulaci. Blender je často vylepšován a podporován aktivní komunitou.

1.2 Skriptovací jazyky[3]

Skript je v Informatice zdrojový kód programu, který je tzv. interpretován, tj. čten a spouštěn za běhu speciálním procesem, tzv. interpretem.

Jakožto programový kód, je i skript vytvořen v programovacím jazyku (hovoří se o tzv. skriptovacím jazyku), který má přesně stanovenou formální gramatiku (tj. pravidla, syntaktické elementy, jazykové konstrukty atd).

Typicky skript těží z výhody, že se nemusí překládat, a často tvoří rozšiřitelnou (parametrickou) část nějakého softwarového projektu, která se může měnit, aniž by bylo potřeba pokaždé rekompilovat hlavní spustitelný soubor. Skripty najdeme u her, složitejších softwarových řešení nebo jako hlavní součást dynamických internetových stránek a podobně.

Protikladem k interpretovanému kódu je program přeložený (kompilovaný) do strojového kódu.

Výhody:

- není nutné mít nainstalovaný kompilátor a provádět po každé změně kódu kompilaci,
- snadnější údržba, vývoj a správa kódu,
- některé skripty umožnují interpretaci kódu z řetezce (jako například funkce eval() v PHP) – něco takového překládané programy z principu nedokáží, a
- interpret obvykle poskytuje programátorovi zabudovaný abstraktní datový typ asociativní pole.

Nevýhody:

- rychlost – interpretace stojí určitý strojový čas a nikdy nebude tak rychlá jako spouštění přeloženého (a optimalizovaného) programu,
- o trochu vyšší paměťová náročnost. Interpret musí být spuštěn a tedy zabírá určitou operační paměť, a
- skriptovací jazyky mají většinou větší omezení než překládané programovací jazyky (např. co do přístupu do paměti, ovládání tzv. rukovětí procesu a kontextových zařízení apod).

1.2.1 Zástupci

Tato sekce slouží jako přehled dostupných skriptovacích jazyků, omezuje se však pouze na ty nejznámnější a teoreticky použitelné v této práci.

Perl[5]

Perl je interpretovaný programovací jazyk vytvořený Larry Wallem v roce 1987. Největšího rozšíření dosáhl ve verzi 4 z roku 1991. Verze 5 přinesla četná vylepšení, především výkonné datové struktury a možnost objektového programování.

Larry Wall se při konstrukci jazyka řídil heslem: dá se to udělat více způsoby (there's more than one way to do it). Umožňuje psát krátké programy jednoduše a rychle a přítom nebrání v psaní těch složitých. Jeden ze způsobu je přitom obvykle velmi stručný, takže Perl získal nezaslouženou pověst jazyka, ve kterém se tvoří nesrozumitelný a neudržovatelný kód. Tato kritika ale není oprávněná, Perl je vhodný k řešení malých i velkých problému. Schopnosti a nástroje, které se používají u velkých projektů, lze použít i v krátkých skriptech.

Python[7]

Python je dynamický interpretovaný jazyk, který v roce 1990 navrhl Guido van Rossum. Někdy bývá zařazován mezi takzvané skriptovací jazyky. Python je hybridní jazyk (nebo také víceparadigmatický), to znamená, že umožňuje při psaní programu používat nejen objektově orientované paradigma, ale i procedurální a v omezené míře i funkcionální, podle toho komu co vyhovuje nebo se pro danou úlohu hodí nejlépe. Python má díky tomu vynikající vyjadřovací schopnosti. Kód programu je ve srovnání s jinými jazyky krátký a dobře čitelný.

K význačným vlastnostem jazyka Python patří jeho jednoduchost z hlediska učení. Bývá dokonce považován za jeden z nejvhodnějších programovacích jazyků pro začátečníky. Tato skutečnost je dána tím, že jedním z jeho silných inspiračních zdrojů byl programovací jazyk ABC, který byl jako jazyk pro výuku a pro použití začátečníky přímo vytvořen. Python ale současně bourá zažitou představu, že jazyk vhodný pro výuku není vhodný pro praxi a naopak. Podstatnou měrou k tomu přispívá čistota a jednoduchost syntaxe, na kterou se při vývoji jazyka hodně dbá.

Dostupnost zdrojového kódu a vlastnosti jazyka C umožňují zabudovat interpret jazyka Python do jiné aplikace psané v jazycích C nebo C++. Takto zabudovaný interpret jazyka Python pak představuje nástroj pro pružné rozšiřování funkčnosti výsledné aplikace zvenčí. Existuje i projekt pro užší spolupráci s C++ nazvaný Boost.Python.

Ruby[8]

Ruby je interpretovaný skriptovací programovací jazyk. Díky své jednoduché syntaxi je poměrně snadný k naučení, přesto však dostatečně výkonný, aby dokázal konkurovat známějším jazykům jako je Python a Perl. Je plně objektově orientovaný – vše v Ruby je objekt.

Tvůrcem Ruby je jediný člověk – Yukihiro Matsumoto, známý také pod přezdívkou Matz. Ten jako zastánce objektově orientovaného programování hledal v první polovine 90. let skriptovací jazyk, který by mu vyhovoval. Avšak Perl mu připadal v té době málo výkonný a Python zase nebyl natolik objektový, jak by chtěl. A tak se rozhodl, že vytvoří vlastní jazyk. Práce na něm započaly v roce 1993, první verze byla uverejnena v roce 1995.

PHP[6]

PHP je skriptovací programovací jazyk, určený především pro programování dynamických internetových stránek. Nejčastěji se začleňuje přímo do struktury jazyka HTML, XHTML ci WML, což je velmi výhodné pro tvorbu webových aplikací. PHP lze ovšem také použít i k tvorbě konzolových a desktopových aplikací.

Od roku 1994 je PHP jedním z nejpoužívanějších způsobu tvorby dynamicky generovaných WWW stránek. Jeho tvůrce Rasmus Lerdorf jej vytvořil pro svou osobní potřebu

přepsáním z Perlu do jazyka C. Sada skriptu byla vydána ještě v témže roce pod názvem Personal Home Page Tools, zkráceně PHP.

Javascript[4]

Javascript je multiplatformní, objektově orientovaný skriptovací jazyk, jehož autorem je Brendan Eich z tehdejší společnosti Netscape. Nyní se zpravidla používá jako interpretovaný programovací jazyk pro WWW stránky, často vkládaný přímo do HTML kódu stránky. Jeho syntaxe patří do rodiny jazyků C/C++/Java.

Program v JavaScriptu se obvykle spouští až po stažení WWW stránky z Internetu (tzv. na strane klienta), na rozdíl od ostatních jiných interpretovaných programovacích jazyků (napr. PHP a ASP), které se spouštějí na straně serveru ještě před stažením z Internetu. Z toho plynou jistá bezpečnostní omezení, JavaScript např. nemůže pracovat se soubory, aby tím neohrozil soukromí uživatele.

Kapitola 2

Implementační analýza

Tato kapitola popisuje návrh řešení zadání, stejně tak rozebírá problémy, které bylo nutné vyřešit před samotným započetím implementační fáze této práce.

2.1 Sloučení skriptovacího jazyka a grafického engine

Otázka spojení skriptovacího jazyka společně s grafickým enginem je hlavním předmětem této práce. Nabízí se několik možných postupu, jak tohoto tíženého výsledku docílit:

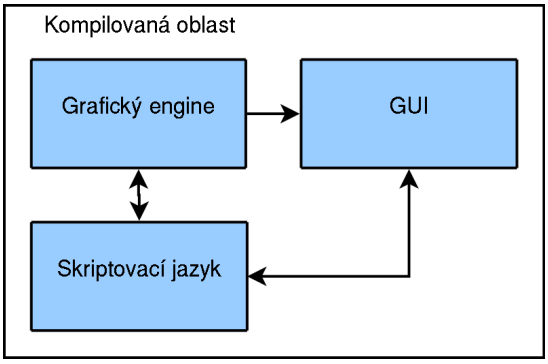
1. vybrat grafický engine napsaný v kompilovaném jazyce a rozšířit ho o vlastní implementaci skriptovacího pseudojazyka,
2. vybrat grafický engine napsaný v kompilovaném jazyce a při sestavování přilinkovat některý z interpretů skriptovacího jazyka,
3. vybrat grafický engine napsaný v kompilovaném jazyce, který interně obsahuje skriptovací jazyk,
4. vybrat grafický engine napsaný v kompilovaném jazyce a použít obal, který zpřístupní rozhraní ve zvoleném skriptovacím jazyce, a
5. vybrat grafický engine napsaný přímo ve zvoleném skriptovacím jazyce.

Z výše uvedených možností byla vybrána varianta kompilovaného grafického engine s použitím obalu, který umožní využití všech možností tohoto engine přímo ze skriptovacího jazyka. Podrobnější rozbor, který také vedl k tomuto závěru je popsán v následujících podsekcích.

2.1.1 Kompilovaný grafický engine s vlastním skriptovacím jazykem

První možnost značně přesahuje rámec této práce a zároveň by vedla ke značnému omezení konstrukcí ve skriptovacím jazyce. Nadruhou stranu by se toto řešení vyznačovalo vysokým zobrazovacím, ale i skriptovacím výkonem[1]. Demonstrační aplikace by byla kompletně napsaná v kompilovaném jazyce, která by využívala knihoven grafického engine a GUI (viz. obr. 2.1).

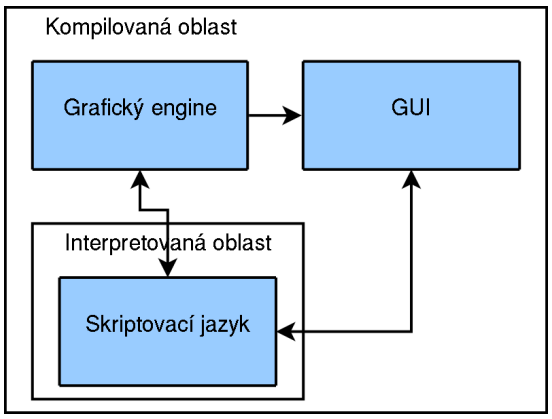
[1] Vzhledem k faktu, že by skriptovací jazyk byl napsán autorem, výkon by byl ovlivněn kvalitou lexikálního a sémantického analyzátoru a samotného prováděče kódu. Přesto ale obecně lze očekávat, že výkon by byl vyšší než u komplexního skriptovacího jazyka.



Obrázek 2.1: Schéma kompilovaného grafického engine s vlastním skriptovacím jazykem

2.1.2 Kompilovaný grafický engine s přilinkovaným interpretrem skriptovacího jazyka

Tato možnost prakticky pouze rozšiřuje variantu v předchozí sekci. Odstraňuje její hlavní nedostatek a to je robustnost skriptovacího jazyka. Velkou nevýhodou pak může být komplikovaná a někdy i nemožná komunikace mezi grafickým enginem a skriptovacím jazykem, tzn. konverze typů, volání funkcí, řešení vyjímek apod (viz. obr. 2.2).

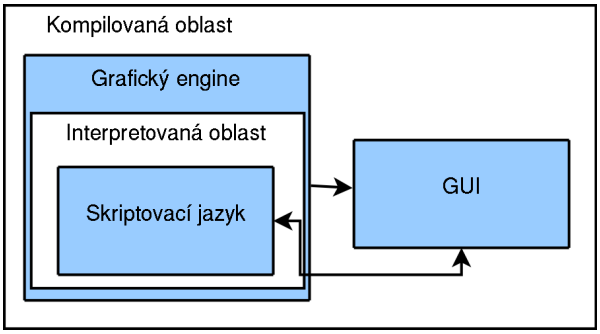


Obrázek 2.2: Schéma kompilovaného grafického engine s přilinkovaným interpretrem skriptovacího jazyka

2.1.3 Kompilovaný grafický engine obsahující skriptovací jazyk

Existuje nepřeberné množství grafických enginů, které v sobě již obsahují a podporují jeden ze skriptovacích jazyků, ať už se jedna o typický jazyk Lua, nebo také Python, TCL, Basic, nebo specifický pro daný engine. Tento způsob řešení není možný hlavně z toho důvodu, že samotné propojení figuruje jako hlavní úkol této práce, ale také proto, že mnohdy skriptovací jazyk vložený do grafického engine omezuje uživatele, který je pak nucen zadávat příkazy operující pouze nad grafickým enginem a je nemožné jakýmkoliv způsobem toto rozhraní v budoucnu uživatelem rozšířit (viz. obr. 2.3).

Výhodou pak zpravidla bývá fakt, že skriptovací jazyk komletně (anebo z větší části) implementuje veškeré grafické funkce na nízké úrovni.



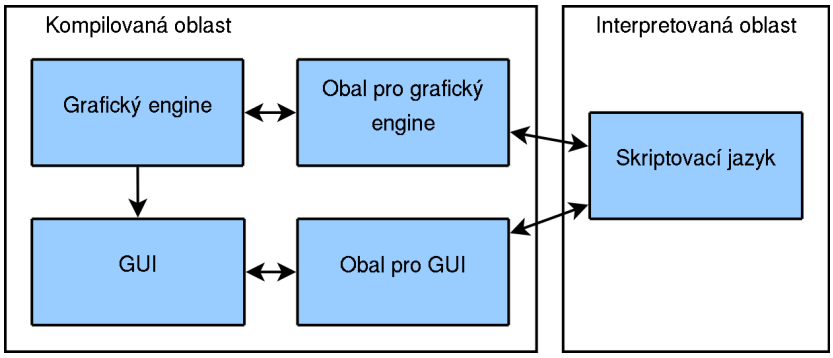
Obrázek 2.3: Schéma kompilovaného grafického engine obsahující skriptovací jazyk

2.1.4 Skriptovací obal nad kompilovaným grafickým enginem

Tato varianta bývá z drtivé většiny vytvářena a udržována komunitou lidí, která se vytvořila kolem populárního grafického engine, mající v oblibě jiný programovací jazyk, v tomto případě jazyk skriptovací, než v jakém je daný engine napsaný. V jednoduchosti jde o vytvoření obalu (označovaného někdy jako port nebo wrapper) kolem API publikující enginem (viz. obr. 2.4).

Hlavním úkolem tohoto obalu je poskytnutí API v cílovém programovacím jazyce, přes které bude docházet k volání originálního API grafického engine. Z toho také vyplývá nutnost řešit konverze argumentů, návratových hodnot, tříd a instancí tříd, zpětná volání, a další. Všechno ale řeší tento obal a z pohledu skriptovacího jazyka se vše jeví transparentně.

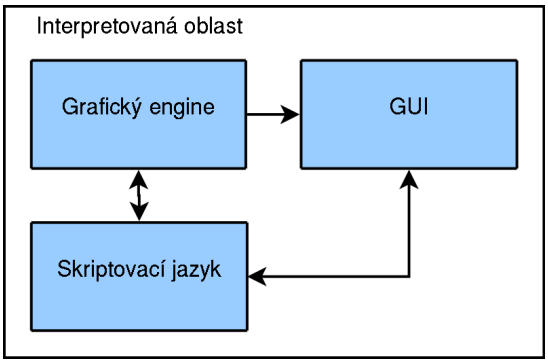
Je důležité také uvést, že někdy výsledné API obalu neodpovídá původnímu API grafického engine. Důvody většinou vychází z rozdílnosti kompilovného a skriptovacího jazyka, jako například syntaktické nebo sémantické omezení, možnost přívětivějšího zápisu v jednom z jazyků, apod. Dalo by se ale říci, že většinou vývojáři obalu se snaží dodržet jednotné rozhraní, hlavně kvůli jednoduchosti obalu a jednotné dokumentace.



Obrázek 2.4: Schéma kompilovaného grafického engine s obalem pro skriptovací jazyk

2.1.5 Skriptovací grafický engine

Poslední zde uvedená možnost je použít grafický engine, který je již implementován ve skriptovacím jazyce. Pro svoji funkci využívá některou ze standarních nizkoúrovňových zobrazovacích knihoven[2]. Z předchozích vět je patrné, že hlavní nevýhoda bude výkonnost samotného grafického engine, naopak integrace skriptovacího jazyka do grafického engine je vyřešena samotným návrhem (viz. obr. 2.5).



Obrázek 2.5: Schéma skriptovacího grafického engine

2.2 Python

Ze sekce 1.2.1 je patrné, že dnešní doba nabízí nepřeberné množství kvalitních skriptovacích jazyků. Přesto byl vybrán jazyk Python, který nabízí výhod:

- je snadné se ho naučit – nezkušený uživatel nebude muset strávit dny a týdny učením syntaxe jazyka, pro základní ovládání postačí běžná znalost objektově orientovaného programování a elementární povědomí o syntaxi Pythonu,
- vysoká úroveň programování v Pythonu předurčuje, že implementace demonstrační aplikace bude přehledná a tedy snadno pochopitelná jak pro budoucí rozšíření samotné aplikace, tak i pro implementaci rozšíření programového rozhraní, a
- obrovská komunita a nepřeberné množství balíčků pro Python zasahuje do jakéhokoliv odvětví Informatiky, ale hlavně v kontextu této práce nalezení kvalitního grafického enginu, pro který existuje obal pro Python, nebude nemožné.

Přestože použití jazyka Python má mnoho výhod, bylo nutné ještě vyřešit několik komplikací s tím spojených. Tyto problémy jsou detailněji popsány v následujících podsekcích.

Lincence jazyka Python je GNU GPL.

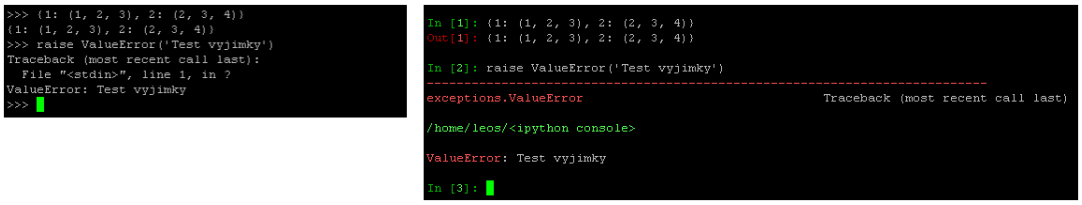
2.2.1 IPython

Samotný Python nabízí interaktivní interpret, který ale nenabízí vysoké pohodlí pro uživatele. Bylo tedy vhodnější použít rozšíření tohoto interpretu a tím je IPython. IPython nabízí

[2] Jedná se zpravidla o knihovny poskytující elementární grafické operace pro daný operační systém, které nelze prohlásit jako plnohodnotný grafický engine v kontextu, v jakém pojednává tato práce.

krom barevného zvýrazňování syntaxe také možnosti historie zadaných příkazů, doplňování textu pomocí klávesy *Tab*, formátování datových struktur a tracebacků, a další (viz. obr. 2.6).

Tento rozšiřující balík je volně dostupny pod licenci BSD.



Obrázek 2.6: Porovnání vzhledu implicitní interaktivní konzole Pythonu a konzole IPythonu

2.2.2 Python-OGRE

Důležitým krokem bylo zvolení vhodného grafického enginu pro jazyk Python. OGRE obsahuje mnoho skvělých vlastností, ale hlavně je obklopen komunitou, která vytvořila obal pro Python pomocí Boost.Python.

Lincence Python-Ogre je GNU LGPL.

2.2.3 PyGTK

Pro vytvoření demonstrační aplikace bylo zapotřebí použití jednoho z GUI. Pro Python je udržováno několik těchto knihoven, jako například Tk, Qt, WxWidgets, nebo GTK+. Zvolení GUI nehraje příliš velkou roli, bylo ale nutné ověřit, že je možné vložit výstup grafického enginu do jedné komponenty v hlavním okně aplikace. Bylo zvoleno GUI PyGTK.

Také bylo zapotřebí nalézt způsob vložení iterpretu IPython do komponenty v GTK+. K tomuto účelu je možné použít modul `ipython_view`, který spojuje vlastnoti IPythonu a GTK+ komponenety TextView.

Licence PyGTK je také GNU LGPL, naopak licence `ipython_view` modulu je BSD.

Kapitola 3

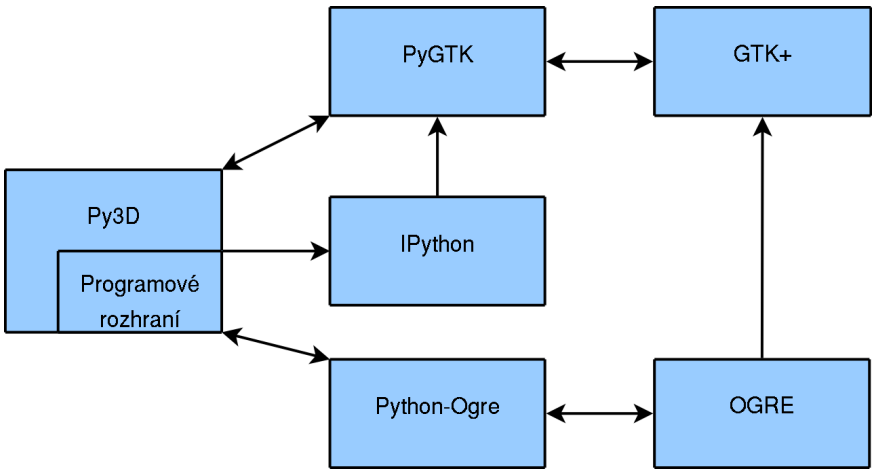
Implementace

Tato kapitola pojednává o samotné implementaci demonstrační aplikace. Výchází z implementační analýzy popsané v kapitole 2 a uvádí zde konkrétní výsledky návrhu, jako jsou datové struktury a algoritmy.

Po spuštění aplikace se nejprve zinicializuje uživatelské rozhraní, dojde k načtení strukury splash okna a jeho zobrazení. Poté se načte rozvržení hlavního okna a provede se jeho základní nastavení – vyplnění sloupců a skupin v seznamu objektů, spuštění a nastavení vzhledu konzoly pro ovládání a nápovědy, a také nastavení časovače pro pravidelné vykreslování renderované scény do komponenty. Jakmile je hlavní okno nastaveno, dochází k inicializaci rozhraní Python-Ogre. To se skládá z nastavení vykreslovacího okna do komponenty hlavního okna, načtení základních zdrojů[1], nastavení manažéru scény, vytvoření implicitní kamery, založení zobrazovacího záběru a vytvoření orientační mřížky ve scéně.

Po úspěšné inicializaci všech těchto kroků dochází k připojení programového rozhraní skriptovacího jazyka. Následně je pak splash okno zrušeno a zobrazí se již nastavené hlavní okno. Dále je pak řízení předáno hlavní smyčce PyGTK a aplikace je pouze řízena událostmi nástávající od akcí uživatele, které jsou řešeny zpětným voláním do aplikace.

Celá zmíněná funkčnost aplikace je zapouzdřena do instance třídy *Py3D* (viz. obr. 3.1).



Obrázek 3.1: Schéma hlavních kompoment a jejích vzájemná komunikace

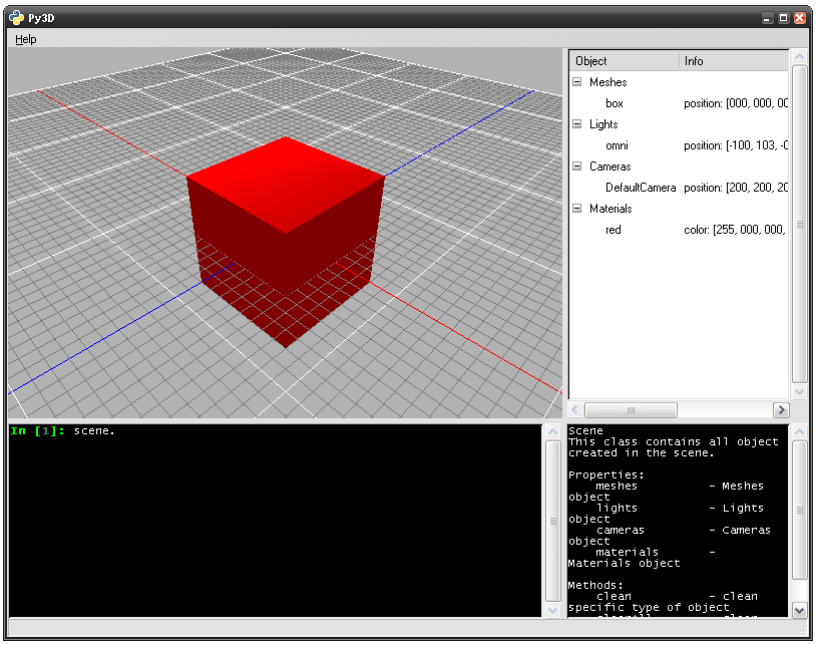
[1] Zde se jedná o zdroje pro renderovací systém, což zahrnuje textury, materialy, modely, atd.

3.1 Uživatelské rozhraní

Jak již bylo uvedeno, při implementaci bylo použito GUI rozhraní GTK+ s použitím obalu PyGTK určeném pro skriptovací jazyk Python. Rozvržení veškerých grafických prvků v oknech bylo provedeno pomocí nástroje Glade, který bývá součástí vývojového balíku GTK+.

Demonstrační aplikace obsahuje celkově tři okna – splash, about, a hlavní okno. Okna splash a about mají pouze informační charakter bez jakékoliv funkcionality, a proto v rámci této sekce již bude rozebíráno pouze hlavní okno aplikace. Toto okno bylo rozděleno na čtyři oblasti (viz. obr. 3.2):

- vykreslovací plocha,
- seznam objektů ve scéně,
- konzola skriptovacího jazyka, a
- nápověda k rozhraní.



Obrázek 3.2: Hlavní okno demonstrační aplikace

3.1.1 Vykreslovací plocha

Vykreslovací plocha je z pohledu GTK+ komponenta typu GtkDrawingArea, ale v inicializaci Python-Ogre je specifikován její handler, který ji jednoznačně určuje jako cílové zařízení, do kterého grafický engine bude vykreslovat vyrenderované snímky. Toto je velice důležité, neboť v této fázi dochází k jednostrannému spojení grafického engine se skriptovacím jazykem. Přesněji řečeno, grafický engine přes svoje knihovny vykresluje scénu do okna aplikace, která běží v interpretu skriptovacího jazyka.

Vykreslovací plocha zároveň zachytává události od myši. Držením levého tlačítka dochází k posunu pozice kamery, přičemž cíl zůstává na stejném místě. Tento posun se snaží zachovat stávající vzdálenost mezi pozicí kamery a cílem, což ve výsledku má efekt pohybu kamery po kouli. Držením prostředního tlačítka je pohyb pozice kamery a cíle totožný, dochází tedy k souběžnému posunu obou bodů. Držením pravého tlačítka je prováděn pohyb cíle kamery, podobně jak v prvním případně kopírující pomyslnou kouli. Rotováním kolečka myši dochází k přiblížení popř. oddálení kamery od cíle. Veskeré pohyby jsou prováděny v lokalních souřadnicových osách obou bodů.

3.1.2 Seznam objektů

Tato část je nemodifikovaná komponenta GTK+ typu GtkTreeView a její hlavní účel je dát uživateli přehled o objektech obsažených ve scéně. Všechny objekty jsou vždy rozřazeny do částí Meshes, Lights, Cameras, nebo Materials a vždy jsou určené svým jednoznačným názvem. K výpisu objektů se také připojuje jejich základní informace (viz. kapitola 4).

Řízení obsahu tohoto výpisu je řešeno pomocí zpětného volání z programového rozhraní a ve výjímečných případech i ze samotné aplikace. Komponenta reaguje na tři akce, popsané v následujících řádcích:

- vytvoření objektu – k této události dochází tehdy, když je pomocí programového rozhraní volaná funkce pro vytvoření objektu ve scéně (ať už je jedná o jakýkoliv typ),
- update objektu – událost, kdy došlo ke změně jakéhokoliv parametru objektu (transformace, linkování, nastavení materialu),
- zrušení objektu – pokud je v programovém rozhraní vyvolána funkce pro odstranění objektu ze scény, je vyvolaná tato událost.

Komponenta pak reaguje na tyto události patřičným způsobem, tedy přidáním, znovunačtením, a odebráním záznamu. Tato komponenta také uchovává odkazy na tyto objekty vytvořené pomocí programového rozhraní a to proto, aby při vybrání záznamu[2] byl tento objekt vyznačen ve scéně jasně ohraničujícím boxem.

3.1.3 Konzola

Konzola je pravděpodobně spolu s vykreslovací plochou nejdůležitejší částí hlavního okna. Po inicializaci se ve spodní části vytvoří komponenta IPythonView, která (což je důležité říci) běží ve stejném vlákně jako samotná aplikace. Tato komponenta je rozšířená verze GTK+ komponenety typu TextView, která vkládá a řídí interaktivní interpret IPython. Do lokálního jmenného prostoru této konzoly je pak při inicializaci hlavního okna přidána instance třídy programového rozhraní a také instance tříd všech rozšíření. Tuto problematiku dále podrobněji popisuje sekce 3.2.

Společně s programovým rozhraním je zde docíleno opačného spojení grafického enginu se skriptovacím jazykem, než je uvedeno v sekci 3.1.1. Konzola díky programovému rozhraní volá funkce ovládající grafický engine.

[2] Omezuje se pouze na typy Meshes, Lights.

3.1.4 Nápověda

Konzola popsána v sekci 3.1.3 vyvolává událost stisku klávesy na klavesnici, která je obsluhována aplikací. Při této události se vždy přečte aktuální stav příkazového řádku konzole, regulárním výrazem $[a \cdots zA \cdots Z0 \cdots 9.]^+$ je z pravé strany vybrán řetězec, s kterým se pak pracuje podle následujícího algoritmu:

1. rozděl řetězec podle znaku [.],
2. vezmi a odstraň z tohoto rozdělení nejlevější část a zkus ji převést na instanci[3],
3. pokud byl převod úspěšný a v rozdělení je ještě prvek, skoč na předchozí bod,
4. použij výsledek posledního převodu a přečti docstring[4] z instance a nahraď jím obsah komponenty Nápověda,
5. pokud se převod alespoň jednou nepovedl, vymaž obsah komponenty Nápověda.

Je patrné, že použití nápovědy v dokumentačních řetězcích bylo nejen výhodné z hlediska jednoduchosti implementace, ale hlavně proto, aby uživatel nebyl nucen používat demonstrační aplikaci s otevřenou nápovědou k programovému rozhraní a také ke skriptovacímu jazyku Python. Toto univerzální řešení dovoluje nejen zobrazovat nápovědu k programovému rozhraní, ale také obecně ke všem instancím, modulům a funkcím, které jsou dostupné ve jmenném prostoru konzole, a tudíž i pro instance případných rozšíření, ale i standardním instancím tříd jazyka Python.

3.2 Programové rozhraní

Programové rozhraní je v tomto případě jakákoliv instance, která se nachází ve jmenném prostoru konzole a dovoluje uživateli přes svoje interní struktury provádět specifický kód. Tato obecnost byla navržena proto, aby pokud možno ničím neomezovala uživatele a případné další rozšíření aplikace, ať už z pohledu vývojáře nové verze, nebo uživatele, který si naimplementuje rozšíření stávající funkčnosti.

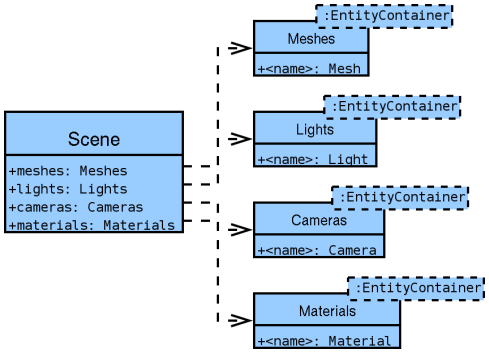
Demonstrační aplikace s sebou přináší základní ovládání scény v podobě předprogramovaného rozhraní, které má pevně definovanou cestu a je neoddělitelnou součástí aplikace. K jeho připojení do jmenného prostoru konzole dochází při inicializaci hlavního okna a instance je označena jako *scene*.

3.2.1 Scene

Třída *Scene*, jejíž instance je zmíněna na předcházejících řádcích, slouží v první řadě jako vstupní bod z uživatelského rozhraní do rozhraní programového, a tedy jako kontejner tříd *Meshes*, *Lights*, *Cameras* a *Materials*, ale také jako prostředník při zpětném volání mezi programovém rozhraní a samotné aplikace (viz. obr. 3.3).

Třída dále obsahuje instanci třídy, která se stará o samotný běh aplikace, pro snadný přístup k funkcím zpětného volání, a funkce `clean` a `cleanAll`. První z nich vymaže pouze specifikované typy objektů, druhá pak kompletně veškeré objekty ve scéně[5].

[3] Zde se využívá jmenný prostor konzole (viz. sekce 3.1.3).
[4] Jedná se o dokumentační řetězec podporující Python. Může být přiřazen k modulu, instanci a funkci.
[5] Výchozí kamera nemůže být odstraněna.



Obrázek 3.3: Třída *Scheme* a její obsah

3.2.2 EntityContainer

Třída *EntityContainer* implementuje základní funkčnost kontejneru, tedy přídávání a odstraňování obsahu, a hlavně objektový přístup k obsahu kontejneru. Tato třída je pak děděna a rozšířena pro konktrétní případy kontejnerů.

3.2.3 Movable, MovableWithTarget a Linkable

Třída *Movable* implementuje základní operace s pohyblivým objektem ve sceně. Jedná se o zjištění a nastavení pozice, zapnutí a vypnutí zobrazení, a standardní informaci, která se používá při zobrazení v seznamu objektů (viz. sekce 3.1.2). Tato třída je pak děděna třídami *Mesh* a *Light*.

Třída *MovableWithTarget* rozšiřuje třídu *Movable* o funkce pracující s objektovým cílem. Předefinovává také objektovu informaci pro seznam objektů. Tato třída je pak děděna třídami *Spotlight* a *Camera*.

Třída *Linkable* implementuje základní operace pro umožnění spojení dvou i více objektů ve sceně. Objekty mohou tvořit klasickou stromovou strukturu, kde manipulace s rodičem ovlivňuje také jeho děti. Třída dovoluje vytvoření a zrušení spojení s rodičem, a zjistit rodiče objektu. Tato třída je pak děděna třídami *Mesh*, *Light* a *Camera*.

3.2.4 Meshes

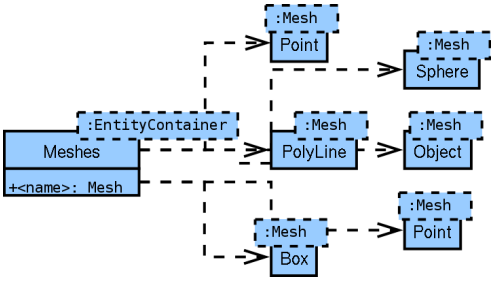
Třída *Meshes* dědí třídu *EntityContainer* a rozšiřuje ji o funkce pro vytváření základních primitiv ve scéně, jako jsou body, navazující linky, boxy, koule, plochy, a objekty v podporovaném formátu `.mesh` (viz. obr. 3.4).

Mesh

Třída *Mesh* dědí třídy *GenericEntity*, *Movable* a *Linkable*, a implementuje transformace rotování a zvětšování, dále pak přiřazení materialu. Rozšiřuje také informace o objektu.

Point, PolyLine, Box, Sphere, Plane, a Object

Tyto třídy dědí třídu *Mesh* a typicky jen implementují vytvoření objektu ve scéně pomocí Python-Ogre funkcí. Třídy drží instance označované jako *entity* a *node*, které jsou používány grafickým enginem. V prvním připadě se jedná o reprezentaci objektu ve scéně, v druhém



Obrázek 3.4: Třída *Meshes* a její obsah

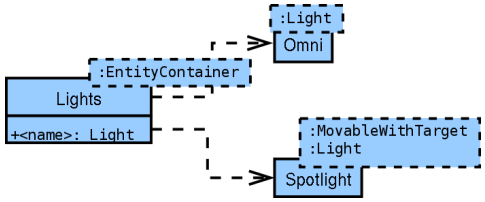
pak uzel ve scénovém grafu. Tyto atributy jsou dostupné přímo z programového rozhraní a je tedy možné bez jakéhokoliv rozšiřování aplikace, přistupovat k funkcím Python-Ogre nad daným objektem přes programové rozhraní. Toto přináší další volnost pro uživatele, ale je nutné mít na paměti, že při tomto způsobu může dojít k desynchronizaci dat mezi grafickým enginem a programovým rozhraním.

3.2.5 Lights

Třída *Lights* dědí třídu *EntityContainer* a rozšiřuje ji o funkce pro vytváření světelných zdrojů ve scéně, konkrétně omni a spotlight (viz. obr. 3.5).

Light

Třída *Light* dědí třídy *GenericEntity*, *Movable* a *Linkable*, a implementuje nastavení barvy světla.



Obrázek 3.5: Třída *Lights* a její obsah

Omni a Spotlight

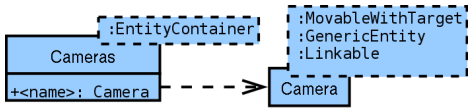
Tyto třídy dědí třídu *Light*, pouze *Spotlight* navíc z očividných důvodu jestě třídu *MovableWithTarget*. Podobně jako v sekci 3.2.4 je i zde implementace vytváření samotných objektů, stejnětak použití atributů *entity* a *node*. U *Spotlight* je ještě navíc atribut *targetNode*.

3.2.6 Cameras

Třída *Cameras* dědí třídu *EntityContainer* a rozšiřuje ji o funkci pro vytvoření kamery ve scéně (viz. obr. 3.6).

Camera

Třída *Camera* dědí třídy *GenericEntity*, *MovableWithTarget* a *Linkable*, a implementuje vytvoření a zrušení objektu typu kamera a funkci `activate`, která tuto kameru nastaví v aplikaci jako aktivní (z aktuální kamery se přepne zobrazení na právě aktivovanou).



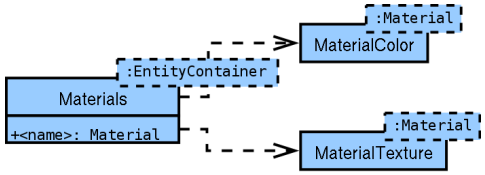
Obrázek 3.6: Třída *Cameras* a její obsah

3.2.7 Materials

Třída *Materials* dědí třídu *EntityContainer* a rozšiřuje ji o funkce pro vytváření materiálů ve scéně, konkrétně konstantní barvy a textury (viz. obr. 3.7).

Material

Třída *Material* dědí třídu *GenericEntity* a pouze rozšiřuje informaci o objektu.



Obrázek 3.7: Třída *Materials* a její obsah

MaterialColor a MaterialTexture

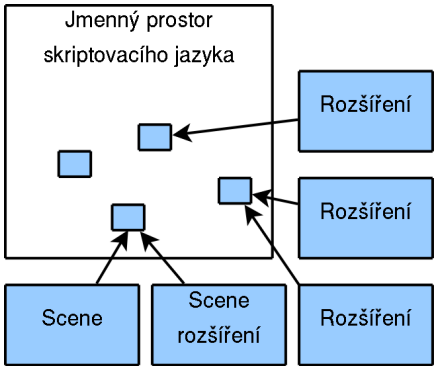
Tyto třídy dědí třídu *Material* a implementují vytvoření materiálu ve scéně a v případě *MaterialTexture* správné načtení textury do zdrojů grafického engine. Na rozdíl od předchozích tříd neobsahují atributy *entity* a *node*, ale pouze atribut *material*, který opět využívá grafický engine.

3.3 Rozšíření programového rozhraní

Kromě základního pevně definovaného programového rozhraní byla implementována možnost připojit do jmenného prostoru konzoly další možné vstupní body rozšíření tohoto rozhraní. Nejenže je možné implementovat rozšiřující vlastnost, která v základní verzi chybí, je ale i možné předefinovat již stávající rozhraní a to včetně původní verze *scene* (viz. obr. 3.8).

V průběhu inicializace hlavního okna dochází k prohledávání adresáře `extensions` a veškeré moduly Pythonu jsou importovány do běžící aplikace. Pokud je v modulu obsažena funkce `register`, dochází k jejímu zavolání. Funkce vrací jméno proměnné, pod kterou bude identifikovatelná ve jmenném prostoru konzole a instanci objektu, který bude pod tímto

jménem figurovat. Pokud modul neobsahuje funci `register`, je modul odstraněn z naimportovaných modulů a dále se za běhu aplikace nepoužije.



Obrázek 3.8: Princip mapování rozšíření do jmenného prostoru konzole

Kapitola 4

Demonstrační aplikace

Tato kapitola obsahuje informace o demonstrační aplikaci, její ovládání a možnosti rozšíření o další funkce, ale také podklady potřebné k sestavení a instalaci demonstrační aplikace.

4.1 Aplikace Py3D

Demonstrační aplikace byla pracovně nazvána *Py3D*, důvodem je evokování dojmu spojení skriptovacího jazyka Python s 3D grafickým enginem, což je jejím hlavním účelem.

4.2 Sestavení

Přestože, jak již bylo zmíněno, je demonstrační aplikace napsána ve skriptovacím jazyce Python a tudíž je možné ji spouštět přímo ze zdrojových souborů, byla přidána možnost vytvoření instalačního balíčku, který je poté nezávislý na ostatních komponentách. Tento způsob je velice vhodný v těch případech, kdy nechceme uživatele nutit k instalaci mnoha balíčků, které mohou být potřebné pouze ke spouštění samotné demonstrační aplikace a za jiných okolností je uživatel nevyužije.

Samotné sestavení je podporováno pouze pro systémy Microsoft Windows a vyžaduje následující komponenty třetích stran:

- Python, verze 2.5,
- Python-Ogre, verze 1.1,
- GTK runtime, verze 2.10,
- PyGTK, verze 2.10,
- PyCairo, verze 1.2,
- PyGObject, verze 2.12,
- IPython, verze 0.8,
- py2exe, verze 0.6, a
- Inno Setup, veze 5.2.

Všechny tyto závislosti jsou dostupné v adresáři `_dependeces`, který je obsažen v cestě zdrojových souborů.

4.2.1 Postup

Tato podsekce popisuje postup sestavení, kde tížený výsledek je samoinstalační balíček demonstrační aplikace pro Microsoft Windows. Celá procedura lze popsat v následujících krocích:

1. instalace závislostí popsané v sekci 4.2,
2. vytvoření kopie adresáře se zdrojovými soubory do zapisovatelné části disku[1],
3. změna aktuálního adresáře do kořene zdrojových souborů,
4. spuštění příkazu:

   ```
   setup.py py2exe
   ```

5. spuštění aplikace Inno Setup a načtění `setup.iss` – konfiguračního souboru, který je umístěn v kořenu zdrojových souborů, a
6. vyvolání přikazu *Build – Compile* z hlavního menu aplikace Inno Setup.

Tento postup nejprve vytvoří adresar `build`, kde vloží zkompilované moduly Pythonu, které demonstrační aplikace používá. Následně pak do adresáře `dist` vytvoří spouštěcí soubor `py3d.exe` a všechny dynamicky linkované knihovny. Zkompilované moduly Pythonu se pak zabalí do archívu `library.zip`. Na závěr se nakopírují datové soubory specifikované v `setup.py`, které jsou součástí aplikace a jsou vyžadované pro její bezproblémový běh. Je nutné říci, že ve své podstatě nedochází ke kompilaci modulů Pythonu do spustitelného souboru, naopak jsou moduly pouze zkompilovány do bytekódu stejně tak, jako při běžném spouštění interpretrem přímo ze zdrojových souborů. Lze tedy očekávat, že tento postup nepřinese žádné výkonnostní zrychlení aplikace.

Pro zabalení zkopilované aplikace do instalačního balíčku se použíje software Inno Setup, jehož výsledkem je pak soubor `Py3D_X.X.exe`[2], který figuruje jako instalační průvodce, který je typický pro tento operační systém.

4.3 Instalace

4.3.1 Hardwarové požadavky

Požadavky na konfiguraci počítače:

- běžný osobní počítač dnešní doby,
- Microsoft Windows XP s instalací DirectX,
- přibližně 100 MB volného místa na disku, a
- akcelerovanou grafickou kartu kompatibilní s DirectX verze 9.

[1] Předpokládá se, že zdrojové soubory jsou umístěny na CD, které je přiloženo k této zprávě.
[2] Konstrukce X.X zastupuje verzi demonstrační aplikace.

4.3.2 Instalace aplikace

Při použití postupu uvedeného v sekci 4.2.1 je instalace demonstrační aplikace omezena pouze na spuštění instalačního balíku a následování průvodce v něm obsaženého. Spouštění samotné aplikace se pak prování pomocí zástupce v nabídce *Start*, nebo pomocí spustitelného souboru `py3d.exe` v cílovém adresáři instalace.

Pokud ale bude požadavek spouštět demonstrační aplikaci přímo ze zdrojových souborů, je nutné nejprve nainstalovat veškeré závislosti uvedené v sekci 4.2. Aplikace se pak spouští pomocí skriptu `py3d.py`, který je prováděn nainstalovaným interpretem Pythonu.

4.3.3 Odinstalování aplikace

Odinstalování lze provést pomocí nástojů systému pro odebírání programů (*Ovládací panely – Přidat nebo odebrat programy*) nebo lze užít i zástupce pro odinstalování.

4.4 Ovládání

Tato sekce obsahuje popis ovládání jak samotného uživatelského rozhraní, tak i rozhraní programového.

4.4.1 Uživatelské rozhraní

Ovládání uživatelského rozhraní se díky jednoduchosti aplikace omezuje pouze na ovládání hlavního okna. Ovládání vykreslovací plochy bylo podrobně uvedeno v sekci 3.1.1, stejně tak ovládání seznamu objektů v sekci 3.1.2.

4.4.2 Programové rozhraní

Tato podsekce popisuje programové rozhraní, které je součástí implementace demonstrační aplikace, a není dále předefinováno žádnou z rozšíření. Tato sekce předpokládá, že čtenář ovládá principy objektově orientovaného programování a je seznámen se syntaxí jazyka Python.

Do lokalního jmenného prostoru konzole se po zinicializování grafického rozhraní vloží dvě třídy *vector* a *color*, jedna instance *scene*, a jeden modul *ogre*.

Třída vector

Třída *vector* slouží k definici vektoru v trojrozměrném prostoru scény. Tato třída slouží zpravidla k zadávání vektoru při manipulaci s objektem ve scéně. Tato třída je pouze alias na interní třídu vektoru grafického engine z důvodu jednoduchosti při zadávaní hodnot v konzole.

Třída color

Třída *color* slouží k definici barvy u jednobarevném materialu a podobně jako *vector* je pouze aliasem na odpovídající třídu v grafickém engine.

Instance scene

Jak bylo podrobněji vysvětleno v sekci 3.2, obsahuje jmenný prostor konzole instanci třídy *Scene*, která implementuje veškeré standarní programové rozhraní demonstrační aplikace.

Instance *scene* obsahuje:

- instance:
  - *meshes* – sdružuje objekty typu Mesh,
  - *lights* – sdružuje objekty typu Light,
  - *cameras* – sdružuje objekty typu Camera,
  - *materials* – sdružuje objekty typu Material,
- funkce:
  - `clean(meshes, lights, cameras, material)` – smaže specifikovaný (nastavením příslušné hodnoty na `True`) typ objektů,
  - `cleanAll()` – smaže všechny objekty ze scény kromě implicitní kamery.

Instance *meshes* obsahuje:

- instance názvu dle zadání – konkrétní typy pro bod, spojená čára, box, koule, plocha a načtený objekt,
- funkce:
  - `createPoint(identifier, material, position)` – vytvoří bod identifikovaný parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`,
  - `createPolyLine(identifier, material, points, position)` – vytvoří spojené čáry identifikované parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`. Body určené polem `points` jsou orientovány vzhledem k pozici objektu,
  - `createBox(identifier, material, position)` – vytvoří box $100 \times 100 \times 100$ bodů identifikovaný parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`,
  - `createSphere(identifier, material, position)` – vytvoří kouli poloměru 100 bodů identifikovanou parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`,
  - `createPlane(identifier, material, position)` – vytvoří plochu 100×100 bodů identifikovanou parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`,
  - `createObject(identifier, filename, material, position)` – vytvoří objekt načtením ze souboru `filename` identifikovaný parametrem `identifier`, s nastaveným materiálem `material` na pozici `position`.

Instance *lights* obsahuje:

- instance názvu dle zadání – konkrétní typy pro omni a spotlight,
- funkce:
  - `createOmni(identifier, position)` – vytvoří světelný bod identifikovaný parametrem `identifier`, na pozici `position`,
  - `createSpotlight(identifier, position, target)` – vytvoří světelný kužel identifikovaný parametrem `identifier`, na pozici `position`, svítící do bodu `target`.

Instance *cameras* obsahuje:

- instance názvu dle zadání – třídy *Camera*,
- funkce:
  - `createCamera(identifier, position, target)` – vytvoří kameru identifikovanou parametrem `identifier`, na pozici `position`, natočenou do bodu `target`.

Instance *materials* obsahuje:

- instance názvu dle zadání – konkrétní typy pro barvu a texturu,
- funkce:
  - `createColor(identifier, color)` – vytvoří materiál identifikovaný parametrem `identifier`, s nastavenou barvou `color`,
  - `createTexture(identifier, filename)` – vytvoří materiál identifikovaný parametrem `identifier`, s nastavenou texturou `texture`.

Instance tříd dědící *Movable* obsahuje funkce:

- `disable()` – skryje objekt ve scéně,
- `enable()` – zobrazí object ve scéně,
- `getPosition()` – vrátí pozici objektu v prostoru,
- `setPosition(position)` – nastaví pozici `position` objektu v prostoru.

Instance tříd dědící *MovableWithTarget* obsahuje funkce:

- `getTargetPosition()` – vrátí pozici cíle objektu v prostoru,
- `setTargetPosition(position)` – nastaví pozici `position` cíle objektu v prostoru.

Instance tříd dědící *Linkable* obsahuje funkce:

- `getParent()` – vrátí rodiče objektu nebo `None`,

- `link(parent)` – vytvoří spojení s rodičem `parent`,
- `unlink()` – zruší spojení s rodičem.

Instance tříd dědící *Mesh* obsahuje funkce:

- `getMaterial()` – vrátí material objektu,
- `setMaterial(material)` – nastaví objektu materiál `material`,
- `getRotation()` – vrátí rotaci objektu v prostoru,
- `setRotation(orientation)` – změní rotaci objektu podle `orientation`,
- `getScale()` – vrátí zvětšení objektu,
- `setScale(dimention)` – změní zvětšení objektu podle `dimention`.

Instance tříd dědící *Light* obsahuje funkce:

- `setLightColor(color)` – nastaví barvu světla na `color`.

Instance třídy *Camera* obsahuje funkce:

- `activate()` – nastaví kameru jako aktivní (vykreslování bude probíhat z nově nastavené kamery).

Poznámka:
Funkce `setPosition`, `setRotation`, a `setScale` mají ještě variantu `setPositionRelative`, `setRotationRelative`, a `setScaleRelative`, kde dochází k relativní transformaci.

Modul ogre

Modul ogre je vložen do jmenného prostoru konzole proto, aby umožnil uživateli další prostor pro experimenty na nižší úrovni, přesněji řečeno na úrovni obalu grafického engine.

4.5 Rozšíření

Rozšiřující moduly se ukládájí do adresáře `extentions` a musí obsahovat metodu `register(application)`, která vrací dvojici jméno, instance, kde jméno určuje název ve jmenném prostoru konzole a instance je pak hodnota figurující za tímto názvem. Na implementaci rozšiřujícího modulu se jiné omezení neklade.

Kapitola 5

Závěr

Cílem této práce bylo navrhnout a realizovat řešení pro spojení grafického enginu společně se skriptovacím jazykem. Přestože bylo možné zpracovat tento úkol mnoha způsoby, byla vybrána nejen jedna z nejvhodnějších kombinací nástrojů, ale také návrh a implementace docílila toho, že výsledná demonstrační aplikace je snadno ovladatelná a rozšířitelná bez jakéhokoliv zásahu do kódu aplikace.

Zde je také vhodné místo k uvedení faktu, že autor bude i po skončení této práce na myšlence propojení grafického engine a skriptovacího jazyka dále pracovat a výslednou aplikaci rozšiřovat. Cílem bude také poskytnout tuto aplikaci a její zdrojový kód široké veřejnosti a to jako open source.

5.1 Návrh pokračování projektu

V první řadě by bylo vhodné, přestože to nebylo hlavní náplní této práce, vylepšit uživatelské rozhraní, tedy hlavní okno demonstrační aplikace. Nabízí se hned několik možných úprav:

- globální nastavení aplikace – barvy písem a pozadí, velikosti okna a jednotlivých podčástí, atd.,
- větší interakce s vykreslovací plochou,
- vylepšení systému informací o objektu,
- nápověda programového rozhraní podporující formátování, a
- systém řízení animací, a mnoho dalších.

Mnohem důležitější část, ve které by bylo možné pokračovat, je programové rozhraní. Podle specifikace byly implementovány pouze základní operace nad objekty ve scéně a to dovoluje poměrně široký prostor pro další vylepšení:

- dokončení implementace veškerých možných transformací objektů podporujících přímo grafickým enginem,
- přidání implementace nejpoužívanějších transformací objektů,
- možnost vkládat složitější objekty do scény,

- plně dokončit poměrně komplexní system materiálů – multipass, shadery, volumetric, video, atd.,
- podpora animací a kosterní animace,
- speciální post-renderové efekty,
- podpora formátu třetích stran, a další.

I přesto, že použití skriptovacího jazyka byla navržena důkladně a s ohledem na další rozšiřování bez toho, aniž by se muselo měnit jadro aplikace, je možné uvést několik možných pokračování v tomto směru:

- konzola a tedy i programové rozhraní může bežet v separátním vlákně,
- načítání rozšíření by mohlo být kontrolovatelné a toto nastavení by mohlo být také ukládáno pro příští běh aplikace,
- vytvoření třídy pro zpřehlednění systému zpětného volání z a do programového rozhraní, a
- vytvoření toolkitu pro další rozšíření.

Literatura

[1] DevMaster.net. 3d game and graphics engines database, 2008.

[2] Ph.D. Ing. Přemysl Kršek. Základy počítačové grafiky, 2006.

[3] Wikipedie. *Skript (program) — Wikipedie: Otevřená encyklopedie.* 2007.

[4] Wikipedie. *JavaScript — Wikipedie: Otevřená encyklopedie.* 2008.

[5] Wikipedie. Perl — wikipedie: Otevřená encyklopedie, 2008.

[6] Wikipedie. *PHP — Wikipedie: Otevřená encyklopedie.* 2008.

[7] Wikipedie. Python — wikipedie: Otevřená encyklopedie, 2008.

[8] Wikipedie. *Ruby — Wikipedie: Otevřená encyklopedie.* 2008.