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

GRAFICKÁ APLIKACE V PYTHONU S POUŽITÍM OPENGL

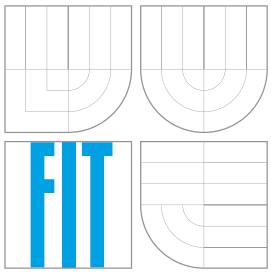
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ISTVÁN SZENTANDRÁSI

BRNO 2009

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

GRAFICKÁ APLIKACE V PYTHONU S POUŽITÍM OPENGL

GRAPHICS APPLICATION IN PYTHON USING OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ISTVÁN SZENTANDRÁSI

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN NAVRÁTIL

BRNO 2009

Abstrakt

Cílem této práce bylo prozkoumání možností použití programovacího jazyka Python v počítačové grafice a porovnat výhody a nevýhody interpretovaného přístupu oproti klasickému programování. Byla vytvořená demonstrační aplikace použitím nadstavby jazyka Python, PyOpenGL. Tato aplikace byla testována spolu s implementací stejné aplikace pomocí OpenGL C API. Výsledky testování ukazují, že Python byl průměrně dvakrát pomalejší a využíval mnohem víc systémových zdrojů. Kvůli těmto skutečnostem Python není vhodný na použití v profesionálních aplikacích. Na druhé straně však má rozšířenou standardní knihovnu, užitečné specializované knihovny, jednodužší syntax. Python v kombinaci s PyOpenGL je proto ideální pro vzdělávací účely.

Abstract

The aim of this work was to study the possibilities of the Python programming language in computer graphics and to determine its competence in this field. To achieve this the cross platform Python binding to OpenGL and related APIs, PyOpenGL was used to create a demo, and compare it in many ways to an implementation of the demo using the standard OpenGL C API. As the result Python was found in average twice as slow as the alternative C demo and using much more CPU load. Because of this fact PyOpenGL is not advisable in professional applications. On the other hand Python has an extensive standard library, very useful specialized libraries, simpler syntax, which makes it ideal to use in education.

Klíčová slova

Python, PyOpenGL, OpenGL, porovnání rychlosti, testování, částicový systém, profilování, optimalizace, mapování stínů, odlesky, stencil buffering

Keywords

Python, PyOpenGL, OpenGL, comparison in speed, testing, particle system, profiling, optimalization, shadow mapping, reflection, stencil buffering

Citace

István Szentandrási: Grafická aplikace v Pythonu s použitím OpenGL, bakalářská práce, Brno, FIT VUT v Brně, 2009

Grafická aplikace v Pythonu s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Navrátila

.....
István Szentandrási
20. května 2009

Poděkování

Chtěl bych poděkovať mému vedoucímu, Ing. Janovi Navrátilovi za jeho připomínky a cenné rady, poskytnutou odborní pomoc a trpělivost.

Obsah

Zoznam obrázkov

[0] **Poznámka:**
Obrázky a grafy sú mojim vlastným dielom.

Kapitola 1

Úvod

Programovací jazyk python bol vytvorený v raných deväťdesiatych rokoch Guido van Rossumom v Centrum Wiskunde and Informatica (CWI, viď http://www.cwi.nl/) v Holandsku ako následník jazyku ABC. Od tej doby sa stal jedným z najrozšírenejších interpretovaných jazykov. Používa sa u webových aplikácií na strane serverov, grafických aplikáciách na desktopoch, na vedecké účely, ale aj v hrách a 3D grafike. Ako príklad je možné uviesť CAD aplikácie, či Blender 3D. Python má okrem rozšírenej štandardnej knižnice veľké množstvo špecializovaných knižníc, napríklad knižnice pre 3D aplikácie - PyGame, PyGlet, PyOpenGL atd. (celý zoznam viď http://www.vrplumber.com/py3d.py). Väčšina týchto knižníc je založená na OpenGL API.

OpenGL (Open Graphics Library) je priemyselný štandard špecifikujúci viacplatformové rozhranie (API) k akcelerovaným grafickým kartám. Prvú verziu vydal Silicon Graphics Inc. v roku 1992. Posledná verzia 3.1 bola vydaná 24. marca 2009, ktorá prináša zásadné zmeny oproti verzii 2.1, ktorá je najviac rozšírená v aplikáciách a bola použitá aj v tejto práci.

Existuje viac nadstavieb Pythonu, ktoré poskytujú OpenGL API. Známejšie sú PyOpenGL a PyGlet. PyOpenGL poskytuje jedna ku jednej funkcie OpenGL API. PyGlet je objektovo orientovaný a podporuje správu okien, prácu s hudbou, videom, či spracovanie udalostí, ktoré by mohli mať vplyv na výkon pri testovaní. Z týchto dôvodov je vhodnejší PyOpenGL na porovnanie rýchlosti medzi implementáciami v iných jazykoch.

Cieľom tejto práce bolo vytvoriť demonštračnú aplikáciu v PyOpenGL, a porovnať výsledok s implementáciou v jazyku C. Python je interpretovaný jazyk, a preto bude pravdepodobne pomalší kvôli réžii. Mojou úlohou bolo zistiť pomer spomalení, a naviac, že kde k nim dochádza, a porovnať výhody a nevýhody interpretovaného prístupu oproti klasickému programovaní.

Kapitola 2

PyOpenGL a použité technológie

PyOpenGL je viacplatformová python väzba k OpenGL a príbuzným API. Používa modul ctypes zo štandardnej knižnice. PyOpenGL v čase písania tejto práce podporuje OpenGL od verzie 1.1 po 3.0, GLU, GLUT 3.7(FreeGLUT) a GLE3. Podporuje aj veľké množstvo OpenGL rozšírení. Medzi nepovinné závislosti patria knižnice, napríklad PIL (Python Imaging Library), či Numpy. PyOpenGL je možné použiť s veľkým množstvom GUI knižnicami pre python: wxPython, FxPy, PyGame či Qt. Ďalším možným riešením na správu okien, spracovanie udalostí atď. je použiť zabudovaný GLUT rozhranie.

2.1 Rozdiely od C OpenGL API

Väčšina funkcií, ktoré sa objavia v PyOpenGL 3.x je identická vo volaní a vo funkcionalite ako alternatívy v C špecifikácii. Existujú však z toho aj výnimky zapríčinené rozdielmi medzi jazykmi C a Python. Najčastejší dôvod týchto výnimiek je daná rozličným spôsobom pracovania s polami v C a Python. Napríklad funkcia v C:

```
void foo(int count, const int* args);
```

bude v PyOpenGL:

```
foo(args) -> None
```

Ďalej C funkcie, ktoré modifikujú obsah polí:

```
void bar(int args[4]);
```

bude:

```
bar() -> args
```

PyOpenGL všeobecne používa striktné OpenGL operácie – chyby vyvolajú výnimky namiesto spoliehania na programátora, aby manuálne kontroloval hodnotu vrátenú funkciou `glCheckError` . Táto funkcionalita je veľmi užitočná pri vývoji grafických aplikácií. Vypnutie spôsobuje obrovské zvýšenie výkonnosti, a preto vo výslednom programe je vhodné vypnúť pomocou:

```
import OpenGL
OpenGL.ERROR_CHECKING = False
```

PyOpenGL štandardne loguje všetky chyby pomocou logovacieho modulu Pythonu. Vypnutie môže tiež znamenať zlepšenie vo výkone (`OpenGL.ERROR_LOGGING = False`).

PyOpenGL 3.x poskytuje široký výber dátových typov, ktoré je možne použiť vo funkciách, ktoré vyžadujú pole ako argument:

- numpy pole
- numeric pole
- numarray pole
- reťazce
- čísla (ako ukazovatele na jednoprvkové pole)
- ctypes pole
- ctypes parametre
- ctypes ukazovatele
- Python zoznamy – `tuple, list, string`
- VBO (vertex buffer objects)

Pythonovské zoznamy, tuples a reťazce vyžadujú vytvorenie dočasných štruktúr na udržanie dát, keďže neobsahujú kompatibilnú kópiu dát, ktorá by mohla byť predaná funkciám nízkoúrovňového OpenGL API. Takže tieto dátové typy nie sú vhodné pri operáciách, ktoré sú kritické na výkon.

PyOpenGL podporuje väčšinu OpenGL rozšírení. Rozšírenia sú dostupné ako normálne ukazovatele na funkcie po importovaní príslušného balíka. Napríklad:

```
from OpenGL.GL.ARB.vertex_buffer_object import *
buffer = glGenBufferARB(1)
```

Žiadna inicializácia nie je potrebná, keďže sa o to postará PyOpenGL sám. V prípade potreby je možné zavolať inicializačnú funkciu, ktorá vráti len bool hodnotu podľa toho, či platforma podporuje dané rozšírenie.

```
if  glInitVertexBufferObjectARB():
    ...
```

Jednoduchší spôsob je otestovať logickú hodnotu požadovanej funkcie, pred jeho prvým použitím.

```
if (glGenBuffersARB):
   buffers = glGenBuffersARB(1)
```

Kompletný zoznam zmien včetne aliasov funkcií je popísaný na stránkach PyOpenGL. ([4])

2.2 Pomocné knižnice - PIL, NumPy

Python má veľmi užitočné špecializované knižnice, ktoré uľahčujú a urýchľujú prácu napríklad pri načítaní obrázkov (PIL), alebo pri práci s maticami či vektormi (NumPy).

PIL (Python Imaging Library) pridáva schopnosť spracovania obrázkov k interpretu Pythonu. Podporuje rozsiahle množstvo typov formátu obrázkových súborov a účinné spracovanie obrázkov. PIL umožňuje načítanie a konvertovanie medzi formátmi, tisk obrázkov,

operácie nad obrázkom: filtrovanie farby, zmenu veľkosti obrázkov, rotáciu a iné transformácie. PIL umožní aj získať štatistiky o obrázku. V tejto práci som využil hlavne schopnosť načítania `.tga` obrázkov, ich rotáciu a schopnosť reprezentovania obrázkov ako reťazec.

NumPy je fundamentálna knižnica potrebná pre vedecké účely v Pythonu. Z pohľadu práce z NumPy je dôležité pokročilá a hlavne rýchla práca s N dimenzionálnymi maticami a vektormi: násobenie a transponent matíc atď.

2.3 Použité technológie

Pri návrhu aplikácie som vybral technológie, ktoré jednak pridajú trocha reálnosti k scéne (tiene, zrkadlový efekt) a aj vyzdvihujú výhody či nevýhody PyOpenGL.

Časticové systémy

Časticový systém je technika, ktorou je možno simulovať náhodné fyzikálne efekty, napríklad oheň, vietor, vodné hladiny, explózie, dážď, sneženie atď. Časticový systém je vlastne kolekcia častíc modelujúca nejaký objekt. Pre každú snímku animácie sú uskutočnené nasledujúce kroky:

1. sú vytvorené nové častice
2. každá nová častica je inicializovaná
3. častice, ktorým končí životnosť, sú odstránené
4. zostávajúce častice sú posunuté a transformované podľa simulovaného efektu
5. snímka je vykreslená

Časticové systémy sú zvyčajne trojdimenzionálne, ale v niektorých prípadoch postačia aj dvojdimenzionálne. ([9])

Odrazy pomocou stencil bufferu

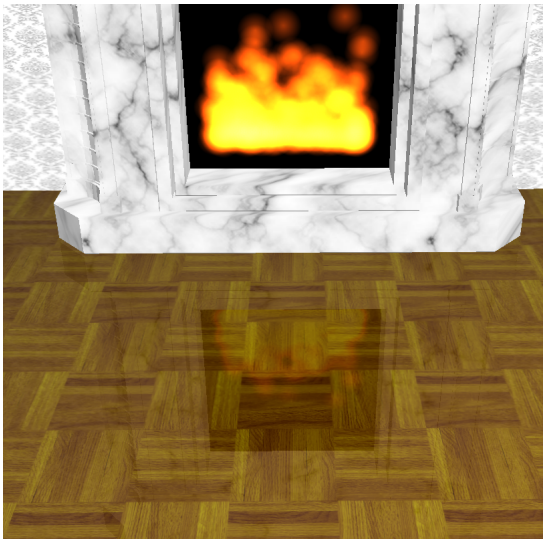
Stencil buffer slúži na obmedzenie výkresu na zvolené časti scény ([8]). Túto vlastnosť je možno použiť na jednoduchý dynamický odrazový efekt. V stencil bufferu je uložená pozícia odrazovej plochy, ktorá potom slúži ako filter pri vykresľovaní odrazov. Nevýhodou je, že objekty, ktoré majú odraz, musia byť vykreslené dvakrát – raz za odrazovou rovinou pomocou stencil testu a raz pred ňou.

Mapovanie tieňov

Existuje viac možností na renderovanie tieňov. Najznámejšie sú ([3]):

Plošné tiene Najjednoduchšia metóda, kde tiene sú mapované na rovinu použitím maticových operácií. Výsledkom sú veľmi jednoduché tiene. Prakticky sú použiteľné na tzv. "drop shadows".

Tieňové telesá Rozšírená metóda, ktorá je však výpočtovo náročná, a preto tento spôsob by bol v Pythonu rádovo pomalší ako v C. V skratke metóda: pre každý objekt sa vypočíta silueta, pomocou ktorých sú vytvorené tieňové telesá, ktoré potom sú uložené do vertex bufferu. Pomocou týchto tieňových telies a stencil bufferu sú vykreslené časti scény, ktoré sú ovplyvnené daným svetlom.



Obr. 2.1: Odrazy pomocou stencil bufferu

Mapovanie tieňov Komplikovanejšia metóda, ale menej náročná na procesor, ako predchádzajúca.

Mapovanie tieňov pomocou textúr prvý krát predstavil Lance Williams v roku 1978 v štúdii názvom „Casting curved shadows on curved surfaces“. Od tej doby sa rozšírilo aj v offline renderovaní a aj v realtime grafike. Bol napríklad použitý v animovanom filme „Toy Story“. ([13])

Výhody

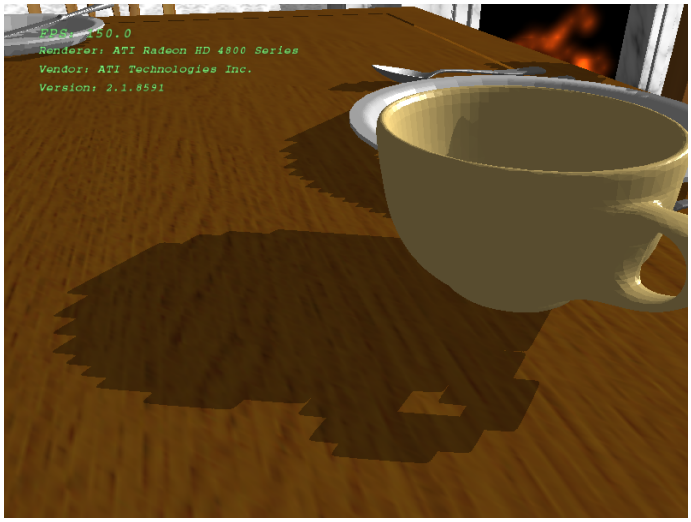
- Nie je potrebná žiadna znalosť o scéne.
- Stačí jedna textúra pre každé svetlo.
- Vyhýba sa veľkému množstvu dát potrebných u tieňových telesách.

Nevýhody

- aliasing
- náročnejší na grafický hardware

Konceptuálne táto metóda je dvojfázová. V prvej fáze je scéna vykreslená z pozície svetla. Výsledkom tejto fázy je “mapa hĺbky” – 2D mapa, ktorá určí najbližší pixel od zdroja svetla. V druhej fáze scéna je renderovaná z pohľadu kamery. Pre každý rastrovaný fragment sa vypočíta relatívna XYZ pozícia od svetla. Táto pozícia by mala byť nastavená, aby zodpovedala frustu používanému pri vytvorení “mapy hĺbky”. V ďalšom kroku porovnáme hodnotu hĺbky na pozícii XY v “mape hĺbky”(ďalej hodnota A) k hodnote Z z relatívnej pozície od zdroja svetla (ďalej hodnota B). Ak B je väčšie ako A, potom medzi svetlom a zodpovedajúcim bodom musí ešte existovať fragment či objekt, takže daný bod je v tieni. Ak B sa rovná približne A, tak fragment je osvetlený. (viď [14], [7])

V tejto práci som použil štyri prechody.



Obr. 2.2: Mapované tiene - Aliasing

Prvý prechod zodpovedá prvej fáze konceptuálnej metódy. Celá scéna je vykreslená z pozície svetla, a hodnoty hĺbky sú uložené do textúry. Pre tento cieľ sa dá použiť rozšírenie OpenGL `ARB_depth_texture`.

Druhú fázu použitej metódy som kvôli prehľadnosti rozčlenil na tri časti. V prvej časti celá scéna je vykreslená stlmeným svetlom, čo reprezentuje osvetlenie telies v tieni. V druhej časti prostredníctvom mapovania textúry, ktorá bola vytvorená v prvom prechode, do stencil bufferu sú uložené časti scény, ktoré sú osvetlené. V poslednej časti použitím stencil testu je vykreslená scéna jasným svetlom. Tieto časti by sa dali zlúčiť použitím rozšírenia `ARB_multitexture`, alebo použitím shaderov.

Kapitola 3

Scéna

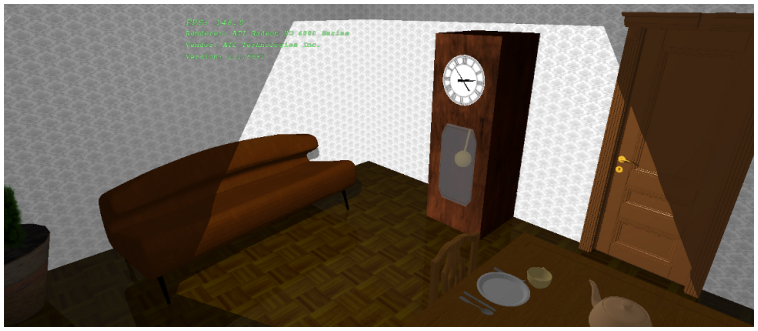
Táto kapitola obsahuje popis scény, ktorá bola použitá pri testovaní.

3.1 Formáty použitých súborov

Zložitejšie modely v scéne sú uložené v súboroch v tzv. “raw” formáte. Súbory majú v každom riadku deväť vertexových súradníc. Tieto definujú jednu plochu v modelu. Tento “raw” formát sa dá jednoducho načítať bez akejkoľvek externej knižnice pomocou základných operácií a pomocou jednoduchých štruktúr. Nevýhodou tohto formátu je to, že neobsahuje ani normály, ani informácie o farbe vertexov, či o materiálu alebo o súradniciach textúr, ako niektoré zložitejšie formáty. Nanesenie textúry na takéto objekty je možné po jeho rozčlenení na časti a následným mapovaním textúr. Modely boli vytvorené alebo konvertované pomocou programu Blender 3D. Na exportovanie do formátu popísanej vyššie Blender má rozšírenie implementované práve v programovacom jazyku Python.

Textúry sú vo formátu tga (Truevision (Targa) File Format). TGA je bitmapový formát, ktorý môže obsahovať 24 a 32 bitové obrázky, ktoré je možné ľahko načítať do textúr. ([10]) Obrázky použité ako textúry pochádzajú väčšinou z internetu.

3.2 Časti scény



Obr. 3.1: Scéna - Hodiny, gauč, dvere

Scéna je jedna malá zariadená izba. Pri dverách sú stojacie hodiny. Na čelnú stranu je nanesená čiastočne priehľadná textúra. Transparentnosť je dosiahnutá pomocou 32 bitovej textúry a povolením blendovania v OpenGL. Táto metóda nezaručuje reálnu transparentnosť.

Jedná sa len o kombináciu farby dvoch pixelov: z ktorých prvá bola už v bufferu (zdroj), a druhá by mala byť vykreslená (cieľ). Na určenie miery, v ktorom budú farby kombinované, slúži funkcia `glBlendFunc(zdroj, cieľ)`. V tomto prípade bol použitý pre faktor zdroje `GL_SRC_ALPHA` a `GL_ONE_MINUS_SRC_ALPHA` pre faktor destinácie.

Hodiny ukazujú presný čas. Získať dátum a presný čas na milisekundy je snadné tak v Pythonu ako aj v C, rozdiel je len v počtu riadkov, ktorý je v C dvojnásobok počtu riadkov implementácie v Pythonu. V Pythonu aj kód je jasnejší a priehľadnejší, a hlavne viacplatformový. V jazyku C je potrebné použiť inú metódu a na iných platformách. Ciferník hodín je vytvorený pomocou kvadriky funkciou `gluDisk()`.

Vedľa hodín je gauč a kvetináč s malým kríkom. Na gauči je mapovaná textúra kvôli dosiahnutiu vzhľadu koženého povrchu. V PyOpenGL pre mapovanie textúr je potrebné dovoliť `GL_TEXTURE_GEN_[STRQ]` v závislosti od toho, že ktoré súradnice textúry chceme generovať. Ďalej je treba určiť parametre a metódu generovania textúry. V tomto prípade bola použitá metóda `GL_OBJECT_LINEAR`. Parametre určia koeficienty pre generovanie textúr. Pri `GL_OBJECT_LINEAR` zvolená súradnica textúry pre daný vertex sa vypočíta podľa následujúcej rovnice: $g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o$, kde p_1, p_2, p_3, p_4 sú koeficienty a x_o, y_o, z_o, w_o sú objektové súradnice vertexu.

Kvetináč, podobne ako ciferník na hodinách, je vytvorený pomocou kvadrík: pôda pomocou funkcie `gluDisk` a valec pomocou `gluCylinder`. Rastlina v kvetináči je reprezentovaná bilboardom. Transparentnosť je riešená podobne ako v prípade čelnej strany hodín.

Obr. 3.2: Scéna - Kozub, stôl

Oproti dverám je mramorový kozub, v ktorom horí oheň. Súradnice mramorovej textúry sú generované podobným spôsobom ako u gauče. Oheň je príkladom na časticový systém. Každá častica ohňa má x, y koordináty, životnosť aj rýchlosť. Pri vytvorení častice súradnica x, životnosť a rýchlosť sú náhodne generované čísla. Po každej snímke častice, ktorým životnosť vypršala, alebo ktoré sa dostali mimo hranice, sú znovu vytvorené. Ostatné častice sú posunuté po ose y na základe ich rýchlosti. Výkres častíc je riešená troma rôznymi spôsobmi:

Priame volanie OpenGL funkcií Táto metóda by mala byť najpomalšia. Pre každú časticu je vytvorený štvorec pomocou funkcií `glVertex` a koordináty textúr (`glTexCoord`).

Použitím Display listov Vytvorenie štvorca a priradenie textúrových súradníc k vertexom sú uložené do display listu.

Použitím polí vertexov a textúr Všetky súradnice textúr a vertexov sú uložené do pole. Vytvorené pole sú predané funkciám `glVertexPointer` a `glTexcoordPointer`, ktoré sa postarajú o výkres všetkých častíc.

Ďalej je možno nastaviť počet častíc. Táto možnosť je užitočná hlavne pri testovaní.

Medzi kozubom a dverami je stôl s dvoma stoličkami. Na stole je pár hrnčekov, tanierov, príbor a klasický čajovník. Na týchto predmetoch sa dá demonštrovať presnosť tieňov.

3.3 Tiene

Tiene v scéne sú realizované podľa metódy popísanej v druhej kapitole. Najprv je scéna renderovaná z pozície svetla. Hodnoty z bufferu hĺbky sú prekopírované do textúry funkciou `glCopyTexImage2D`. Za interný formát som zvolil `GL_DEPTH_COMPONENT32_ARB`, ktorý je súčasťou rozšírenia `GL_ARG_depth_texture`. Po úspešnom uložení hodnôt hĺbky scéna je vykreslená matným svetlom. V ďalšom kroku je použitá textúra z prvého kroku. Textúra je namapovaná na celú scénu. Koeficienty na generovanie textúrových súradníc sú vypočítané z rovnice (vysvetlenie viď [7]):

$$\begin{bmatrix} p_{s1} & p_{s2} & p_{s3} & p_{s4} \\ p_{t1} & p_{t2} & p_{t3} & p_{t4} \\ p_{r1} & p_{r2} & p_{r3} & p_{r4} \\ p_{q1} & p_{q2} & p_{q3} & p_{q4} \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Light \\ Projection \\ matrix \end{bmatrix} \begin{bmatrix} Light \\ ModelView \\ matrix \end{bmatrix}$$

O zbývajúcich výpočtoch sa postará `glTexGen`. Na rozdiel od generovania súradníc textúry na modely v scéne je použitá metóda `GL_EYE_LINEAR`. Pri tomto kroku je využité ešte aj rozšírenie `GL_ARB_shadow`, ktoré umožňuje filtrovanie tieňovou mapou. Pomocou alpha testu určíme, ktoré pixely sú osvetlené. Pozícia týchto pixelov je uložená do stencil bufferu. Pri poslednom kroku bol použitý stencil test na renderovanie osvetlených častí scény.

Odlesk objektov na leštenej parkete je riešený stencilovaním a vykreslením scény otočenej a následne posunutej pod podlahu.

Kapitola 4

Testovanie a implementácia

Cieľom tejto práce bolo vytvoriť testovaciu aplikáciu v Pythonu, na ktorom bude možné demonštrovať možnosti jazyka Python v počítačovej grafike a určiť výhody či nevýhody tohoto prístupu oproti klasickému programovaniu. Pri využití v počítačovej grafike najdôležitejší faktor na určenie výhod či nevýhod Pythonu spolu s PyOpenGL je meranie výkonu a porovnanie dosiahnutých výsledkov s implementáciu v programovacom jazyku C.

4.1 Spôsob testovania

Program som testoval na desktopovom hardveru, ktorý je možné v súčasnosti považovať za štandard. Grafická karta bola ATI Radeon HD 4850 s 512MB GDDR3. Druhá najdôležitejšia súčiastka v počítačovom grafike je procesor, na ktorom závisí výkon programu. Pri testovaní v počítačovej zostave som mal procesor Intel Core 2 Quad 8200 na frekvencii 2.8 GHz. Grafické aplikácie sú citlivé na taktovanie procesoru. Na vyšších frekvenciách dávajú rádovo väčší výkon, a preto väčšina súčasných mobilných procesorov, ktoré bežia pri frekvenciách 1.8 GHz - 2 GHz a menej pri testovaní takýchto aplikácií nepripadajú do úvahy.

Demonštračnú aplikáciu som testoval na dvoch platformách: Microsoft Windows Vista, Linux (Archlinux). Verzie spoločných závislostí boli:

- cPython 2.6.2
- PyOpenGL 3.0.0
- PIL (Python Imaging Library) 1.1.6
- NumPy 1.3.0

Ďalej závislosti u Visty:

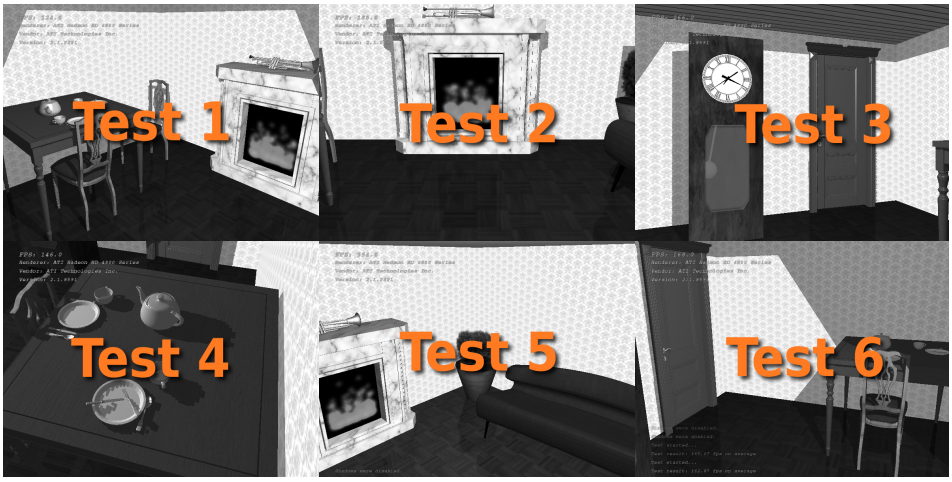
- glut 3.7
- MinGW 5.1.4

V prípade linuxu:

- kernel 2.6.29
- xorg 1.6.1

- freeglut 2.4.0
- gcc 4.4

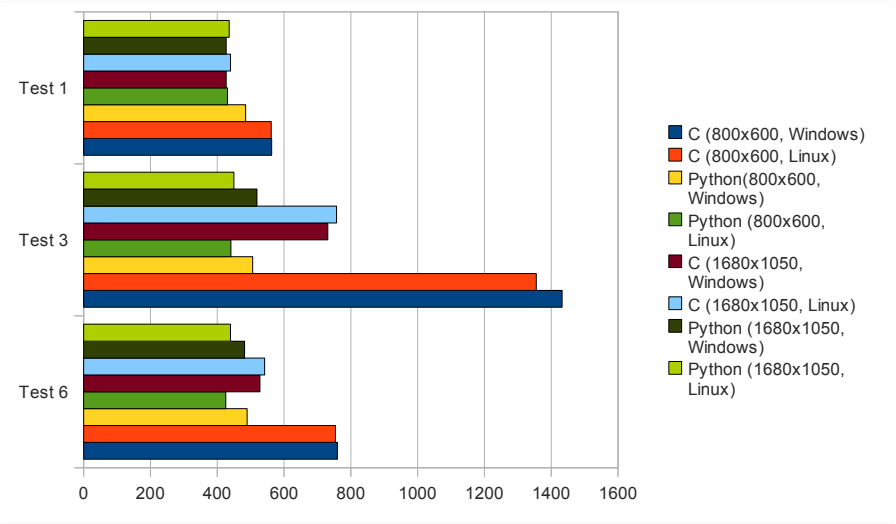
V oboch prípadoch som použil oficiálne ovládače od ATI, ktoré v linuxu ešte nie sú tak vyzrelé ako vo Windows, takže som očakával horšie výsledky v prípade tejto platformy.



Obr. 4.1: Testy

V programe som implementoval šesť testov. Päť z nich sú statické merajúce snímky za sekundu z päť odlišných pozícií scény. U posledného testu kamera sa otáča okolo centra scény. Počas statických testov svetlo je tiež statické a u posledného testu sa otáča opačným smerom ako kamera okolo centra scény. Z výsledkov zvolených testov som vytvoril graf pre každú testovanú vlastnosť. Výsledky všetkých vykonaných testov sú uvedené v prílohe.

4.2 Základná scéna



Obr. 4.2: Graf - Základná scéna

V tejto časti som sa snažil vypnúť všetky možné efekty, aby som mohol porovnať čiste rýchlosť volaní OpenGL funkcií u jazyku C a Python. Medzi vypnutými efektmi boli predovšetkým: časticový systém, odraz, textúrovanie, tiene, svetlá a všetky informačné texty okrem monitoru snímkov za sekundu. Zapnutie týchto vlastností som testoval zvlášť: testy z tejto časti môžu slúžiť ako základ na získanie miery zhoršenia výkonu.

Obr. 4.3: Scéna - Základná scéna

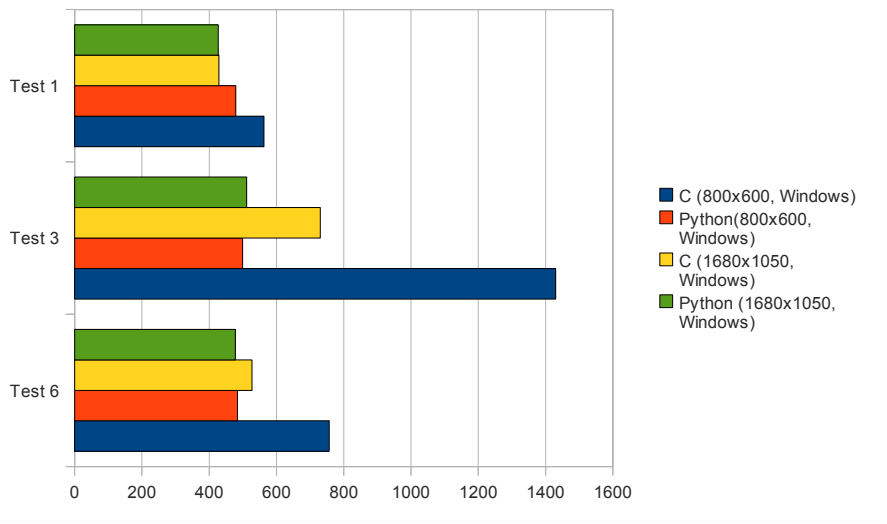
Vo väčších rozlíšeniach rozdiel medzi počtom snímkov za sekundu v prípade implementácií v C a Pythonu sa zmenšilo. Diferencia bola ešte menšia ako v testu číslo 1, ak vo viditeľnej časti scény sa nachádzalo veľké množstvo objektov: stôl so stoličkami, príbor, taniere, hrnčeky, čajovník atď. Naopak, ak v aktuálnom fruste bolo málo polygónov, rozdiel bol rádovo väčší. V prípade testu číslo 3 pri rozlíšení 800x600 počet snímkov u implementácie v C bol až trojnásobok počtu snímkov než v prípade implementácie v Pythonu.

K priemernému rozdielu má najbližšie asi test číslo 6. V tomto teste sa kamera otáča okolo stredu scény a tak každá časť scény je vykreslená aspoň raz. Rozdiel podľa tohoto testu bol v prípade rozlíšenia 800x600 40% a pri rozlíšení 1680x1050 asi 15%.

Z týchto testov vyplýva, že pri základných funkciách, kde interpret Pythonu nemá hlavnú rolu pri vykonávaní programu, je výkonnosť Pythonu postačujúca.

Čo sa týka diferencie medzi platformami, Python pod Vistou je o 5-10% rýchlejší. Program napísaný v programovacom jazyku C je zase na oboch platformách približne rovnako rýchly. Z toho vyplýva, že rozdiel v prípade Python programu nepochádza z rozdielu medzi ovládačmi zmienených platforiem, ale z rozdielu rýchlostí interpretov Pythonu. V nasledujúcich testoch, ak rozdiel medzi platformami neprekročilo tieto medzery v grafoch, som uviedol len hodnoty získané z jednej platformy.

4.3 Osvetlenie



Obr. 4.4: Graf - Osvetlenie

OpenGL používa Phongov osvetľovací model. Aproximuje svetlá a osvetlenie rozdelením svetla na červenú, zelenú a modrú zložku. Zdroje svetla sú potom charakterizované množstvom červeného, zeleného či modrého emitovaného svetla, a povrchy percentom odrazeného svetla po jednotlivých zložiek. Výpočty osvetlenia u OpenGL sú len apriximáciou reálneho svetla a dajú sa vypočítať relatívne rýchlo. ([12])

Osvetľovací model OpenGLu predpokladá, že svetlo sa dá rozdeliť na štyri nezávislé komponenty: emisný, ambientný, difúzny a spekulárny. Všetky štyri komponenty sú vypočítané zvlášť a následne sčítané. V OpenGL mieru odrazu týchto zložiek je možné nastaviť zvlášť funkciami `glMaterial[fv]()`. Problém s rýchlosťou `*v()` funkcií v Pythonu som popísal v kapitole venovanej profilovaniu. Keďže všetky výpočty sú prevedené hardverom, zapnutie osvetlenia by nemala spôsobovať významný pád výkonu.

V scéne je len jeden zdroj svetla umiestnená o málo vyššie od centra scény. Je to bodové svetlo. Na imitáciu svetla vo vnútri miestností je vhodný tento typ svetla, kým na modelovanie slnka smerové svetlo. Osvetľovací model OpenGLu je veľmi jednoduchý. Ani jeden typ svetla nepočíta s tieňmi. Spôsob výpočtu a renderovanie tieňov sú preto ponechané na programátora. Ďalšia nevýhoda OpenGL, že emisné svetlo objektov nepridá žiadny zdroj svetla

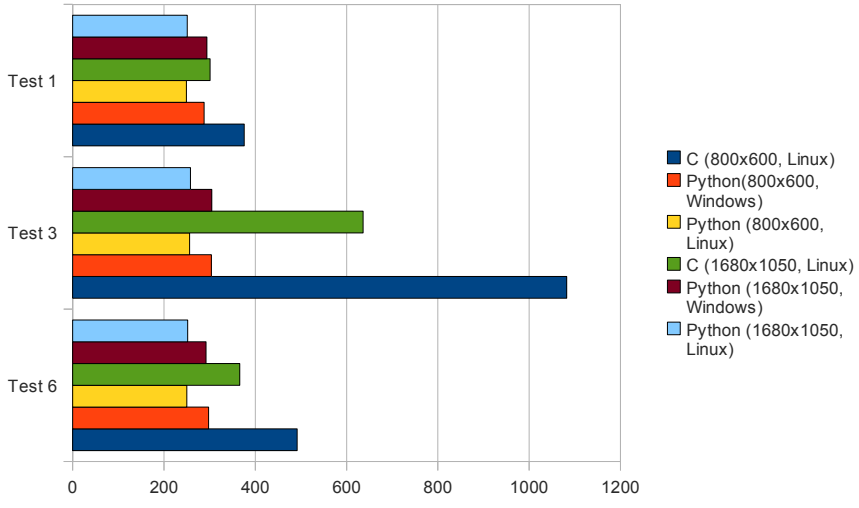
Obr. 4.5: Scéna - Osvetlenie

k scéne, a preto nie je možné vytvoriť plošné zdroje svetla. Všetky tieto problémy je možné riešiť vlastným výpočtom osvetlenia v softveru. Takýto výpočet je však veľmi zložitý.

Tieto testy potvrdzujú, že zapnutie osvetlenia scény nemá veľký vplyv na výkon. Rozdiel v prípade oboch implementácií sa pohybuje v intervale 0 až 5 snímkov za sekundu. Malou námahou takto získame oveľa reálnejší pohľad na scénu.

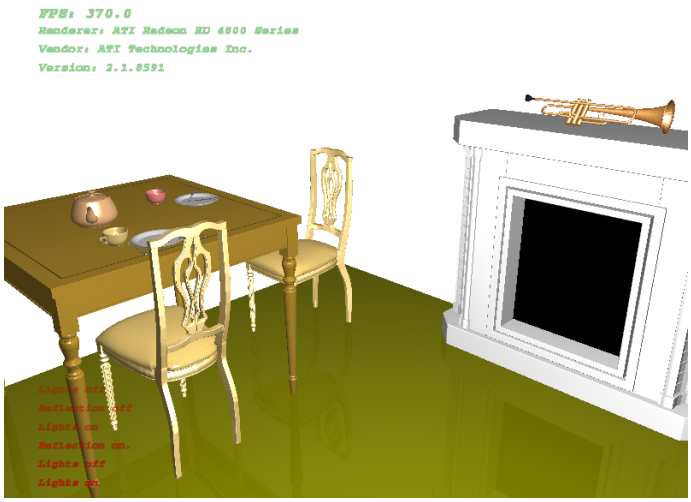
V ďalších testoch som nechal osvetlenie zapnuté. V prípade testov tieňov je to nutné, v ostatných prípadoch som to použil len kvôli reálnejšiemu efektu.

4.4 Odlesk



Obr. 4.6: Graf - Odlesk

Spôsob vytvorení odleskového efektu je popísaný v kapitole 2.3 Použité technológie. Celá scéna je vykreslená ešte raz pod odleskovej vrstvy. Očakával som preto pád počet snímkov za minútu asi na polovinu oproti počtu bez tohoto efektu.



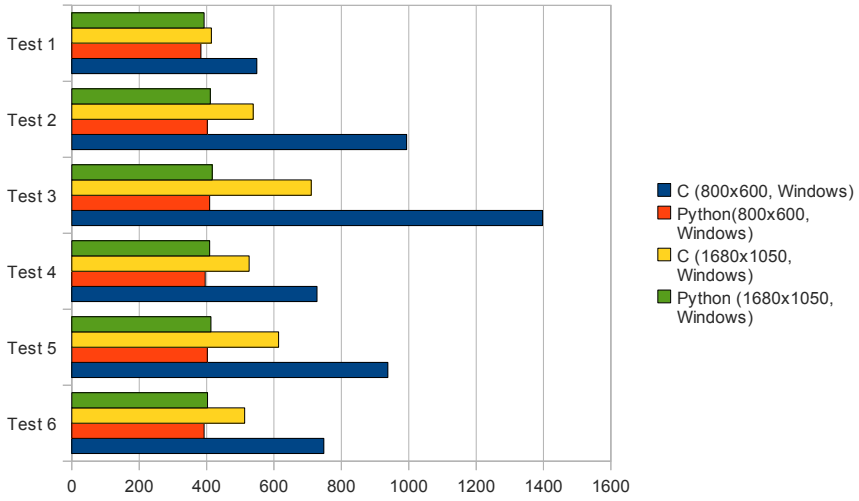
Obr. 4.7: Scéna - Odlesk

Testy ukazujú pokles o 40%. Výnimku tu tvorí test číslo 3, kde je v popredí v porovnaní s ostatnými testami najmenší počet objektov. V prípade testu číslo 1, kde je najväčší počet objektov, výkony u implementácie v programovacom jazyku C a v Pythonu sú dosť vyrovnané. Tento výsledok je veľmi dobrý vzhľadom na to, že Python je interpretovaným jazykom.

Existuje tu niekoľko problémov s týmto spôsobom renderovania odleskov. Implementoval som toto renderovanie čo najjednoduchším spôsobom. Nebral som do úvahy tiene, ani svetlá. Svetlá aj tiene by bolo treba transformovať rovnakým spôsobom ako scénu. Pri svetlách nie je to tak veľký problém, ale o to pomalší a ťažší by bolo vykreslenie tieňov aj pre odzrkadlené objekty. Pri vykreslení tieňov pre pôvodné objekty používa program stencil test, ktorý je potrebný aj pre určenie odleskového povrchu.

Ďalším problémom, ktorý však nesúvisí s implementáciou, ale s technológiu, je neschopnosť odrážať svetelné lúče. Riešením tohoto problému je použitie ray-tracingu.

4.5 Textúry



Obr. 4.8: Graf - Textúry

Venoval som sa aj testovaniu vplyvu povolení textúr na výkon. Skúmam tu len textúry popisujúci povrch objektov. Koordináty pre textúry sú v programu jednak definované, a v niektorých prípadoch aj generované. Generovanie bolo treba použiť v prípade modelov, ktoré program načíta zo súborov. Formát týchto súborov neobsahuje ani pozície textúr, ani normály, a preto bolo potrebné oba, aj koordináty textúr, aj normály softverovo vypočítať či generovať. Textúrovanie je náročnejšia operácia ako počítanie osvetlenia v OpenGL. U Pythonu je procesor už úzkym profilom kvôli interpretu, a ďalšie spomalenie pri niektorých operáciách by mohli ešte viacej spomaliť vykresľovanie.

Obr. 4.9: Scéna - Textúry

Dosiahnuté výsledky je možné porovnávať s testami s povoleným osvetlením a bez odleskov. Pri porovnaní som vychádzal z testu číslo 6. Výsledky tohoto testu sú viacmenej priemerom všetkých testov.

Kým pri menšom (800x600) rozlíšení pri C verzii došlo len k malému poklesu v počte snímkov za sekundu – z 757 FPS na 748, tak u Python verzií počet snímkov klesol v priemeru asi o 20%. Možné vysvetlenie som načrtol vyššie. Pri väčších rozlíšeniach (1680x1050) u implementácie v jazyku C výkon klesol opäť len málo, približne o 2%. V tomto prípade výkon programu v Pythonu sa zhoršil približne o 15%.

Zvláštnym javom je zvýšenie počtu snímkov za sekundu pri testovaní Python programu pri natívnom, teda väčšom rozšírení, oproti testovaniu pri menšom rozšírení. Možným vysvetlením môže byť optimalizácia ovládačov a hardveru na takéto rozlíšenie. Zvýšenie počtu snímkov sa však neprejaví pri implementácii v C, kde procesor nie je preťažený v takej miere.

V tejto časti by som ešte rád spomenul na výpisy v popredí scény, keďže znaky sú vykreslené pomocou textúry, ktorá obsahuje všetky potrebné písmená. Vykreslenie každého znaku je uložené zvlášť do display listu. Pre každý riadok je potom určená rada charakterov a tieto sú vykreslené použitím funkcie `glCallLists()`.([11]) Pri implementácii v C dĺžka riadku je limitovaná na 1023 znakov. Toto obmedzenie sa programu v programovacom jazyku Python netýka. Python je flexibilnejší pri použití reťazcov, než všetky interpretované jazyky bežne.

Na obrazovke je možné povoliť výpis oznamovacích správ, výpis informácií o grafickej karte a počtu snímkov za sekundu. Informácie o grafickej karte sú získané pomocou funkcie `glGetString()`.([10]) Oznamovacie správy sú uložené vo fronte. Táto fronta u Pythonu je implementovaná ako jednoduchý zoznam. Bohatý zoznam funkcií nad typom `list` umožnil celkom jednoduchú implementáciu. Táto práca so zoznamami je jeden z mnohých výhod Pythonu oproti klasickému programovaniu v C. V jazyku C som musel vytvoriť pomocné štruktúry a funkcie, aby som dostal podobný výsledok.

Ďalším problémom, na čo som narazil pri implementovaní programu v jazyku C, bolo získanie presného času pre výpočet počtu snímkov. Potreboval som presnosť aspoň na milisekundy, ktorú je možné získať na oboch platformách rôznym spôsobom: vo Windows použitím WinAPI a pod linuxom pomocou funkcie `gettimeofday`, ktorá je definovaná v hlavičke `<sys/time.h>`. Tento krok ďalej komplikoval kód, keďže na unixových platformách nie sú k dispozícii hlavičkové súbory pre WinAPI. Aj pri tom sa ukážu prednosti Pythonu, ktorý pre tento účel má viacplatformové riešenie.

Počas testovania som zakázal výpis oznamovacích a informačných textov, keďže by mali veľký vplyv na výkon programu. To platí hlavne v prípade oznamovacích správ, ktoré môžu mať počas testovania rozdielne dĺžky. Po vykonaní testu sú oznamovacie správy zase povolené na zobrazenie výsledkov testu.

4.6 Tiene

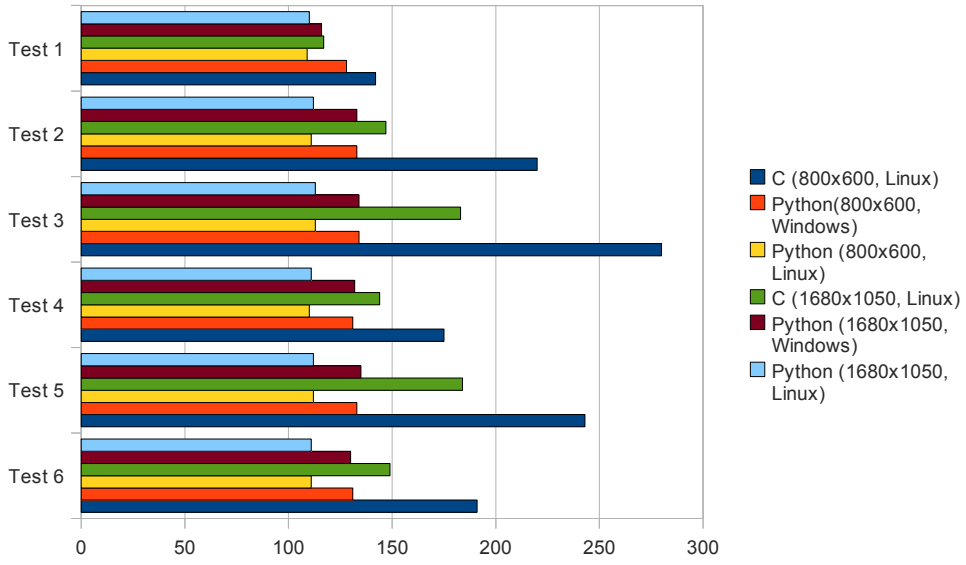
Tiene sú po osvetľované jedným z najdôležitejších efektov pri vytváraní scén, ktoré ich približujú k realite. Existuje mnoho spôsobov ako vypočítať a vykreslovať tiene. V tejto práci som použil jednoduché mapovanie tieňov. Proces vytvárania tieňov je opísaný v druhej kapitole.

Implementácia tohoto efektu je dobrým príkladom na demonštrovanie výhod Pythonu proti C. Python má nielen mohutnú štandardnú knižnicu, ale aj veľmi užitočné špecializované knižnice akou je napríklad aj NumPy. Kým pomocou NumPy násobenie, vypočítanie transponentu matíc či získanie riadky alebo stĺpce matice je otázka zavolania zodpovedajúcej funkcie, tak v C je treba všetky výpočty urobiť manuálne. Ba naviac, v knižniciach ako NumPy, ktoré sú napísané v programovacom jazyku C, sú tieto funkcie už optimalizované. V prípade manuálneho výpočtu nie je isté, že programátorom zvolený spôsob implementácie

Obr. 4.10: Scéna - Tiene

je najrýchlejší. Existujú síce knižnice na prácu s maticami aj pre jazyk C, ale tieto ďaleko nie sú tak mocné, flexibilné a rozšírené ako NumPy v prípade Pythonu.

Ďalšou výhodou PyOpenGLu je práca s rozšíreniami. V PyOpenGLu na zistenie prístupnosti a pre inicializáciu rozšírenia stačí naimportovať príslušný balík zostavený z mena rozšírenie. Napríklad pre rozšírenie použité na mapovanie textúr `GL_ARB_depth_texture` by import vyzeral následovne: `import OpenGL.GL.ARB.depth.texture`. V jazyku C je treba najprv parsovať reťazec vrátenou funkciou `glGetString(GL_EXTENSIONS)` na zistenie, či dané rozšírenie je podporované na danej platforme a potom je potrebné ešte inicializáciu previesť ručne.



Obr. 4.11: Graf - Tiene

Keďže počas vykreslenia jednej snímky na obrazovku scéna je vykreslená okrem ohňa a odleskov štyrikrát, čakal som, že sa výkon zníži na štvrtinu. Výsledky je možné prirovná-

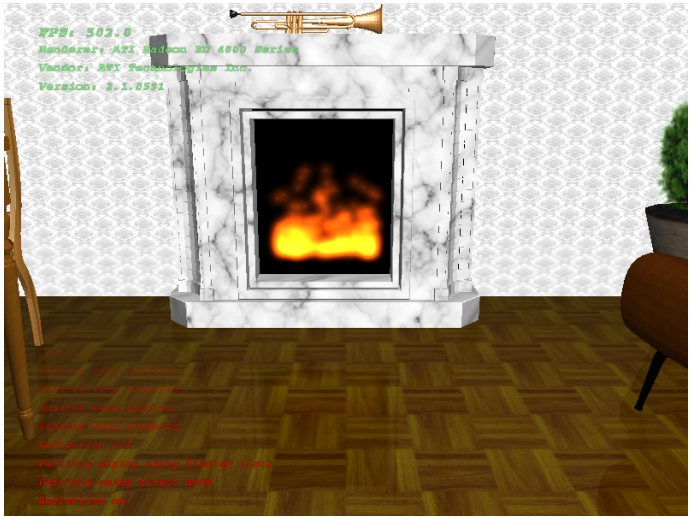
vať k výsledkom z predchádzajúcej časti, lebo textúrovanie som nechal povolené pre lepší výsledný vzhľad. Výsledky testov potvrdili môj odhad. Programy sa spomalili v priemeru okolo 70%. Jediný väčší rozdiel bol u testu číslo 3. Pri rozlíšení 800x600 implementácia v C stratil trojnásobný náskok oproti implementácii v Pythonu. Počet snímkov za sekundu pri C verzii bol v tomto prípade dvojnásobný ako v Pythonu.

Najväčšou nevýhodou tejto metódy vykresľovania tieňov je aliasing. Ukážka je na obrázku 2.2. Táto metóda nepočíta ani s transparentnosťou. V dôsledku toho vznikajú tiene aj na miestach, kde by ich nemalo byť. Napríklad za bilboardom zobrazujúci krík, kde transparentnosť je riešená pomocou blendingu a transparentnou textúrou. Podobným, ale menej výrazným príkladom je kyvadlo hodín, ktoré je celé v tieni. Krajné obrázky ukazujú objekty po povolení tieňov, pri stredných obrázkoch bolo povolené len osvetlenie.

Obr. 4.12: Tiene - Problémy s mapovaním

Aliasing je pomerne snadné liečiť pomocou mutlitexturovania a lineárne filtrovanou textúrou. Ani to však nezmení na tom, že presnosť metódy je daná veľkosťou tieňovej textúry. Oba pred chvíľou spomenuté problémy by vyriešil ray-tracing alebo použitie komplikovanejšej metódy.

4.7 Časticový systém



Obr. 4.13: Scéna - Časticový systém

Časticové systémy slúžia na simuláciu náhodných fyzikálnych javov a rýchlosť takýchto simulácií je priamo úmerná rýchlosti procesoru. Simulácie vyžadujú veľkú výpočtovú silu

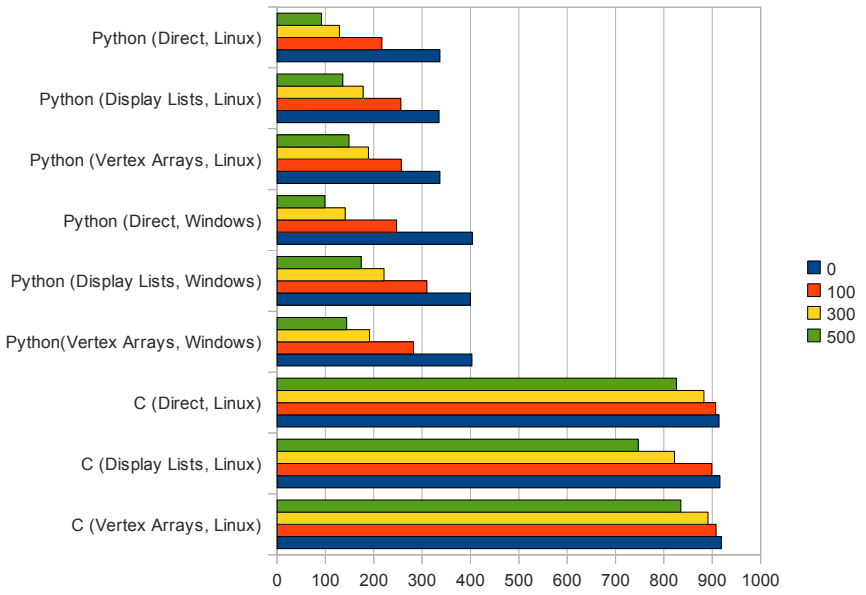
a preto na tieto účely v grafike sa zvyčajne používajú shadery. Shadery sú programy, ktoré sú vykonané priamo na grafickej karte, a tým možno dosiahnuť podstatne rýchlejšieho spracovania, ako keby program spracovával univerzálny procesor počítača. Keďže však v tejto práci porovnávam výhody a nevýhody Pythonu, na všetky výpočty boli použité testované programovacie jazyky.

Na grafe som uviedol len výsledky testov finálnej verzie programov. Na začiatku implementácia verzie v Pythonu bola neuveriteľne pomalá a potrebovala optimalizáciu.

Najprv som sa snažil využiť prostriedky poskytované jazykom Python ([6]), teda využiť objektovo orientovaný prístup, čo sa ukázal byť nevhodným pre tento účel. Každá častica bola inštanciou triedy `FireParticle`. Trieda obsahovala premenné pre zostávajúci život častice, XY pozície a rýchlosť. Ďalej obsahovala ešte metódu buď na posunutie častice podľa rýchlosti po ose Y, alebo na generovanie náhodných hodnôt pre X, pre rýchlosť a životnosť, ak častici došiel život. Po vykreslení častice bola vždy zavolaná táto metóda pre každú inštanciu triedy zvlášť. Volanie takéhoto množstva funkcií je v Pythonu veľmi pomalé. Na vykonanie týchto operácií sa v Pythonu používa interpret, čo je radovo pomalší než alternatívny kód v jazyku C, ktorý je známy rýchlosťou volania funkcií.

Počas optimalizácii som sa snažil vyhýbať kódu, ktorý vykonáva priamo interpret, a použiť funkcie a operácie, ktoré sú implementované v knižniciach v jazyku C. Posunutie všetkých častíc som dal do jednej funkcie, a na generovanie náhodných čísiel som použil funkcie poskytnuté knižnicou NumPy.

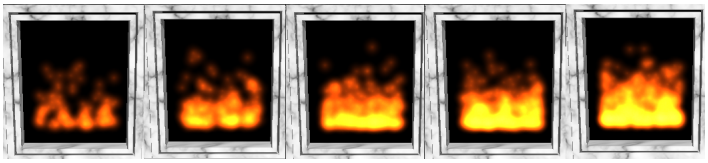
Pôvodne som všetky častice vykresľoval priamym kódom. Aj tu sa prejavila pomalosť volania funkcií v Pythonu. Prvá logická voľba pri optimalizácii bola použitie display listov. Tým počet volaných OpenGL funkcií pre jednu časticu klesol z desať na štyri. V ďalšom kroku som implementoval vykresľovanie častíc prostredníctvom polí vertexov a textúr. Táto metóda požadovala zavolanie celkom sedem funkcií OpenGL. Z týchto štyri funkcie sa týkali aktivácie a po vykreslení deaktivácie polí vertexov a textúr. Nevýhodou poslednej metódy však bola zvýšená spotreba pamäti a komplikovanejší posuv častíc.



Obr. 4.14: Graf - Časticový systém

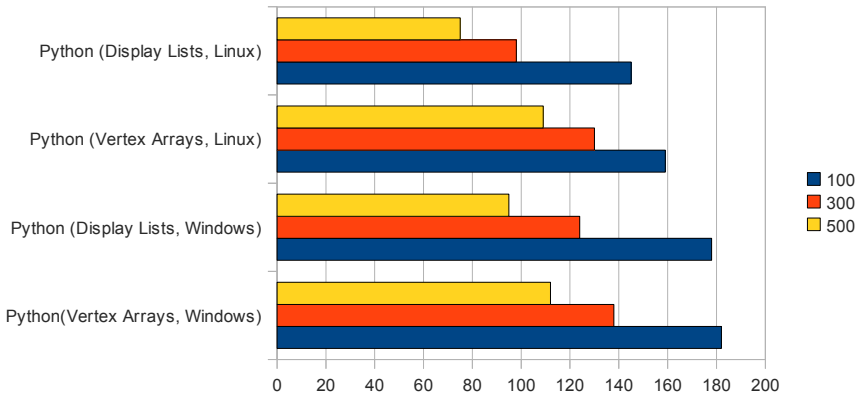
Testoval som všetky tri metódy vyššie spomenutej metódy na demonštráciu rozdielu v rýchlosti: per-vertex kód, použitie display listov a pole vertexov. V grafe som uviedol výsledky pre 0, 100, 300 a 500 častíc.

Ukázalo sa, že v Pythonu bol najpomalší priamy kód. Použitie display listov v týchto testoch bolo približne rovnako výkonné, ako v prípade pole vertex, a pod Vindows aj rýchlejší. V implementácii v jazyku C na rozdiel od Pythonu bola najpomalšou práve metóda pracujúca s display listmi. Display listy sú uložené na strane klienta v OpenGL a po každom zavolaní sa inštrukcie prekopírujú na server. Táto réžia mohla zapríčiniť pokles vo výkone.



Obr. 4.15: Časticový systém - Výsledok podľa počtu častíc

Rozdiel vo výkone v prípade Pythonu bol minimálne 50% medzi stavom so žiadnymi časticami a stavom s maximálnym počtom častíc. Kód v jazyku C bol až päťnásobne rýchlejší. Pri implementácii v C pokles výkonu bol maximálne len 20 % nižší. Z toho vyplýva, že Python je absolútne nevhodný na simulácie a rozsiahle výpočty. Možným riešením by bolo: celý systém implementovať v C s použitím Python C API, alebo by bolo potrebné vytvoriť Python modul v C pre všetky matematické a rozsiahle výpočty. Tým by sa dalo vyhnúť pomalému spracovaniu operácií interpretom Pythonu. Podobnému cieľu slúži aj nástroj SWIG (Simplified Wrapper and Interface Generator), ktorý medzi mnohými jazykmi podporuje aj Python. SWIG vytvára spojenie medzi knižnicami napísané v jazyku C a skriptovacím jazykom. SWIG je vlastne prekladač, ktorý vezme deklarácie z jazyka C/C++ a vytvorí kód umožnujúci prístup k týmto prvkom z vysoko úrovňového jazyka.

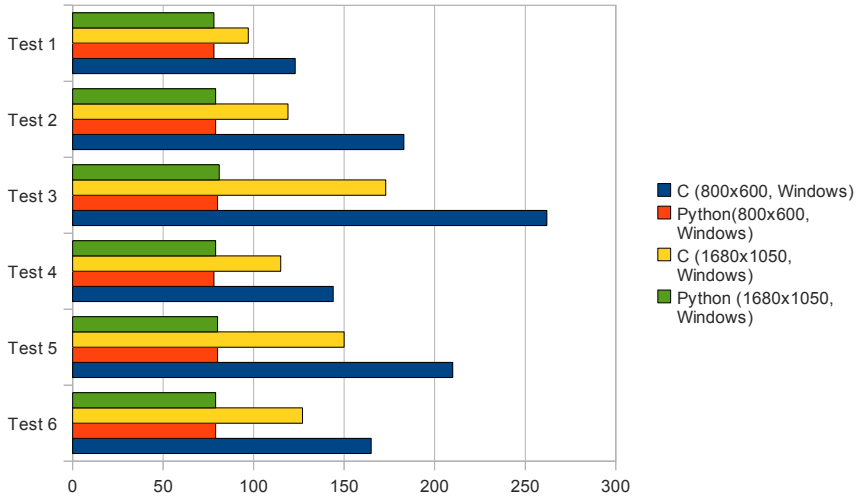


Obr. 4.16: Graf - Časticový systém s odleskom

Pri testovaní som očakával, že najrýchlejšia bude implementácia s polami vertexov a textúr, ale pod Vistou test s display listmi bol len o málo rýchlejší. Preto som sa rozhodol urobiť ďalšie testy so zapnutými odleskmi. Takto všetky častice sú vykreslené dvakrát a posunuté len raz. Na výkon tak bude mať vlastne väčší vplyv vykreslenie častíc, než vypočítanie nových súradníc a zostávajúceho života častice.

Z grafov vyplýva, že použitie pole vertexov a textúr je v skutočnosti naozaj rýchlejšie. V predchádzajúcich testoch k náskoku rýchlosti použitia display listov došlo kvôli pomalšiemu posunu častíc u pole vertexov.

4.8 Všetko zapnuté



Obr. 4.17: Graf - Všetko zapnuté

V tejto časti som povolil všetky zatiaľ testované vlastnosti a nechal som zapnutý aj výpis o informáciách o grafickom hardvere. Oznamovacie správy, ako som už spomenul, som nechal vypnuté, keďže ich dĺžka je premenná a dlhšie správy by mali príliš veľký vplyv na rýchlosť vykresľovania celej scény. Počet častíc u ohňa som nastavil na 300. Do grafu som dával len výsledky merané na platforme Windows. Výsledky na platforme linux v prípade programu v C boli na snímok rovnaké ako vo Windows. Pri programu v Python na Windows dosiahol o 10($\pm$2) % lepšie výsledky než na linuxu, a to bez výnimiek. Preto som nevidel dôvod dávať ich do grafu.

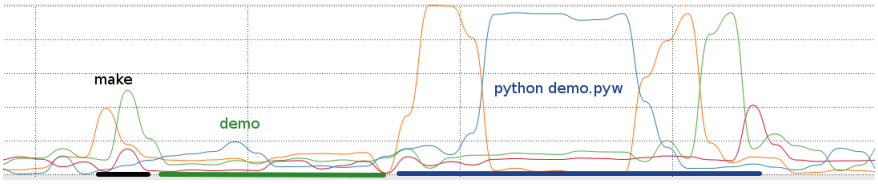
Ako pri testovaní skoro každého efektu či vlastnosti, najväčší rozdiel medzi implementáciami v jazykoch C a Python bol v prípade testu číslo 3 pri menšom rozlíšení (800x600). Dôvodom toho pravdepodobne je malý počet objektov vo fruste. Najmenší rozdiel bol v prípade testu číslo 1. Program v jazyku C bol o 38 % rýchlejší pri väčšom rozšírení ako interpretovaný program v Pythonu. V prvých testoch, kde všetky efekty boli ešte vypnuté, rozdiel nedosiahol ani 2 %. Hlavnú príčinu zvýšenia rozdielu vidím v zapnutí časticového systému, ktorý je najslabším článkom v programu napísanom v Pythonu.

Aj v tomto prípade som bral za určujúce výsledky testu číslo 6, ktoré zodpovedajú s najväčšou pravdepodobnosťou reálnemu rozdielu v rýchlosti medzi interpretovaným a preloženým programom. Pri menšom rozlíšení diferencia vo výkone medzi implementáciami bola väčšia ako pri väčšom rozlíšení. V číslach je to 52 % percent pri rozlíšení 800x600 a zhruba 38 % v prípade rozlíšenia 1680x1050. Dosiahnutý výsledok z pohľadu Pythonu je ešte stále prijateľný. Python je asi desaťkrát pomalší pre rovnaký algoritmus ako alternatívny C kód (viď [5]), ale vhodnou optimalizáciou je možné dosiahnuť aj lepšie výsledky ([1], [2]).

Obr. 4.18: Scéna - Tiene, osvetlenie, časticový systém

4.9 Súhrn testov

Pri prvom pohľadu na grafy je možné si všimnúť, že kým u programu v C je možné sledovať značný pokles v počte snímkov za sekundu po prechodu na vyššie rozlíšenia, v prípade programu v Pythonu výsledky sú približne rovnaké, a niekedy aj väčšie pri vyšších rozlíšeniach. Príčinu toho vidím opäť vo vysokej záťaži procesoru interpretom Pythonu. Na potvrdenie môjho predpokladu som sledoval zátaž procesoru počas behu v prípade oboch programov (Python a C) pomocou monitoru systému z grafického prostredia GNOME. Počas behu programov som spustil test číslo 6, ktorý trvá približne 20 sekúnd. Výsledok je na nasledujúcom grafe:



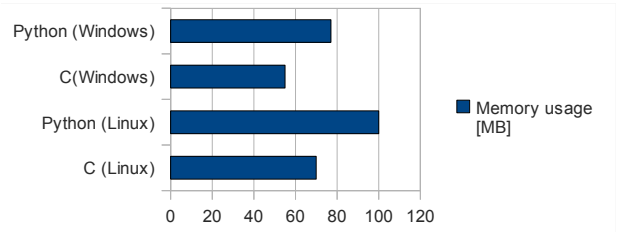
Obr. 4.19: Zátaž procesoru

Na grafe som vyznačil odlišne farbou tie časti, v ktorých intervaloch ktorý program bol spustený a ako zaťažoval procesor. Štyri čiary na grafe reprezentujú štyri jadra procesoru.

Preklad programu prebehlo relatívne rýchlo, a po tejto operácii som hneď spustil preložený binárny súbor. Z grafu je vidieť, že program len veľmi málo zaťažil procesor. Najviac času bolo strávených vykonaním inštrukcií na grafickej karte. Procesor čaká len na výsledok a počíta nové koordináty pre časticový systém.

Hneď po spustení programu v Python zaťaženosť procesoru dosiahla skoro maximálnu výšku pri jednom jadre. Počas prvých päť sekúnd, kde je záťaž najväčšia, prebieha načítanie modelov, vypočítanie normálov pre líce v modeloch a načítanie textúr. Dôsledkom toho je aj to, že prvá vykreslená snímka sa objaví na obrazovke až po piatich sekundách. Nejaké oneskorenie je možné si všimnúť aj pri programe v C, ale v tomto prípade oneskorenie nie je také výrazné. V grafe sa to ani neobjavuje.

Počas behu Python interpretu zaťaženosť procesoru bola stále veľmi vysoká. Nepomohlo ani vypnutie časticového systému (asi počas posledných piatich sekúnd). V tomto prípade grafická karta musí čakať na inštrukcie od procesoru, čo vysvetľuje prečo zmena rozlíšenia nemá veľký vplyv na rýchlosť programu.



Obr. 4.20: Zátaž pamäti

V ďalšom kroku som testoval spotrebu pamäti. Tento test nepriniesol žiadne prekvapenia. Programy v Pythonu potrebovali o 30 MB viac pamäti kvôli interpretu.

Okrem záťaže procesoru a časticového systému program implementovaný v programovacom jazyku Python bol v priemeru dvakrát pomalší ako alternatívny C program. Za daných okolností môžeme to považovať za dobrý výsledok. Možné spôsoby zrýchlenia programu a zníženia zaťaženia procesoru by boli: prepísanie niektorých častí do jazyku C, použitie shaderov a ďalšie optimalizácie Python kódu.

4.10 Profilovanie

Profilovanie je jedna z tých vecí, ktorej sa málo venuje, ale každý by to mal používať. Profilovanie je vlastne nástroj, ktorý určí, že v ktorej časti kódu koľko času strávi program. Obecne sa to dá použiť na zrýchlenie programu. V prípade Pythonu je to zvlášť užitočné, keďže tento jazyk je rádovo pomalší než napríklad jazyk C. V Pythonu k tomu slúži modul `cProfile`, ktorý je súčasťou štandardnej knižnice. Po povolení profilovania pomocou `cProfile` každé volanie funkcie je zabalené vo vnútri interpretu a bude podobné nasledujúcemu kódu:

```
t = time.time()
try:
      function()
finally:
      delta = time.time() - t
```

Delty sú potom sčítané a uložené pre každú funkciu zvlášť. `cProfile` okrem počtu volaní funkcie a času stráveného k tomu uloží aj informácie o funkcii, v ktorej sa nachádza volaná funkcia. Je to užitočné napríklad na zistenie toho, či daná funkcia zaberala t_1 času, keď bola volaná z funkcia A, a t_2 času, keď bola volaná z funkcie B. Pomocou profilovania je teda možné identifikovať tie časti kódu, kde stratíme najviac času.

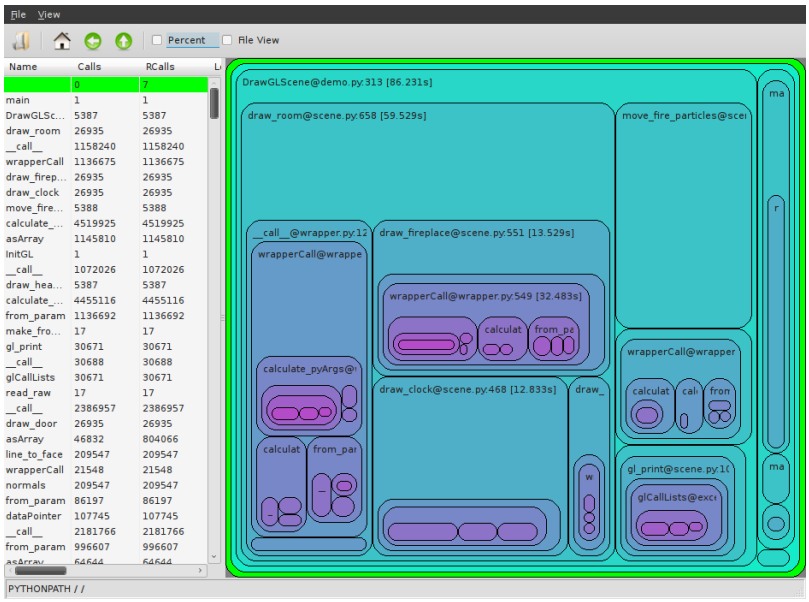
Z výstupu profilovania je možné získať dáta pomocou modulu `pStats`. Výsledky výpisu použitím modulu `pStats` sú znázornené na obrázku 4.21. Výsledky na obrázku sú ešte z verzie programu pred optimalizáciou. Je možné si všimnúť, že program strávi veľa času vo funkciách `asArray`, ktoré vytvárajú pole pre nízko-úrovňové funkcie OpenGL API z parametrov, ktoré neobsahujú kompatibilnú kópiu dát. Príkladom môžu byť aj PyOpenGL

```
>>> a.strip_dirs().sort_stats(2).print_stats(20)
Thu May 14 21:23:36 2009    profile.out

         22205771 function calls (20744968 primitive calls) in 48.523 CPU seconds

   Ordered by: cumulative time
   List reduced from 458 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.185    0.185   48.523   48.523 demo.py:863(main)
     1864    1.929    0.001   41.025    0.022 demo.py:274(DrawGLScene)
     9320    5.513    0.001   29.126    0.003 scene.py:642(draw_room)
   365382    0.621    0.000   19.909    0.000 wrapper.py:1279(__call__)
   357912    5.018    0.000   18.678    0.000 wrapper.py:549(wrapperCall)
  1441098    1.199    0.000   10.002    0.000 wrapper.py:320(calculate_pyArgs)
   371822    1.069    0.000    9.125    0.000 arraydatatype.py:53(asArray)
   344859    0.275    0.000    8.394    0.000 converters.py:114(__call__)
1772732/312153    3.868    0.000    7.194    0.000 lists.py:114(asArray)
        1    0.090    0.090    7.185    7.185 demo.py:121(InitGL)
     9320    2.823    0.000    6.564    0.001 scene.py:459(draw_clock)
     9320    1.426    0.000    6.160    0.001 scene.py:537(draw_fireplace)
       17    1.972    0.116    5.653    0.333 tools.py:118(make_from_file)
     1865    4.553    0.002    4.574    0.002 scene.py:381(move_fire_particles)
       17    0.229    0.013    3.195    0.188 tools.py:103(read_raw)
  3725940    2.495    0.000    2.668    0.000 error.py:168(glCheckError)
   312153    1.053    0.000    2.482    0.000 lists.py:141(dimensions)
     1864    0.155    0.000    2.312    0.001 scene.py:796(draw_header)
    90893    0.120    0.000    2.257    0.000 lazywrapper.py:7(__call__)
    12037    0.428    0.000    1.998    0.000 scene.py:100(gl_print)
```

Obr. 4.21: pStats príklad

funkcie `*v`, ktorým som predával jednoduché pythonovké zoznamy. Po výmene týchto zoznamov s alternatívnymi ctypes polami program sa značne zrýchlil.

Na grafické reprezentovanie výsledkov profilovania je možné použiť aj programy ako KCacheGrind alebo RunSnakeRun. Profil výsledného programu je znázornený na obrázku. Na ľavej strane sú funkcie zoradené podľa celkovej dĺžky času, ktorý program v týchto funkciách strávil. Tieto výsledky už pochádzajú z výsledného programu. Podľa očakávaní čas strávený vo funkcii `asArray` sa znížil. V zoznamu sa táto funkcia dostala až za funkcie `draw_clock, draw_fireplace, move_fire_particles`, v ktorých program strávil podstatne menej času ako v predchádzajúcom prípade.



Obr. 4.22: RunSnakeRun profilovanie

Toto je len jeden z príkladov úspešného použitia profilovania. Keďže som počas implementovania programu použil nástroj profilovania viackrát, považoval som za vhodné o tom aj zmieniť sa. Často výhody a nevýhody použitia daného programovacieho jazyka na nejaký účel neleží v samotných schopnostiach daného jazyka, napríklad že ako rýchlo vykoná určitý kód, ale v tých nástrojoch, ktoré je možné použiť na vývoj aplikácií. Toto platí viacnásobne pre aplikácie s účelom vyučovania.

Kapitola 5

Záver

Cieľom tejto práce bolo navrhnúť testovaciu aplikáciu, na ktorej je možné demonštrovať možnosti jazyka Python v počítačovej grafike. Výsledkom tejto snahy je aplikácia, ktorá v sebe zahrňuje technológie a efekty, ktoré sú bežne používané pri vytvorení akejkoľvek grafickej aplikácie: textúrovanie, práca so zložitými modelmi, tiene, časticový systém atď. K ďalším cieľom práce patril aj porovnanie výhod a nevýhod interpretovaného prístupu oproti klasickému programovaniu, napríklad v jazyku C. Na tento účel demonštračná aplikácia bola implementovaná v jazykoch Python a C. Výsledky testovaní programov ukázali, že Python je asi dvakrát pomalší, nevhodný na implementáciu časticových systémov, a že veľmi výrazne zaťažuje procesor v porovnaní s C aplikáciou. Na druhej strane možno povedať, že polovičná rýchlosť z pohľadu Pythonu nie je až taká katastrofálna. Ďalšími výhodami jazyka Python je: obrovská štandardná knižnica, široký výber nástrojov, ktoré pomôžu aj pri implementácii a neskoršie aj pri optimalizácii. Python je jazyk vyššej úrovne, a má všetky výhody súvisiace s tým, k čomu patrí: jednoduchšia práca napríklad s reťazcami, zoznamami, zložitými štruktúrami, rýchlejší vývoj, jasnejší a kratší kód.

Celkovo podľa môjho názoru Python v dnešnom stave nie je vhodný na využitie v profesionálnych aplikáciách kvôli nadmerného využitiu procesoru. Na vzdelávacie účely alebo pri začiatku vývoja je však úplne vyhovujúci. Má jednoduchší syntax a nástroje, ktoré urýchľujú a uľahčujú prácu pri programovaní. Keďže PyOpenGL API obecne príliš nelíši od OpenGL C API, vytvorený OpenGL kód je možné znovu použiť pri implementácii v C/C++.

Možným pokračovaním práce by bola implementácia ďalších technológií bežných pri vývoji grafických aplikácií (napríklad shadery antialiasing atď.), či ďalšia optimalizácia – prepísanie pomalých častí kódu do jazyk C s využitím Python C API alebo pomocou SWIG.

S príchodom OpenGL špecifikácie 3.0 a 3.1 došlo k obrovským zmenám v API.([15]) Napríklad bude nutné použiť VBO (Vertex Buffer Objects) namiesto display listov, či pervertex operácií. Ďalej bude treba manuálne počítať všetky transformačné matice a eventuálne použiť len shadery. Cieľom týchto zmien bolo zjednodušiť API hlavne pre vývojárov hier, a ďalej k tomu, aby model OpenGL viac zodpovedal dnešným grafickým kartám. Ako pokračovanie tejto práce by bolo zaujímavé sledovať aj dopad týchto zmien na PyOpenGL.

Literatúra

[1] PythonSpeed - PythonInfo Wiki [online]. `http://wiki.python.org/moin/PythonSpeed`, posledná modifikácia: 2009-05-15 [cit. 2009-5-16].

[2] PythonSpeed/PerformanceTips – PythonInfo Wiki [online]. `http://wiki.python.org/moin/PythonSpeed/PerformanceTips`, posledná modifikácia: 2009-02-04 [cit. 2009-5-10].

[3] Burian, T.: Algoritmy pro zobrazování stínů ve scéně, 2005, ročníkový projekt, Vysoké učení technické v Brně Fakulta informačních technologií.

[4] Fletcher, M. C.: PyOpenGL pre programátorov OpenGL [online]. `http://pyopengl.sourceforge.net/documentation/opengl_diffs.html`, posledná modifikácia: 2009-04-01 [cit. 2009-05-06].

[5] Fletcher, M. C.: Introduction to Python Profiling. In *PyCon*, 2009, `http://us.pycon.org/2009/conference/schedule/event/15/`.

[6] Harms, D. D.: *Začínáme programovat v jazyce Python*. Computer Press, 2008, ISBN 978-80-251-2161-0.

[7] Kilgard, M. J.: Shadow Mapping with Today's OpenGL Hardware. In *SIGGRAPH*, 2002.

[8] Martz, P.: *OpenGL röviden*. Kiskapu Kiadó, 2007, ISBN 978-96-396-3725-2.

[9] Molofee (Nehe), J.: Nehe Productions: OpenGL Lesson #19 Particle Engine Using Triangle Strips [online]. `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19`.

[10] Molofee (Nehe), J.: Nehe Productions: OpenGL Lesson #24 Bump-Mapping, Multi-Texturing & Extensions [online]. `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=24`.

[11] Molofee (Nehe), J.; D'Agata, G.: Nehe Productions: OpenGL Lesson #17 2D Texture Font [online]. `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=17`.

[12] Neider, J.; Davis, T.; Woo, M.: *The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Publishing Company, 1994, ISBN 0-201-63274-8.

[13] Projects, P.: Shadow Mapping Tutorial [online]. `http://www.paulsprojects.net/tutorials/smt/smt.html`.

[14] Rost, R. J.: *OpenGL®Shading Language, Second Edition*. Addison-Wesley Professional, 2006, ISBN 0-321-33489-3.

[15] Segal, M.; Akeley, K.: The OpenGL®Graphics System: A specification (Version 3.1 – March 24, 2009). `http://www.opengl.org/registry/doc/glspec31.20090324.pdf`.

Dodatok A

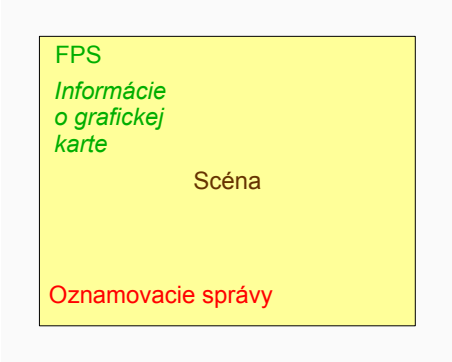
Obsah CD

- *sources* – Zdrojové súbory
  - *c* – moduly C implementacie
  - *models* – použité modely v scéne
  - *pyopenglDemo* – moduly Python impelementácie
  - *textures* – použité textúry
  - demo.exe – spustitelný Windows súbor (32 bitový)
  - demo.pyw – spustitelný Python modul
  - makefile
- *doc* – Dokumentácia
  - *tex* – TEX súbory
  - technicka_sprava.pdf – Technická správa
  - tests.ods – výsledky testov vo formáte OpenOffice.org Calc
- *videos* – Videá kvôli vysokým požiadavkám programu na hardver
  - demo.mkv – Demo aplikácie
  - test1.mkv – Test 1
  - test2.mkv – Test 2
  - test3.mkv – Test 3
  - test4.mkv – Test 4
  - test5.mkv – Test 5
  - test6.mkv – Test 6
- README

Dodatok B

Manual

Program v C pred spustením je treba preložiť pomocou make. Po spustení programu (demo[|.exe|.pyw]) sa objaví okno aplikácie:



Obr. B.1: Rozloženie v okne

Na pohyb v scéne je možné použiť myš.

Ľavé tlačítko Otáčanie

Stredné tlačítko Zoom

Na ovládanie programu je možné použiť buď kontext menu (pravé tlačítko na myši), alebo klávesové skratky. Klávesové skratky:

A Otáčanie svetla

S Tiene

F Celá obrazovka

W Zobrazenie v okne

T Textúry - zapnutie, vypnutie

I Filter textúry - GL_NEAREST, GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST

G Hmla

R Odlesky

P Zmena metódy pri časticového systému

N Počet častíc (0, 100, 200, 300, 400, 500)

V Informácie o grafickom hardvere

M Oznamovacie správy

C Kamera

L Osvetlenie

F1 – F5 Testy 1 – 5

F10 Test 6

ESC Ukončiť aplikáciu

←, → Otočanie kamery okolo osy Y

↑, ↓ Zoom

PageUp, PageDown Otáčanie kamery okolo osy X

1, 3 Posunutie svetla po ose X

0, 5 Posunutie svetla po ose Y

7, 9 Posunutie svetla po ose Z

4, 6 Otočenie svelta okolo Y

2, 8 Otočenie svetla okolo X

Dodatok C

Výsledky testov

C.1 Linux

Tabuľka C.1: Základná scéna

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	562	431	440	436
Test 2	994	441	581	448
Test 3	1355	441	757	450
Test 4	740	433	570	445
Test 5	941	437	665	450
Test 6	754	426	542	440

Tabuľka C.2: Osvetlenie

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	563	412	438	422
Test 2	995	419	581	428
Test 3	1337	423	756	433
Test 4	738	415	569	427
Test 5	942	419	665	432
Test 6	754	413	542	424

Tabuľka C.3: Odlesky

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	376	249	301	251
Test 2	589	253	375	254
Test 3	1082	256	636	258
Test 4	395	249	297	251
Test 5	653	254	475	256
Test 6	492	250	366	252

Tabuľka C.4: Textúry

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	555	330	424	337
Test 2	919	337	552	341
Test 3	1205	341	740	346
Test 4	925	334	538	339
Test 5	932	336	635	344
Test 6	747	331	527	338

Tabuľka C.5: Tiene

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	142	109	117	110
Test 2	220	111	147	112
Test 3	280	113	183	113
Test 4	175	110	144	111
Test 5	243	112	184	112
Test 6	191	111	149	111

Tabuľka C.6: Časticový systém

	C (VA)	Python (VA)	C (DL)	Python (DL)	C (priamy)	Python (p)
0	919	337	916	335	914	337
100	908	257	899	256	907	217
200	915	217	851	208	900	161
300	891	189	822	178	883	129
400	865	167	777	154	847	107
500	835	149	747	136	826	92
Test 6	717	149	641	135	709	92

Tabuľka C.7: Časticový systém, odlesky

	C	Python	C, Display Lists	Python, Display Lists
100	554	159	537	145
300	518	130	474	98
500	486	109	425	75

Tabuľka C.8: Všetko zapnuté

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	123	70	98	60
Test 2	183	71	120	65
Test 3	263	72	174	72
Test 4	144	71	116	71
Test 5	210	71	153	72
Test 6	165	71	128	71

C.2 Microsoft Windows Vista

Tabuľka C.9: Základná scéna

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	563	485	427	427
Test 2	1007	498	568	514
Test 3	1433	506	731	519
Test 4	738	490	554	504
Test 5	950	498	645	517
Test 6	760	490	528	482

Tabuľka C.10: Osvetlenie

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	563	479	429	427
Test 2	1006	492	567	503
Test 3	1430	499	730	511
Test 4	738	484	554	500
Test 5	950	494	646	508
Test 6	757	484	527	478

Tabuľka C.11: Odlesky

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	376	288	297	294
Test 2	592	295	370	301
Test 3	1117	304	618	305
Test 4	396	288	292	291
Test 5	655	298	465	303
Test 6	493	292	360	292

Tabuľka C.12: Textúry

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	549	383	414	393
Test 2	994	402	538	411
Test 3	1397	409	711	417
Test 4	728	396	526	409
Test 5	938	402	614	413
Test 6	748	393	513	403

Tabuľka C.13: Tiene

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	142	128	116	116
Test 2	221	133	147	133
Test 3	278	134	182	134
Test 4	176	131	143	132
Test 5	242	133	182	135
Test 6	192	131	147	130

Tabuľka C.14: Časticový systém

	C (VA)	Python (VA)	C (DL)	Python (DL)	C (priamy)	Python (p)
0	992	403	992	400	992	404
100	955	282	925	310	953	247
200	924	227	849	256	922	179
300	894	191	826	221	895	141
400	872	164	784	194	866	117
500	845	144	750	174	844	99
Test 6	724	143	647	172	712	98

Tabuľka C.15: Časticový systém, odlesky

	C, Vertex A.	Python, Vertex A.	C, Display Lists	Python, Display Lists
100	558	182	539	178
300	519	138	477	124
500	487	112	425	95

Tabuľka C.16: Všetko zapnuté

	C(800x600)	Python(800x600)	C(1680x1050)	Python(1680x1050)
Test 1	123	78	97	78
Test 2	183	79	119	79
Test 3	262	80	173	81
Test 4	144	78	115	79
Test 5	210	80	150	80
Test 6	165	79	127	79