UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Luka Čović

RAZVOJ GRAFIČNEGA POGONA ZA UPODABLJANJE V 2D PROSTORU

DIPLOMSKO DELO NA
UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Saša Divjak

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01515/2008
Datum: 01.09.2008

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **LUKA ČOVIĆ**

Naslov: **RAZVOJ GRAFIČNEGA POGONA ZA UPODABLJANJE V 2D PROSTORU**

DEVELOPMENT OF A GRAPHICS ENGINE FOR RENDERING IN 2D SPACE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Opišite problem razvoja grafičnega okolja za upodabljanje v 2D prostoru. Predstavite primerne odprtokodne rešitve in principe snovanja takega pogona. Podrobno predstavite teoretične osnove prikazovanja in postopke za rokovanje z objekti v 2D prostoru. Podajte rešitev zastavljenega problema in opišite praktično implementacijo pogona za nizko nivojsko grafično knjižico Direct3D9.

Mentor:

prof. dr. Saša Divjak

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Franc Solina

[Tu pride original izdane teme]

Zahvala

Zahvaljujem se mentorju prof. dr. Saši Divjaku za strokovno pomoč in nasvete pri pisanju diplomske naloge.

Zahvaljujem se tudi vsem prijateljem, še posebej pa moji Barbari in Juliju ter družini, ki so mi ves čas študija stali ob strani.

Kazalo

Seznam uporabljenih kratic in simbolov

2D	Dvodimenzionalen
3D	Trodimenzionalen
API	Application Programming Interface
BSD	Berkeley Software Distribution
CLSID	Class Identifier
COM	Component Object Model
D3DX	Direct3D Extension
Direct3D8	Direct3D Version 8
Direct3D9	Direct3D Version 9
DS	DirectShow
FPS	Frames Per Second
GNU GPL	GNU General Public License
GNU LGPL	GNU Lesser General Public License
HAL	Hardware Abstraction Layer
MIT	Massachusetts Institute of Technology
OpenGL	Open Graphics Library
SDL	Simple DirectMedia Layer
STL	Standard Template Library
VFW	Video For Windows

VMR7	Video Mixing Renderer version 7
VMR9	Video Mixing Renderer version 9

Povzetek

V diplomskem delu sem predstavil razvoj in snovanje grafičnega pogona za upodabljanje v dvodimenzionalnem prostoru.

Predstavil sem zastavljen problem, opisal podobne odprtokodne rešitve in principe snovanja takega pogona. Obstaja več odprtokodnih rešitev, ki posredno oziroma neposredno omogočajo delo z dvodimenzionalno grafiko. Le te se razlikujejo po uporabljenem programskem jeziku, po naboru funkcionalnosti, ki jo omogočajo in po načinu licenciranja.

Nato sem bolj podrobno predstavil tudi teoretične in matematične osnove v dvodimenzionalnem prostoru, prav tako pa tudi postopke za transformacijo nad objekti v takem prostoru.

V tretjem poglavju sem prikazal praktično rešitev zastavljenega problema, kjer je poudarek na prenosljivosti med operacijskima sistemoma Windows in Linux ter neodvisnosti od uporabljene nizko nivojske grafične knjižnice, kakršni sta Direct3D in OpenGL. Poleg rešitve zastavljenega problema lahko najdemo tudi opis uporabljenih tehnologij. Praktična implementacija pogona uporablja nizko nivojsko knjižnico Direct3D za upodabljanje na zaslon ter DirectShow API za upodabljanje video vsebin.

Zaključek je namenjen predvsem razmisleku o tem, kje je še prostor za optimizacijo pogona.

Ključne besede:
grafični pogon, dvodimenzionalni prostor, Direct3D, OpenGL, nizko nivojska grafična knjižnica, DirectShow, API

Abstract

This Graduate thesis provides an overview of construction and development of two dimensional graphics rendering engine.

In first chapter of the thesis proposed problem is presented, together with current open source solutions for two dimensional graphics rendering. There are quite few open source solutions which provide functionality for rendering in 2D space primarily, or expose such functionality through 3D rendering interface. These solutions differ mostly in programming language used, licence type and functionality exposed.

In following part theoretical and mathematical principles used are presented together with procedures for object transformations in two dimensional space. Transformations, such as scaling, traslation, rotation and shearing together with texture filtering procedures are thoroughly described.

In third chapter solution to proposed problem is presented, with emphasis on portability and low level graphic library independency. Low level graphic libraries referenced are Direct3D and OpenGL. Used technology for practical implementation is Direct3D for rendering on screen and DirectShow API for rendering video sources.

Last chapter proposes possiblities for further optimisation of an engine coupled with an overall conclusion of the thesis.

Key words

graphic engine, two dimensional space, Direct3D, OpenGL, DirectShow, low level graphic library, DirectShow, API

1 Uvod

Grafični pogon je programski vmesnik, ki na eni strani predstavlja abstrakcijo med nizko nivojsko grafično knjižnico, ter aplikacijo, ki ga uporablja, na drugi strani. Vsebuje skupek metod, ki omogočajo upodabljanje različnih objektov, kot so slike in tekst, pa tudi manipulacijo nad njimi. Objekti so hierarhično organizirani v drevesno strukturo, ki ji pravimo graf scen. Graf scen je skupek vozlišč, ki vsebuje podatke o objektih in definira relacije med njimi.

Razlikovati je potrebno med nizko nivojsko grafično knjižnico, kakršna sta na primer Direct3D in OpenGL, ter med grafičnim pogonom. Tako prvi kot tudi drugi omogočata upodabljanje elementov na zaslon, vendar nizko nivojska grafična knjižnica navadno ne implementira grafa scen. Kadar pogon poleg upodabljanja omogoča tudi širšo funkcionalnost, na primer predvajanje zvoka, je ta razlika jasno razvidna, medtem ko je v ostalih primerih lahko precej zabrisana. Tak primer je SDL, ki je kljub precej širokemu naboru funkcionalnosti grafična knjižnica in ne grafični pogon.

Enostavno vzdrževanje in zmožnost nadgrajevanja funkcionalnosti programskega vmesnika z vmesnim preverjanjem optimalnosti ter hitrosti programske kode med samim snovanjem je bilo osnovno vodilo pri implementaciji pogona. Cilj je bil implementacija pogona, ki bi bil enostaven za uporabo, obenem pa bi bil dovolj zmogljiv, tudi za upodabljanje zahtevnejših vsebin, kot so video posnetki in živa slika iz video zajemalnika. Pomembna je bila tudi prenosljivost programske kode na operacijskih sistemih Windows in Linux.

V diplomskem delu bom predstavil snovanje takega pogona, težave s katerimi sem se soočil tekom implementacije, pa tudi tehnologije, ki so bile pri snovanju uporabljene.

1.1 Nizko nivojska grafična knjižnica

Nizko nivojska grafična knjižnica nam predstavlja abstrakcijo med strojno opremo (grafično kartico) in aplikacijo, ki jo uporablja. Omogoča enovit dostop do funkcionalnosti strojne opreme različnih proizvajalcev. Danes sta najpogosteje uporabljeni 3D knjižnici Direct3D na operacijskem sistemu Windows in OpenGL na sistemih GNU/Linux, UNIX in Mac OS X.

Direct3D je 3D programski vmesnik, ki se uporablja izključno na operacijskih sistemih podjetja Microsoft. Razlog je predvsem ta, da gre za lastninski programski vmesnik, kjer programska koda, ki bi omogočala prenosljivost na druge sisteme, ni prosto dostopna. Obstaja sicer nekaj poizkusov, da bi knjižnico prenesli na operacijski sistem Linux, vendar z mešanimi rezultati. Težava je v tem, da v zasnovi Direct3D uporablja še cel kup drugih knjižnic in funkcionalnosti sistema Windows.

OpenGL je za razliko od Direct3D odprt standardni programski vmesnik, zato je tudi najpogosteje uporabljen 3D programski vmesnik na operacijskih sistemih, ki ne bazirajo na sistemu Windows. Predvsem zaradi razširjenosti, pa tudi dejstva, da teče na vseh najpogostejših operacijskih sistemih, kar omogoča razmeroma enostavno prenosljivost kode, se zanj odloča vedno več programerjev.

1.2 Obstoječe odprtokodne rešitve

Princip odprte kode je pristop k razvoju programske opreme. Zanj je značilen decentraliziran način razvoja. Programska koda programa oziroma programskega vmesnika je javno dostopna. Pod kakšnimi pogoji uporabnik lahko to programsko kodo uporablja, je določeno z pogoji licenciranja. Poznamo več vrst licenc, kot so na primer BSD, GNU GPL, GNU LGPL in MIT odprtokodna licenca.

Obstaja kar nekaj sorodnih rešitev, razlikujejo pa se predvsem po kompleksnosti in načinu licenciranja. Nekatere izmed opisanih knjižnic so primarno sicer namenjene 3D upodabljanju, vendar podpirajo tudi upodabljanje 2D elementov. Omeniti velja, da so nekatere izmed njih celotni igralni pogoni, kar pomeni, da poleg upodabljanja grafike vsebujejo tudi podporo za predvajanje zvoka, knjižnice s fizikalnimi modeli in omrežno podporo.

1.2.1 OGRE 3D

Ogre 3D je primarno 3D pogon, vendar lahko upodabljamo tudi 2D elemente. Je neodvisna od 3D nizko nivojske knjižnice (OpenGL, Direct3D). Podpira operacijske sisteme Linux, Windows in Mac OS X. V celoti je objektno programirana v programskem jeziku C++. Na spletu je precej aktivna uporabniška skupnost, z veliko dostopne dokumentacije. Številna je tudi razvijalska ekipa [1].

Knjižnica je predvsem scensko orientirana, kjer so scene med seboj hierarhično organizirane. Elementi so lahko med seboj povezani na različne načine, na primer preko vgrajenih vtičnikov (BSP, Octree), lahko pa te vtičnike napišemo tudi sami.

Licencirana je pod GNU Lesser GPL, kar pomeni, da jo lahko poljubno uporabljamo, vendar moramo pri distribuciji programa zraven posredovati izvorno kodo knjižnice, prav tako vse morebitne popravke na njej. V primeru, da je program na voljo na spletu, moramo ponuditi povezavo, kjer se izvorno kodo knjižnice lahko pretoči.

1.2.2 SDL

SDL (Simple DirectMedia Layer) je odprtokodna večplatformna multimedijska knjižnica. Napisana je v programskem jeziku C, vendar jo lahko uporabljamo tudi v programskem jeziku C++. Podpira več različnih operacijskih sistemov, poleg Linux in Windows okolja, tudi MacOS, Mac OS X, FreeBSD, OpenBSD in druge [2].

Primarno je namenjena delu z 2D grafiko, uporablja pa se jo v igrah, emulatorjih in multimedijskih aplikacijah. Sestavljena je iz več modulov in poleg upodabljanja grafike omogoča delo z omrežji, dogodki, nitmi, omogoča pa tudi predvajanje zvoka. Uporablja OpenGL oziroma Direct3D nizko nivojsko grafično knjižnico.

Distribuirana je pod GNU LGPL licenco.

1.2.3 Irrlicht

Irrlicht je podobno kot Ogre primarno 3D knjižnica, ki pa podpira tudi delo z 2D grafiko. Primarno je namenjena programskemu jeziku C++, čeprav obstajajo ovojnice, ki omogočajo programiranje tudi v drugih programskih jezikih, kot so C#, Visual Basic in Java. Je več

platformna, podpira pa nizko nivojske grafične knjižnice, kot so Direct3D8, Direct3D9 in OpenGL [3].

Distribuirana je pod licenco, ki temelji na zlib/libpng licenci. Knjižnico lahko uporabljamo poljubno, tudi v komercialnih produktih, brez omejitev, kar pomeni, da nam dejstva, da uporabljamo to knjižnico, ni potrebno eksplicitno navajati, prav tako ni potrebno priložiti izvorne kode. Ta licenca je bolj ohlapna od GNU LGPL licence.

2 Prostor

Kartezijski koordinatni sistem v ravnini $\mathbb{R}^2$ je definiran s parom medseboj pravokotnih osi. Horizontalna os ima oznako x, vertikalna os pa oznako y. Vsako točko v prostoru lahko torej predstavimo s parom (x,y). Točka (x=0, y=0) nam predstavlja koordinatno izhodišče.

Nekaj lastnosti točk in vektorjev v ravnini:

- Vektor v ravnini je definiran kot:

$$\vec{V} = (x, y) \tag{2.1}$$

- Dolžina vektorja v ravnini:

$$|V| = \sqrt{x^2 + y^2} \tag{2.2}$$

- Normalizacija vektorja:

$$V_n = \frac{V}{\|V\|} \tag{2.3}$$

- Razdalja med dvema točkama $T_1(x_1, y_1)$ in $T_2(x_2, y_2)$:

$$|T_1 T_2| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2.4}$$

2.1 Afina transformacija

Afina transformacija med dvema vektorskima prostoroma je linearna transformacija, kateri sledi translacija [4]:

$$\vec{x} \rightarrow A\vec{x} + \vec{b} \tag{2.5}$$

kjer nam A predstavlja matriko transformacije in b vektor translacije.

Geometrično gledano, je taka transformacija v prostoru tista, ki ohranja :

- kolinearnost med točkami
- razmerje razdalj odsekov na premici: če so točke P1, P2, P3 kolinearne, potem se razmerje $\frac{|P2 - P1|}{|P3 - P2|}$ ohranja

Najbolj pogoste transformacije so translacija, rotacija, skaliranje, striženje in zrcaljenje. Te operacije so med seboj heterogene, saj imamo pri translaciji matrično seštevanje, pri ostalih

operacijah pa matrično množenje. Če želimo te operacije združiti v eno, kompleksno operacijo, moramo dimenzijo koordinatnega sistema povečati za ena. Dobimo tako imenovani homogeni koordinatni sistem. V dvo dimenzionalnem prostoru tako dobimo matriko transformacije velikosti 3x3.

Z vpeljavo homogenega koordinatnega sistema so vse transformacije med seboj homogene, saj so vse tipa matrično množenje. Pri operacijah moramo biti pozorni na komutativnost, saj matrično množenje ni komutativno. Primeri med seboj komutativnih operacij:

- translacija - translacija
- rotacija - rotacija
- skaliranje - skaliranje
- skaliranje (uniformno, Sx = Sy) - rotacija
- striženje - striženje

Vrstni red transformacij je torej pomemben in vpliva na končni rezultat.

2.1.1 Translacija

Translacija je najosnovnejša transformacija. Pri translaciji se točka (x,y) linearno preslika v novo točko (x',y') preko vektorja $\vec{T}=(T_x, T_y)$. Matrika translacije za vektor T:

$$M_T=\begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \qquad (2.6)$$

2.1.2 Rotacija

Rotacija je sprememba orientacije objekta, oziroma njegove množice točk. Komutativna je sama z seboj in v kombinaciji s skaliranjem. Matrika rotacije za kot Θ:

$$M_R=\begin{bmatrix} \cos(\Theta) & -\sin(\Theta) & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (2.7)$$

Zaporedne rotacije lahko tudi seštevamo. Matrika rotacije za kot Θ in za kot Φ:

$$M_R = \begin{bmatrix} \cos(\Theta)+\cos(\Phi) & -\sin(\Theta)-\sin(\Phi) & 0 \\ \sin(\Theta)+\sin(\Phi) & \cos(\Theta)+\cos(\Phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.8)$$

2.1.3 Skaliranje

Naslednja transformacije je skaliranje. Skaliranje je spreminjanje velikosti objekta. Če je faktor S manjši od ena, objekt pomanjšamo, če je faktor večji od ena, predmet povečamo. Kadar je faktor enak ena, se objekt ne spremeni. Ko velja enakost $s_x = s_y$, je skaliranje uniformno, kjer se objekt spremeni sorazmerno po obeh oseh. Matrika skaliranja za faktor S:

$$M_S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.9)$$

2.1.4 Striženje

Striženje je transformacija, kjer točke na neki premici ostanejo fiksirane, ostale točke objekta pa se pomaknejo vzdolž osi, sorazmerno z razdaljo od opazovane premice. Striženje lahko uporabimo pri animaciji, uporabno pa je na primer tudi pri popravljanju perspektive slikam. Objekt lahko transformiramo po osi x in po osi y. Matrika striženja za faktor g v smeri osi x:

$$M_H = \begin{bmatrix} 1 & g & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.10)$$

Matrika striženja za faktor h v smeri osi y:

$$M_H = \begin{bmatrix} 1 & 0 & 0 \\ h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.11)$$

2.1.5 Zrcaljenje

Zrcaljenje nam tvori zrcalno sliko predmetov. Objekt lahko zrcalimo preko osi x, preko osi y, lahko pa ga zrcalimo preko obeh osi. V tem primeru se objekt zrcali preko koordinatnega izhodišča. Matrika zrcaljenja preko osi x:

$$M_T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

Matrika zrcaljenja preko osi y:

$$M_T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

Matrika zrcaljenja preko koordinatnega izhodišča:

$$M_T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

2.2 Skaliranje tekstur

V prejšnjem razdelku je bilo omenjeno skaliranje, kot ena izmed afinih transformacij. Taka transformacija se uporablja predvsem nad poligoni. Kadar tako skaliramo poligon, kot množico oglišč v 2D prostoru, ne pride do izgube informacije. Kadar pa tako skaliramo neko teksturo na zaslon, pa se zgodi, da imamo mesta, kjer vrednost piksla ni definirana, oziroma v obratnem primeru več tekslov zasede mesto enega piksla. Če na primer teksturo povečamo, se posamezni teksli sicer preslikajo v novo, večjo teksturo, vendar je med njimi praznina, saj ti piksli nimajo definirane vrednosti. To praznino lahko zapolnimo z različnimi metodami interpolacije [5].

Omenjene metode interpolacije se razlikujejo predvsem po kvaliteti tako skalirane teksture, pa tudi po računski zahtevnosti. Omeniti je potrebno, da najbolj kompleksna rešitev ne prinese vedno najboljšega rezultata. V nekaterih primerih se bolje odreže enostavnejša metoda interpolacije, ki sicer v splošnem da slabše rezultate. Podrobnejše poznavanje metod interpolacije programerju omogoča, da izbere najbolj ugodno rešitev za zastavljen problem.

V nadaljevanju sta omenjena piksel in teksel. Oba nam predstavljata koordinato točke, vendar je piksel točka na zaslonu, teksel pa predstavlja točko na teksturi. V nadaljevanju je

piksel omenjan kot rezultat filtrirane, skalirane teksture, saj se tekstura največkrat upodablja na zaslon. Seveda pa se tekstura lahko upodobi ne le na zaslon, temveč na primer tudi v neko drugo teksturo.

2.2.1 Interpolacija najbližjega soseda

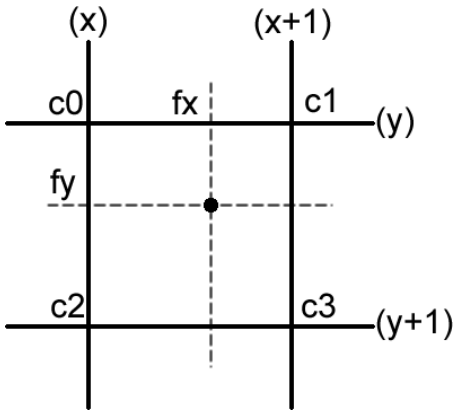
Ta metoda je ena izmed najbolj enostavnih in računsko najmanj zahtevnih. V osnovi deluje tako, da za vsak piksel določimo sorazmerno realno vrednost koordinate teksla. V teksturi nato izberemo teksel, ki je tej izračunani koordinati najbližji. Pri povečavi gre pravzaprav za replikacijo pikslov, saj se vsak teksel preslika v več pikslov. Pri pomanjšavi pa je ravno obratno, posamezne teksle se preskoči.

Metoda je sicer izjemno hitra, saj praktično ni zahtevnejših računskih operacij, vendar pa v splošnem ni najbolj ugodna. Pride namreč do neželenih pojavov, kot so preveč poudarjeni robovi, pri večji povečavi pa slika izgleda zrnata. Ta lastnost je še posebej poudarjena, saj je človeško oko precej občutljivo na robove.

Uporablja se pri upodabljanju, kjer sta hitrost in odzivnost na prvem mestu, pa tudi tam, kjer so prej omenjeni sicer v splošnem negativni pojavi zaželjeni.

2.2.2 Bilinearna interpolacija

Biliniearna interpolacija prinaša nekoliko bolj napreden pristop k interpolaciji pikslov. Vrednost piksla se izračuna na podlagi vrednosti sosednjih štirih tekslov. Gre pravzaprav za linearno interpolacijo v dveh smereh, oziroma koordinatnih oseh.



Slika 1: *Bilinearna interpolacija*

Zgornja slika nam predstavlja štiri sosednje teksle c0, c1, c2 in c3 z koordinatami c0(x, y), c1(x+1, y), c2(x, y+1) in c3(x+1, y+1). Vrednosti fx in fy sta realni vrednosti, iz katere se teksel preslika v piksel. Koordinati x in y sta celoštevilski vrednosti. Interpolirano vrednost piksla c dobimo z enačbo:

$$c = c_0 * (1 - f_x) * (1 - f_y) + c_1 * f_x * (1 - f_y) + c_2 * (1 - f_x) * f_y + c_3 * f_x * f_y \qquad (2.15)$$

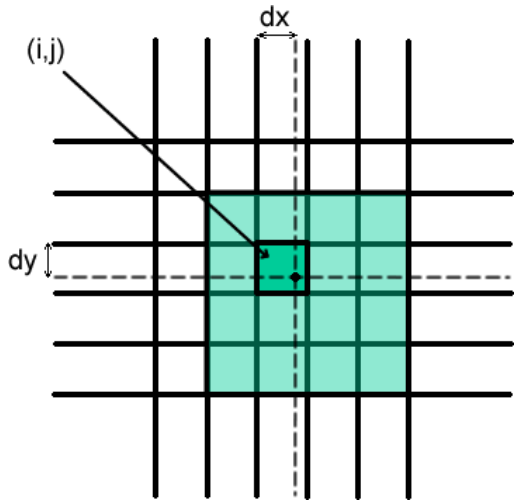
Rezultat bilinearnega filtriranja je najboljši, kadar je faktor povečave manjši od 2 in večji od 0.5. To pomeni, da pri teksturi 256x256 pikslov, skalirana tekstura ni večja od 512x512 pikslov in manjša od 128x128 pikslov. Kadar presežemo te meje, postane videz teksture preveč gladek v prvem primeru, v drugem primeru, pa pride do izgub informacije, saj se posamezni teksli preskočijo in se ne preslikajo v skalirano teksturo.

Omeniti je potrebno, da moramo pri večbarvnih teksturah, tako interpolirati vsak barvni kanal piksla.

2.2.3 Bikubična interpolacija

Pri bikubični interpolaciji opazujemo skupino šestnajst sosednjih tekslov formata 4 x 4. Obstaja več pristopov h kubični interpolaciji, tu se bom osredotočil na enega izmed najpogostejših.

Na spodnji sliki nam par (i,j) predstavlja celoštevilsko koordinato opazovanega teksla, par (dx+i, dy+j) pa realno koordinato piksla, ki se preslika iz opazovanega teksla.



Slika 2: *Bikubična interpolacija*

Spodnja formula nam poda interpolirano vrednost piksla:

$$F(x,y)=\sum_{m=-1}^{2}\sum_{n=-1}^{2}F(i+m,j+n)\,R(m-dx)\,R(dy-n) \tag{2.16}$$

Kubična utežna funkcija R(x):

$$R(x)=\frac{1}{6}\left[P(x+2)^3-4\mathrm{P}(x+1)^3+6\mathrm{P}(x)^3-4\mathrm{P}(x-1)^3\right] \tag{2.17}$$

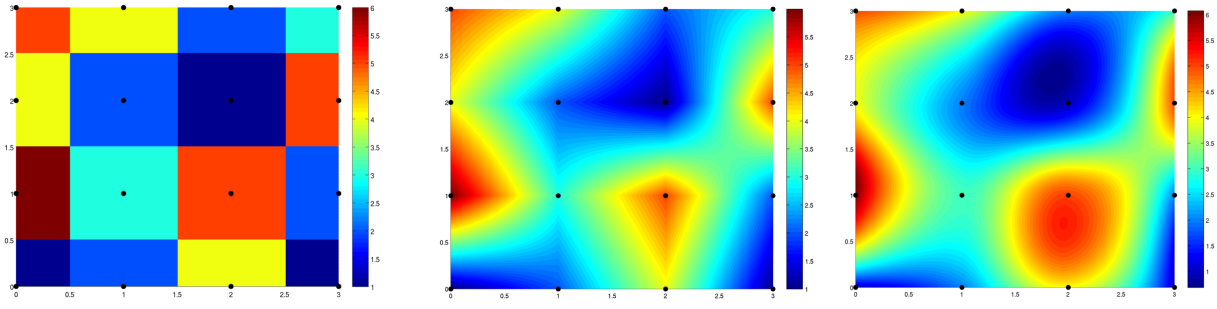
Funkcija P(x):

$$P(x)=\begin{Bmatrix}x & x>0\\ 0 & x\le 0\end{Bmatrix} \tag{2.18}$$

Bikubična interpolacija je računsko razmeroma zahtevna in se je ne uporablja tam, kjer je zahtevana visoka hitrost. Uporablja se predvsem tam, kjer je zahtevana visoka kvaliteta skalirane teksture, hitrost pa ni prvotnega pomena, oziroma zaradi same lastnosti teksture (na primer velikosti) ne pride do prevelikega časovnega pribitka pri upodabljanju. Uporabljamo jo na primer pri pripravi mipmapov, ki so opisani v razdelku 3.3.1.

2.2.4 Primerjava opisanih metod

Spodnja slika predstavlja rezultate različnih metod interpolacije. Črne točke nam prikazujejo, kam na zaslon se preslikajo teksli izvorne teksture.



Slika 3: *Rezultat različnih metod interpolacije na teksturi velikosti 4x4 tekslov (vir: Google slike)*

Kot je bilo omenjeno, se opisane metode razlikujejo predvsem po kvaliteti končnega rezultata in po hitrosti delovanja.

Najhitrejša metoda je metoda najbližjih sosedov, vendar je v večini primerov končni rezultat razmeroma slab v primerjavi z ostalimi metodami. Izjema je manipulacija tako imenovanih pikselnih slik, kjer so ostri robovi in prehodi zaželjeni. Iz slike je lepo razvidno samo delovanje metode in replikacija posameznih tekslov.

Metoda bilinearne interpolacije je nekje tri do štirikrat računsko bolj zahtevna od metode najbližjih sosedov. Na sliki pa je precej opazen nezvezen prehod med posameznimi teksli.

Najboljši rezultat nam vsekakor da metoda bikubične interpolacije, vendar je računsko približno desetkrat bolj zahtevna od najhitrejše metode.

Izbira prave metode interpolacije programerju omogoča najboljšo rešitev za zastavljen problem.

3 Grafični pogon

Kot je bilo omenjeno v uvodu, nam grafični pogon nudi precej širšo funkcionalnost od grafične knjižnice. V tem poglavju se bom osredotočil predvsem na praktično implementacijo pogona in na težave, s katerimi sem se med delom soočil.

Po pregledu podobnih odprtokodnih rešitev je bilo jasno, da nobena izmed obstoječih rešitev ne ustreza zastavljenemu problemu v celoti. Nastali pogon združuje elemente, ki jih najdemo v množici opisanih rešitev prvega poglavja. Ti so predvsem enostaven API, prenosljivost, pa tudi računska nezahtevnost.

Grafični pogon je v celoti napisan v programskem jeziku C++. Za razvoj na Windows operacijskem sistemu sem najprej uporabil razvojno okolje Microsoft Visual Studio 2003, kasneje pa sem projekt prenesel v razvojno okolje Microsoft Visual Studio 2008. Taka odločitev se je kmalu izkazala za pravilno. Izkazalo se je, da je prevajalnik v različici 2003 prikril nekaj hroščev, ki bi, oziroma so, v nekaterih primerih povzročili nedefinirano in posledično nepravilno delovanje.

3.1 Prenosljivost in neodvisnost

Cilj pri snovanju pogona je bila knjižnica, ki bi delovala tako v okolju Windows kot tudi v okolju Linux, obenem pa bi bila neodvisna od uporabljene nizko nivojske grafične knjižnice (Direct3D na Windows platformi, OpenGL na Linux platformi). Tu sem moral upoštevati dve smernici, prva je prenosljivost programske kode med Linux in Windows okoljem. Ta del je rešen z izključno uporabo tistih knjižnic, ki jih podpirata obe okolji. Druga smernica pa je neodvisnost od uporabljene nizko nivojske knjižnice. Ta del je zasnovan tako, da je koda, ki kliče metode nizko nivojskih grafičnih knjižnic za upodabljanje na zaslon, strogo ločena in enkapsulirana v razrede. Seveda je bilo potrebno pri pisanju programske kode upoštevati tudi C++ standard.

3.1.1 Prenosljivost programske kode v Linux in Windows okolju

Precej pozornosti je posvečeno temu, da se uporablja le standardne knjižnice, torej knjižnice, ki jih podpirata tako Linux kot tudi Windows okolje.

Uporabljena je bila le ena zunanja knjižnica, STL (Standard Template Library). STL je podmnožica C++ standardne knjižnice. Je edina knjižnica, poleg grafičnih knjižnic v specifični

implementaciji, ki jo grafični pogon uporablja. S tem je omogočena enostavna prenosljivost programske kode na linux platformo.

Najpomembnejša gradniki STL knjižnice so vsebovalniki ter iteratorji. Namen vsebovalnikov je, da vsebujejo oziroma hranijo ostale objekte. Na voljo je kar nekaj različnih vsebovalnikov, ki se razlikujejo predvsem po tem, na kakšen način so elementi organizirani v pomnilniku, kako se do njih dostopa, ter kakšen je njihov razred časovne zahtevnosti dostopa, dodajanja, brisanja in iskanja hranjenega elementa. Nekaj v pogonu najpogosteje uporabljenih vsebovalnikov:

- **vector:** za vektor je značilno, da se njegova velikost spreminja dinamično, čas za dostop in vstavljanje elementa na začetek ali konec je konstanten, dodajanje in iskanje elementov drugje pa zahteva linearen čas

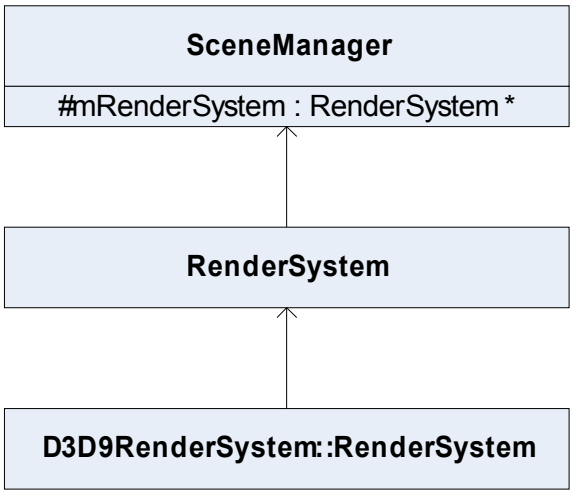
- **map:** je urejen asociativen vsebovalnik, ki vsebuje pare unikatnega ključa ter podatka. Iskanje, brisanje in vstavljanje elementa je časovne zahtevnosti O(log n).

- **list:** je implementiran v obliki dvojno povezanega seznama

Iteratorji nam omogočajo iteracijo nad prej omenjenimi vsebovalniki. Iterator je lahko veljaven skozi celotno življenjsko obdobje elementa shranjenega v vsebovalniku, lahko pa postane neveljaven ob vsaki spremembi, torej dodajanju, brisanju ali spremembi vrstnega reda elementov.

3.1.2 Neodvisnost od uporabljene nizko nivojske grafične knjižnice

Neodvisnost od nizko nivojske grafične knjižnice je rešena tako, da je logika za izrisovanje elementov na zaslon logično ločena od pogona samega. Implementacija neke druge nizko nivojske grafične knjižnice v najosnovnejši izvedbi tako zahteva le implementacijo abstraktnega razreda RenderSystem in abstraktnih razredov virov (Font, Texture).

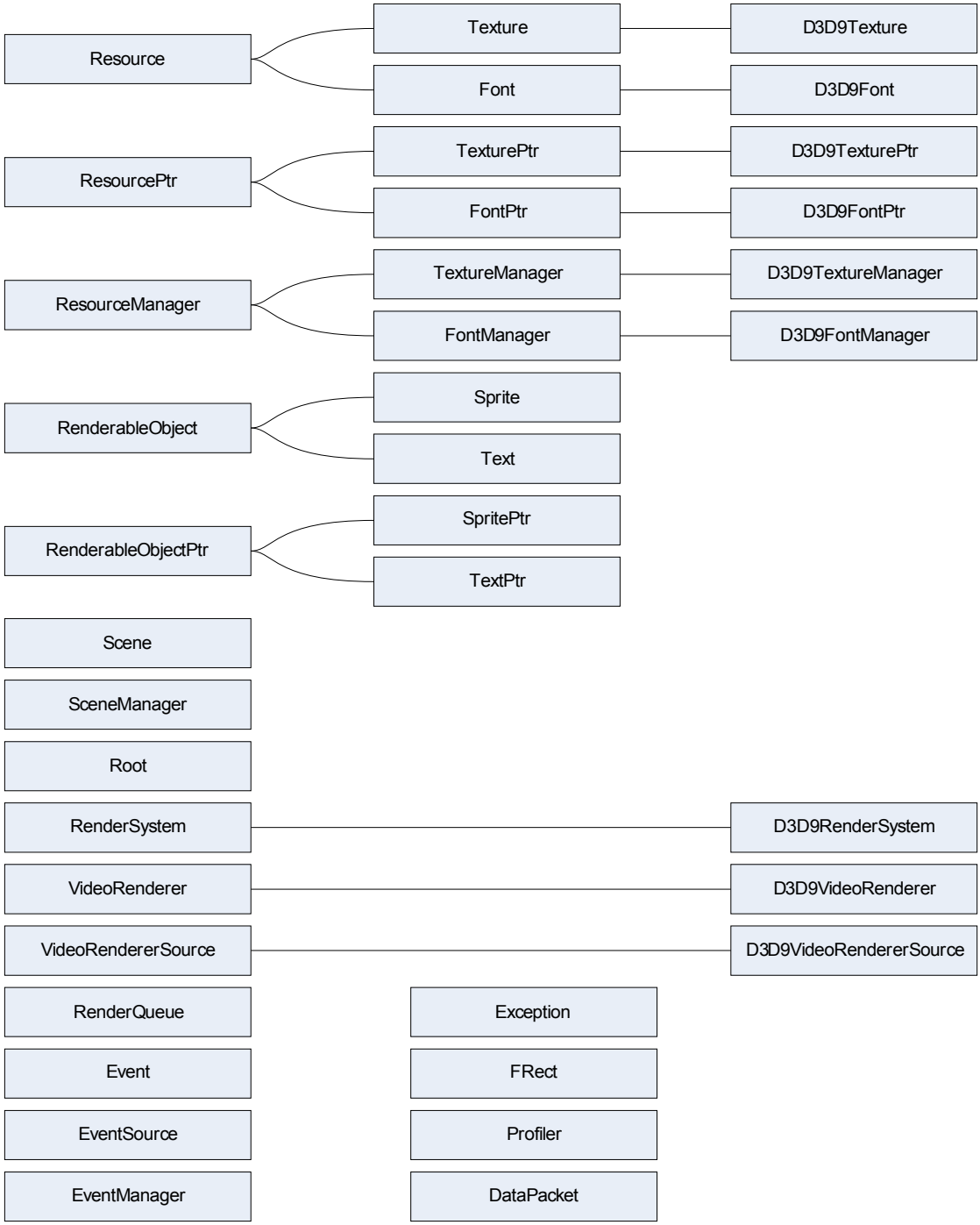


Slika 4: *D3D9 implementacija razreda RenderSystem*

Pri tem je potrebno upoštevati specifične lastnosti različnih nizko nivojskih grafičnih knjižnic. OpenGL na primer uporablja RH (RightHanded), Direct3D pa LH(LeftHanded) koordinatni sistem, kar sicer rešujemo z matričnimi transformacijami. Drug podoben problem je izhodišče teksture. V OpenGL knjižnici je koordinatno izhodišče teksture v levem spodnjem kotu, pri Direct3D pa v levem zgornjem kotu. Prav tako se pri obeh knjižnicah razlikujeta tudi programska jezika za programiranje senčilnikov. To je le nekaj poglavitnih razlik med knjižnicama. Najpomembnejše je, da je že programski tok pogona napisan tako, da avtomatično ne izključuje ne ene, ne druge.

3.2 Zgradba pogona

Spodnja shema prikazuje razredni diagram za specifično implementacijo z grafično knjižnico Direct3D9, kjer so poleg razredov virov in razreda upodobljevalnika implementirani tudi razredi za delo z video vsebinami.



Slika 5: *Razredni diagram pogona*

Pogon je implementiran v obliki statične knjižnice, v neki aplikaciji pa ga lahko uporabimo z implementacijo razreda Root:

```
class Root
{
	public:
				Root();
```

```
            virtual       ~Root();

                          void init();
                          void mainLoop();
                          void render();
            virtual       void step();
                          void destroy();
      protected:
            RenderSystem*       mRenderSystem;
            SceneManager*       mSceneManager;
            TextureManager*     mTextureManager;
            FontManager*        mFontManager;
            Profiler*           mProfiler;
};
```

Koda 1: *Zgradba razreda Root*

Iz programske kode je razvidno, da moramo za uporabo pogona v neki aplikaciji implementirati metodo step(). Kot bo to opisano kasneje, v razdelku 3.6.1, je ta metoda izhodišče celotne logike upravljanja, vodenja in prikazovanja objektov, ki jih upodabljamo na zaslonu. Dejansko upodabljanje preko upodobljevalnika pa se odvija v metodi render(), ki jo v uporabniški aplikaciji direktno sicer nikjer ne kličemo.

3.3 Viri

Viri nam predstavljajo osnovne gradnike informacij, ki se tako ali drugače upodabljajo na zaslon. V samem grafičnem pogonu lahko najdemo tri tipe virov:

- font
- tekstura
- video vir

Vsak vir se mora pred uporabo prenesti oziroma naložiti v delovni pomnilnik sistema. Pogosto je pred tem potrebno vir pretvoriti v format, ki ga razume programski vmesnik nizko nivojske knjižnice. Najlažje je, če vire predhodno pripravimo na upodabljanje, saj se tako izognemo *on-the-fly* operacijam, ki navadno predstavljajo nek časovni pribitek, če ne drugje pri nalaganju virov v pomnilnik. Različni načini hranjenja virov nam lahko precej zmanjšajo prostor, ki ga vir zaseda na trdem disku.

Vire v pogonu lahko upravljamo samo z razredi za upravljanje virov. Tak pristop omogoča enostavno dodajanje funkcionalnosti in prinese enovit pristop k upravljanju z viri v pomnilniku. Izkazalo se je, da tak pristop tudi precej zmanjša število neželenih hroščev.

3.3.1 Teksture

Teksture v pogonu nam predstavljajo bitne slike, ki se upodobijo na zaslon. Kot je bilo že prej omenjeno je tekstura na trdem disku lahko shranjena v različnih formatih.

Uporabljen grafični format moramo izbrati glede na namen uporabe. Vsi formati namreči ne podpirajo precej pogosto uporabljene transparence (nimajo alfa kanala) in mipmapov. Ločiti je potrebno tudi med izgubnimi in breizgubnimi formati. Tu je potrebno skleniti kompromis. Nekateri grafični formati sicer resda zasedejo razmeroma malo prostora na trdem disku, vendar se lahko dlje časa nalagajo v grafični pomnilnik. To je še posebej izrazito pri teksturah, kjer potrebujemo mipmape, grafični format pa jih ne podpira. Takrat jih moramo generirati sproti pri nalaganju teksture.

Mipmapi

Mipmapi so pomanjšave neke izvorne teksture, ki se navadno shranijo poleg izvorne teksture. Poleg izvorne teksture imamo več nivojev pomanjšanih tekstur. Vsak nivo, je za pol manjši v višino in širino od prejšnjega nivoja, tako vse dokler velikost naslednjega nivoja ni več večkratnik števila dve. Velikosti mipmapov teksture velikosti 512x512 tekslov so torej:

Nivo	Velikost	Št. tekslov
Izvorna tekstura	512x512	262144
1.nivo	256x256	65536
2.nivo	128x128	16384
3.nivo	64x64	4096
4.nivo	32x32	1024
5.nivo	16x16	256
6.nivo	8x8	64
7.nivo	4x4	16
8.nivo	2x2	4
9.nivo	1x1	1

Tabela 1: *Velikost mipmapov teksture velikosti 512x512 tekslov*

Iz tabele je razvidno, da je vsak nivo za četrtino manjši od prejšnjega. Iz tega lahko izračunamo, da je nekomprimirana tekstura z dodanimi mipmapi približno za tretjino večja

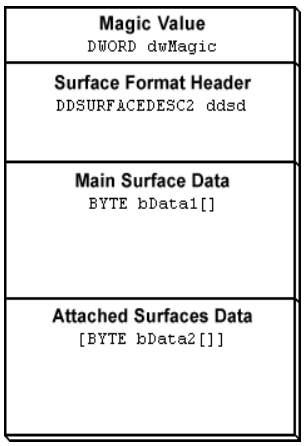
od tiste brez dodanih mipmapov. Spodnja slika nam prikazuje tako teksturo z dodanimi mipmapi:

Slika 6: *Primer mipmapov (vir: Google slike)*

Kadar želimo koristiti prednosti mipmapov in jih naš grafični format ne podpira, jih moramo kreirati *on-the-fly*. Kadar tako nalagamo večje število tekstur, je lahko časovni pribitek precejšen. Največji časovni pribitek pri kreiranju mipmapov je skaliranje in posledično filtriranje. Najlažje je, če uporabim grafični format, ki podpira mipmape, na primer DDS.

DDS format

V pogonu je uporabljen format DDS (DirectDraw Surface). Podpira prej omenjene mipmape, transparenco, pa tudi kompresijo. V tem formatu lahko poleg tekstur shranjujemo tudi kubne teksture, volumenske teksture in polja tekstur. V osnovi je DDS binarna datoteka, ki vsebuje glavo in podatke :



Slika 7: *Vsebina .dds datoteke (vir: MSDN)*

- **Magic Value:** nam označuje verzijo. Format DDS se namreč v Direct3D verziji 9 in 10 razlikujeta.
- **Surface Format Header:** je struktura, ki nam opisuje shranjeno teksturo in vsebuje podatke kot so: velikost teskture, bitna širina teksture, format pikslov, število mipmapov, kadar gre za volumenske teksture tudi podatek o globini.
- **Main Surface Data:** tu so shranjeno dejanski podatki o posameznih pikslih teksture
- **Attached Surfaces Data:** vsebuje podatke o dodatnih površinah, kot so na primer mipmapi

Format pikslov

Obstaja več različnih načinov zapisa podatkov o pikslih oziroma prej omenjenih formatih pikslov. Razlogov za uporabo različnih formatov pikslov je več. Nekaterih so predvsem historične narave, nekateri so pa pogojeni s tehnologijo, kjer se jih največ uporablja. Tu se bom osredotočil le na tista, s katerima sem se največkrat srečal:

- RGB
- YUV

RGB formati so lahko sestavljeni iz treh različnih barvnih kanalov. Red, Green in Blue nam predstavljajo posamezne barve v kanalu. Tem trem kanalom imamo lahko dodan tudi alfa kanal, ki je navadno označen s črko A. Včasih imamo poleg tudi oznako X. Le ta nam predstavlja piksle za poravnavno, ki nimajo nekega posebnega pomena. Format je navadno opisan s kombinacijo črke (R,G,B,A,X) in številke, ki nam predstavlja število bitov za ta kanal. Na primer R8G8B8A8. Ta format ima 8 bitni R kanal, 8 bitni G kanal, 8 bitni B kanal in 8 bitni A kanal. Razvidno je tudi, da vsak piksel zasede 4 bajte pomnilnika (32bitov). Nekaj pogostejših formatov:

- R8G8B8 (24bit)
- A8R8G8B8 (32 bit)
- X8R8G8B8 (32 bit)
- R5G6B5 (16 bit)
- X4R4G4B4 (16 bit)

YUV formate lahko razdelimo v dve skupine. Prva je tista, kjer si vrednosti komponent posameznih pikslov sledijo v sosledju, pri drugi pa so posamezne komponente pikslov ločene v poljih. Pri formatu YUV uvedemo pojma luminanca in krominanca. Krominanco nam predstavljata komponenti U in V, luminanco pa kanal Y. Kanal Y je pravzaprav svetlost, kanala U in V pa sta razliki (R – Y) in (B - Y).

Med YUV in RGB prostorom velja naslednja relacija [6]:

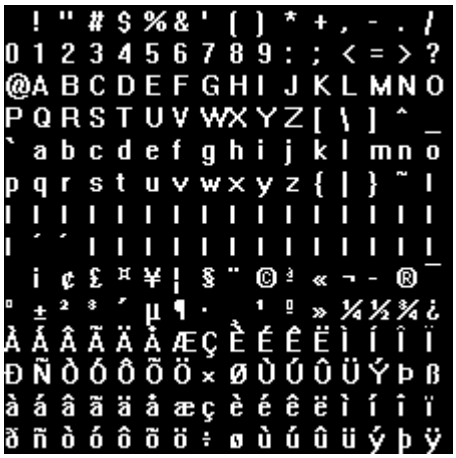
$$
\begin{aligned}
Y &= ((66 * R + 129 * G + 25 * B + 128) \gg 8) + 16 \\
U &= ((-38 * R - 74 * G + 112 * B + 128) \gg 8) + 128 \\
V &= ((112 * R - 94 * G - 18 * B + 128) \gg 8) + 128
\end{aligned}
\tag{3.1}
$$

Razlogov za uporabo YUV formata je več. Eden izmed njih je lastnost človeškega očesa, ki je precej bolj občutljivo na luminano kot na krominanco, kar je lastnost uporabna pri kompresiji.

3.3.2 Fonti

V pogonu poznamo dva tipa fontov. Prvi je font, ki je že predhodno upodobljen na neko bitno sliko, drugi tip pa je truetype font.

Za font, ki je predhodno upodobljen na neko bitno sliko, morajo obstajati neke zakonitosti. Vsak znak iz nekega nabora znakov, mora zasedati natanko vnaprej določeno mesto. Pri uporabi takih fontov moramo skrbno predviditi največjo možno velikost znaka v neki aplikaciji, saj pri skaliranju navzgor prihaja do neželenih pojavov, ki so posledica skaliranja, opisanega v razdelku 2.2. Tak font je navadno upodobljen na neki teksturi, koordinato posameznega znaka pa izračunamo. Poleg take teksture potrebujemo tudi podatek o širini posameznega znaka. Le tako lahko dosežemo enakomeren razmak med posameznimi znaki v tekstu. Spodnja slika prikazuje bitno sliko, na kateri je upodobljen nabor znakov:



Slika 8: *Nabor znakov v bitni sliki velikosti 16x16 znakov (vir: Google slike)*

Koordinato posameznega znaka lahko izračunamo z enačbo:

$$x = (\text{ASCII}) >> 4 \qquad (3.2)$$
$$y = (\text{ASCII}) \bmod 16$$

kjer nam (ASCII) predstavlja ASCII vrednost želenega znaka, '>>' nam označuje bitni pomik desno, 'mod' pa deljenje po modulu.

Prednosti uporabe takih fontov so:

- hitro nalaganje v pomnilnik, saj ni potrebno *on-the-fly* upodabljanje iz nekega obstoječega fonta pri nalaganju v pomnilnik
- enostavno kreiranje (obstaja precej prosto dostopnih programov za ta postopek, če ga kreiramo predhodno)

Slabosti pa izhajajo iz že prej omenjenih lastnosti:

- velikost bitne slike: vsak nabor znakov moramo preslikati v bitno sliko, za vsak tip (na primer ležeče, krepko) potrebujemo svojo bitno sliko
- neželeni pojavi pri skaliranju

Drugi podprt način je upodabljanje iz nekega obstoječega TrueType fonta. TrueType je standard za shranjevanje vektorskih fontov, ki ga je razvil Apple v poznih osemdesetih letih. V fontu so shranjeni obrisi v obliki ravnih segmentov premic in kvadratičnih Bézierovih krivulj.

Prednosti uporabe TrueTypa fonta:

- ne zasede veliko prostora na disku
- ne potrebujemo dodatnih datotek z opisom lastnosti znakov
- enostavno skaliranje brez izgube informacij, saj gre za vektorske fonte

Slabosti:

- pred uporabo je potrebno nabor znakov upodobiti v neko teksturo, kar lahko pomeni časovni pribitek pri upodabljanju, če to počnemo on-the-fly
- včasih potrebujemo kak znak iz razširjenega nabora Unicode, kar lahko predstavlja težavo v nekaterih primerih

3.3.3 Video vsebine

Eden izmed ciljev pri snovanju pogona je bila zmožnost predstavitve video vsebin. Video vsebine lahko razdelimo v dve skupini:

- video vsebine, ki se upodabljajo iz nekega vnaprej pripravljenega posnetka
- živa slika

Pogon podpira obe skupini video vsebin.

Za upodabljanje vnaprej pripravljenih posnetkov potrebujemo dekoder, ki komprimiran tok podatkov pretvori v podatke, ki jih lahko upodobimo na zaslon. Video je lahko komprimiran v več različnih formatih, razlikujejo se po strojni zahtevnosti komprimiranja, dekomprimiranja in po velikosti, ki jo zasedejo na trdem disku.

V diplomski nalogi je poudarek predvsem na specifični implementaciji pogona z Direct3D9 knjižnico, zato bom v nadaljevanju opisal potek na Windows platformi.

Za dekompresijo je uporabljen dekoder in enkoder ffdshow. Ffdshow je odprtokodni dekoder-enkoder video in avdio vsebin, ki teče na operacijskem sistemu Windows. Implementiran je v obliki DirectShow in VFW (Video For Windows) filtra. Zanj je značilno, da ponudi enoten filter, ki sam prepozna vsebino in izbere ustrezen nizko nivojski kodek za dekompresijo videa. Ima še nekaj, za to aplikacijo precej uporabnih lastnosti. Ena izmed nastavitev omogoča, da definiramo oziroma omejimo, kakšen bo format pikslov izhodnega toka podatkov. To je precej uporabna značilnost, saj se izognemo pretvarjanju formatov pikslov kasneje, pri upodabljanju na teksturo. Format pikslov dekomprimiranega toka videa je namreč v večini primerov različen od toka, kakršnega uporablja na primer tekstura, na katero se upodablja video.

Druga različica video vsebin je živa slika. Živa slika se v sistem prenese preko zajemalnika žive slike. Načinov, kako se video pretoči iz zajemalnika, je več:

- preko PCI vodila
- preko USB vodila
- preo Firewire vodila

V kasneje opisani specifični implementaciji je bil realiziran zajemalnik na PCI vodilu, njegov programski vmesnik pa je bil realiziran preko DirectShow filtra.

Pri implementaciji se je izkazalo, da je bila realizacija žive slike nekoliko lažja od realizacije drugih video vsebin. Razlog za to je predvsem v tem, da je format pikslov izhodnega toka

podaktov zajemalnika fiksen, medtem ko se pri video vsebinah lahko spreminja od posnetka do posnetka.

3.3.4 Upravljanje z viri

Kot je bilo omenjeno upravljanje z viri prinaša enoten pristop k kreiranju, brisanju in spreminjanju virov. V osnovi upravljanje z viri obsega:

- kreiranje vira
- nalaganje vira v grafični pomnilnik
- brisanje vira iz grafičnega pomnilnika
- brisanje vira iz upravljalnika virov

Razvidno je, da se postopki za zgornje operacije razlikujejo pri različnih vrstav virov, zato sem vsaki vrsti virov dodelil svoj upravljalnik:

- upravljalnik tekstur
- upravljalnik fontov
- upravljalnik video vsebin

Omeniti je potrebno tudi, da gre v osnovi za abstraktne razrede, saj je razen funkcionalnosti, ki omogoča vodenje evidenc o objektu, torej dodajanje v evidenco, brisanje in iskanje po evidencah, stvar implementacije odvisna predvsem od uporabljene nizko nivojske grafične knjižnice.

3.4 Dvodimenzionalni objekti

Dvodimenzionalni objekti so tiste entitete, ki poleg informacije o viru enkapsulirajo vse lastnosti, preko katerih se ta vir upodablja na zaslon. Pogon pozna dva tipa dvodimenzionalnih objektov, ki se upodabljajo na zaslon:

- sprite
- tekst

Lastnosti, ki so vsem dvodimenzionalnim objektom skupne, so:

- enolično ime objekta
- podatek o koordinati na zaslonu
- podatek o vidljivosti
- kot pod katerim se upodablja glede na horizontalno os zaslona
- podatek o transparenci

- z vrednost

Enolično ime objekta je identifikator, po katerem posamezne objekte nekega tipa ločimo med seboj. Ime lahko generiramo naključno, kadar gre za večje število objektov, lahko pa mu ime tudi določimo sami. Ime objekta je enako kot v upravljalniku dvodimenzionalnih objektov, kot bo to opisano kasneje.

Podatek o vidljivosti je prva stvar, ki jo upodobljevalnik preveri, ko se sestavlja vrsta za prikaz objektov. Lastnost omogoča, da na enostaven način nek objekt na zaslonu začasno izključimo, ne da bi ga morali popolnoma izključiti iz scene, ki ji pripada.

Predmet, ki se upodablja na zaslonu, mora imeti definirano transparenco in z vrednost. Transparenca nam določi, kolikšen odstotek informacije objekta pod njim nek objekt zakriva. Dinamično spreminjanje te lastnosti je poleg linearne translacije in rotacije eden izmed najlažjih načinov animacije objekta. Pomembna lastnost je tudi z vrednost, ki nam pove, v katerem nivoju neke scene se objekt upodablja. Tako lahko določimo vrstni red elementov v neki sceni.

3.4.1 Sprite

Sprite je objekt, ki se navadno upodobi na zaslon preko neke teksture. Koncept takega upodabljanja sega v sredino sedemdesetih let in v osnovi predstavlja upodabljanje neke bitne slike v nekem večjem okolju oziroma sceni.

Sprite v pogonu se vedno upodablja preko neke vnaprej pripravljene teksture. Pri tem moramo poznati osnovne lastnosti teksture, ki jo upodabljamo. Najpomembnejša lastnost teksture je razmerje izvorna višina/širina, v katerem je tekstura shranjena. Kot je bilo že omenjeno, nekatere, predvsem starejše grafične kartice, podpirajo le teksture, kjer sta višina in širina potenci števila 2. Kadar imamo teksturo, ki ni teh mer, pride pri nalaganju v pomnilnik do popačenja. To lahko popravimo tako, da ustrezno prilagodimo razmerje velikosti našega sprite objekta.

Vpliv upodobljenih tekstur na zmogljivosti sistema je odvisen predvsem od absolutnega števila prikazanih tekstur, pa tudi od števila različnih prikazanih tekstur. Razlog za drugo lastnost je predvsem način optimizacije, kjer se objekti na zaslonu z enakimi teksturami paketno obdelajo. Veliko število različnih tekstur tako zmanjša možnost paketne obdelave. V tej smeri je mogoče optimizirati aplikacijo tudi tako, da v primeru upodabljanja namesto več manjših tekstur v pomnilnik naložimo te teksture v obliki neke enotne večje teksture. Seveda je tu potrebno potem voditi evidenco, kje v taki teksturi se naša manjša tekstura nahaja.

3.4.2 Tekst

Tekst na zaslonu nam lahko predstavlja katerikoli znak iz vnaprej upodobljenega nabora znakov. Pri upodabljanju teksta veljajo nekoliko drugačna pravilo kot pri upodabljanju objekta sprite. Razlika izvire iz osnovnih lastnosti samih virov. Tekstura je objekt, ki ga lahko poljubno skaliramo, font pa ima fiksno višino znaka. Pri dvodimenzionalnem objektu tipa Text ima velikost tako nekoliko drugačen pomen, kot pri prej opisanem objektu Sprite. Predstavlja nam okvir, v katerega se naš tekst preslika. Kako se le ta preslika, je odvisno predvsem od načina poravnave. Tekst podpira naslednje poravnave na horizontalno os:

- leva poravnava
- sredinska poravnava
- desna poravnava

Vsakemu tekstu lahko določimo tudi barvo. Barva teksta je zapisana z 32 bitnim številom v formatu X8R8G8B8:

- **X** biti 31:24 0xAA000000 (ti biti so samo za poravnano in so vedno 0)
- **R**ed biti 23:16 0x00AA0000
- **G**reen biti 15:8 0x0000AA00
- **B**lue biti 7:0 0x000000AA

Kaj posamezen kanal pomeni, je bilo podrobneje obrazloženo v razdelku 3.1.1.

Upodabljanje in animiranje teksta se je izkazalo za bolj kompleksen problem, kot je bilo na začetku predvideno. Razloga za to sta predvsem dva.

Barva teksta je v osnovi homogena (pri bitnih fontih je lahko tudi drugače) in kot taka razmeroma kontrastna glede na ozadje, zato se na primer pri animaciji vidi vsaka anomalija pri interpolaciji položaja glede na čas. To je še posebej opazno v konkretni aplikaciji, ki ta pogon uporablja, saj je bel tekst upodobljen na zeleno ozadje.

Druga anomalija, ki je tudi posredno posledica kontrasta in je še posebej opazna, kadar upodabljamo večji tekst, pa je pojav, ki mu pravimo trganje strani. Tako anomalijo lahko odpravimo z vertikalno sinhronizacijo. Več o trganju strani je napisano v razdelku 3.6.

3.4.3 Upravljanje z objekti

Z objekti se upravlja preko upravljalnika scen, ki je podrobneje opisan v razdelku 3.4.3.

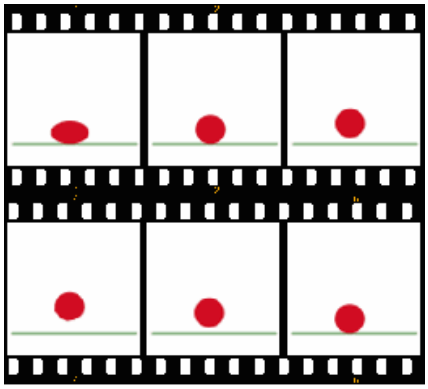
Objekti in scene se namreč logično prepletajo, zato je njihovo upravljanje poenoteno v enem upravljalniku imenovanem SceneManager.

3.4.4 Animacija

Pogon podpira dva načina animacije:

- animacija z predpripravljenimi sličicami objekta
- animacija z transformacijo objekta

Animacija s predpripravljenimi sličicami je pravzaprav eden izmed prvih uporabljenih načinov animacije in je bila prvič uporabljena še pred prihodom televizije in računalnikov.



Slika 8: *Zaporedje sličic animacije (vir: Google slike)*

Ideja je v tem, da imamo zaporedje sličic, kakršno je na sliki 8. Če to zaporedje sličic zaporedno dovolj hitro predvajamo, dobimo občutek, da je objekt na sliki animiran. Pomembno je, da je časovni razmak med prikazoma posamezne sličice ustrezno definiran, lahko tudi fiksen. Največja prednost takega pristopa je, da omogoča učinke, ki samo z transformacijami v 2D prostoru niso mogoči. Animacija, ki je v 2D prostoru ni mogoče implementirati, je na primer imitacija animacije tretje dimenzije objekta. Zaporedje sličic nam namreč lahko prikazuje objekt, ki se vrti v vseh treh dimenzijah. Največja slabost takega pristopa je, da moramo imeti sličice predpripravljene, včasih je tako primernejša animacija z transformacijo objekta.

Animacija s transformacijo objekta je postopek, kjer uporabljamo metode za transformacijo, kakršne so opisane v poglavju 2. Spremembo vrednosti koraka animacije izrazimo v odvisnosti od časa. Nekaj primerov takih korakov:

- opravljena pot (gibanje, translacija)
- sprememba alfa vrednosti (fade efekt)

- sprememba višine, dolžine objekta (skaliranje)
- sprememba kota objekta (rotacija)

3.5 Scena

Scena je objekt, ki enkapsulira vse vidne in nevidne Renderable objekte na zaslonu. Vsebuje lahko tako objekte, ki se upodabljajo, kot tudi ostale scene, ki so ji hierarhično podrejene. Scene so strukturirane v drevesa, tako imenovane grafe scen, kjer nam objekti v drevesu predstavljajo liste, posamezne veje pa nam predstavljajo podscene. Vsaka scena ima lahko neomejeno število listov, kot tudi neomejeno število vej, podscen. Omejitev je seveda performanca ter količina grafičnega in delovnega pomnilnika ciljnega sistema.

Scena ima več lastnosti, ki definirajo prikaz njenih elementov na zaslonu. Večina lastnosti se prenese neposredno po drevesni strukturi navzdol. Lastnosti, ki definirajo prikaz njenih elementov in podscen so:

- enolično določeno ime
- velikost in položaj
- starševska scena
- z vrednost
- vidljivost
- alfa vrednost, transparenca
- seznam podscen
- seznam objektov, ki jih scena vsebuje

Vsaka scena je natanko identificirana z enolično določenim imenom, ki ga lahko generiramo tudi naključno.

Velikost je podana v strukturi s štirimi polji: izhodišče na x osi, izhodišče na y osi, ter višina in širina. Koordinatni izhodišči sta vedno relativni glede na starševsko sceno in sta lahko negativni, lahko pa sta tudi večji od širine ali višine zaslona v pikslih. To omogoča animacijo, kjer se scena (pravzaprav njeni elementi) „pripelje“ preko zaslona. Poleg velikosti imamo v razredu tudi podatek o absolutni velikosti. To je uporabno predvsem, kadar potrebujemo podatek o tem, kje natanko se neka scena ali objekt na zaslonu nahaja. Primer so kompleksnejše animacije, kadar želimo uskladiti medsebojno gibanje oziroma pozicijo dveh scen, oziroma objektov, ki pa nista neposredno v sorodu. To vrednost izračuna in osveži ob vsaki iteraciji zanke pogon sam.

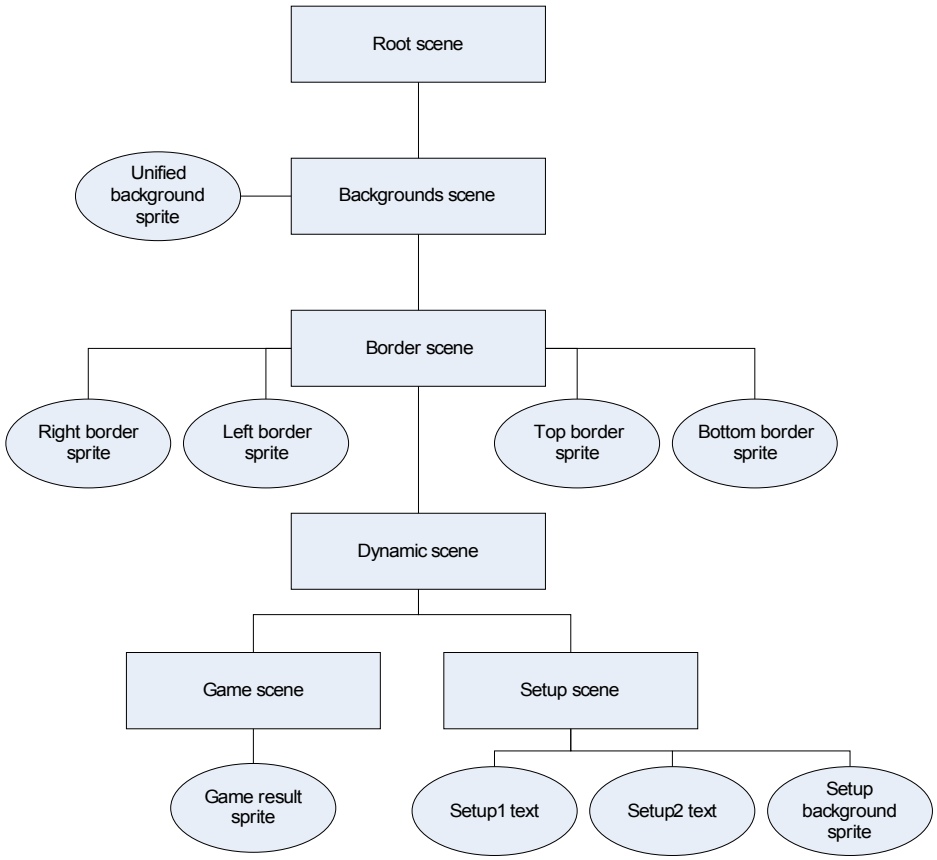
Z-vrednost nam omogoča nadzor nad vrstnim redom upodabljanja scen, ki so v drevesni strukturi na enakom nivoju.

Spreminjanje lastnosti vidljivosti scene omogoča enostavno izključitev iz vrste za upodabljanje. To pomeni, da se v naslednjih iteracijah zanke ta scena, njene podscene in pripadajoči objekti ne bodo upodabljali. Je prav tako lastnost, ki se prenaša navzdol po drevesni strukturi.

Transparenca tudi spada v skupino lastnosti, ki se prenaša po drevesni strukturi navzdol, vendar z omejitvijo. Za vse elemente, ki so hierarhično gledano pod neko sceno, predstavlja najmanjšo vrednost transparence, kar pomeni, da morajo biti vsi elementi nižje od te scene vsaj tako prosojni, kot je scena sama. Tak pristop se je izkazal kot potreben in precej učinkovit predvsem zaradi efekta, kjer želimo neko vejo drevesa postopoma zatemniti. Namesto popravljanja vseh transaparenc po drevesu navzdol spremenimo samo alfa vrednost scene, ki to vejo vsebuje.

3.5.1 Graf scen

Graf scen je drevesna struktura, ki predstavlja hierarhične povezave med scenami in objekti. Pri upodabljanju se preslika v vrsto za upodabljanje. Vrsta se gradi od najvišjega elementa, ki je navadno izhodišče strukture drevesa (Root), proti nižjim elementom. Vrstni red upodabljanja različnih nivojev je tako definiran z globino nivoja samega. Položaj oziroma vrstni red upodabljanja scen, ki so v istem nivoju definira z-vrednost, lastnost, ki jo ima vsaka scena.



Slika 9: *Graf scen*

Zgornja slika prikazuje graf scen enostavne aplikacije, ki vsebuje okno, okvir okna in podatek o aktualnih nastavitvah aplikacije.

3.5.2 Upravljanje s scenami in objekti

Upravljanje s scenami in objekti je poenoteno v enem razredu. Razlog za to je predvsem centraliziran način vodenja in upravljanja z objekti in scenami. Brez poenotenja bi objekte bilo potrebno evidentirati v več razredih, kar pa bi zaradi njihove medsebojne odvisnosti predstavljalo težavo in nepotrebno komplikacijo.

Upravljanje z objekti in scenami obsega:

- kreiranje
- brisanje

Vse to je zajeto v razredu SceneManager. Ta razred hrani kazalce na vse objekte scen, pa tudi kazalce na vse ostale Renderable objekte. Ti kazalci so shranjeni v asociativnem vsebovalniku

tipa map, opisanim v razdelku 3.1.1. Tak način omogoče enostavno dostopanje do nekega elementa preko njegovega imena. V razredu sta dva taka vsebovalnika, eden za Renderable objekte, drugi pa za scene.

Upravljalnik vedno vsebuje vsaj eno sceno, tako imenovano Root sceno. Root scena je izhodiščna scena v grafu scen, iz katere se potem vejijo posamezne scene.

3.5.3 Vrste za upodabljanje

V pogonu imamo dva tipa vrst za upodabljanje:

- **RenderQueue:** je vrsta, kjer so elementi med seboj urejeni tako, da se upodablja od najvišjega nivoja (root scena) do najnižjega nivoja v drevesu. Hierarhično gledano, se vsak naslednji element upodablja preko prejšnjega elementa, oziroma vsaka scena se upodablja preko prejšnje scene.
- **OverlayRenderQueue** je vrsta, ki se vedno upodablja za vrsto RenderQueue. Namenjena je predvsem poenostavljenem prikazu različnih podatkov, na primer tekstovnih oznak, vodnih žigov, sistemskih nastavitev in menijev. Le ti pa se vedno upodabljajo zadnji, navadno kar preko ostalih objektov.

Objekti se v vrsto dodajo s sprehodom po drevesu navzdol. Scene, ki imajo zastavico nastavljeno tako, da se ne upodabljajo, se preskoči. S tem preskočimo celotno vejo drevesa, ki jo ta scena predstavlja. Spodnja koda prikazuje algoritem sprehoda po drevesu navzdol:

```
while(root != NULL)
{
        // Sprehodimo se skozi vse podscene trenutne scene
        ChildSceneIterator csIt = root->getChildSceneIterator();
        while(csIt.hasMoreElements())
        {
                // Ce scena ni vidna, potem njenih podscen in objektov ne upodabljamo
                Scene* tmpScene = csIt.getNext();
                if (tmpScene->getVisible() == true)
                {
                        tmpNodes.push_back(tmpScene);
                        size++;
                }
        }
        // Poiscemo vse Renderable objekte trenutne scene
        ObjectRenderIterator oIt = root->getObjectIterator();
        while (oIt.hasMoreElements())
        {
                RenderableObjectPtr ro = oIt.getNext();
```

```
                    if (ro->getVisible() == true)
                    {
                            queue->addRenderable(ro);
                            ++numberOfObjects;
                    }
            }
            // Ce scena vsebuje se kako podsceno se premaknemo vanjo
            if (size > 0)
            {
                    root = tmpNodes[queuePointer];
                    queuePointer++;
            }
            else root = NULL;
            --size;
            ++numberOfScenes;
}
```

Koda 2: *Algoritem za sprehod po drevesu navzdol*

Vsaka scena vsebuje poleg seznama vseh svojih objektov tudi seznam, v katerem so samo vidni objekti. To sicer prinaša neke vrste redundanco, saj imamo kazalec na objekt v dveh seznamih, vendar tak pristop prinaša določeno pohitritev, saj pri vstavljanju elementov v vrsto za upodabljanje ni potrebno pregledovanje vseh elemetov v sceni, temveč samo tistih, ki so vidni in se nahajajo v prej omenjeni vrsti.

3.6 Upodabljanje

Razred za upodabljanje vsebuje funkcionalnost, ki ob dejanski implementaciji na osnovi neke grafične knjižnice, neposredno upodablja objekte na zaslon.

V osnovi omogoča več različnih načinov delovanja.

Celozaslonski in okenski način: v tem načinu je velikost okna enaka velikosti zaslona. Pri tem je potrebno omeniti, da nekatere knjižnice razlikujejo tako imenovani celozaslonski način in okenski način, kjer je pri zadnjem velikost okna sicer lahko enaka velikosti zaslona, vendar je funkcionalnost omejena v primerjavi s celozaslonskim načinom. Nekatere zmožnosti grafične kartice se lahko uporablja samo v celozaslonskem načinu.

Vertikalna sinhronizacija: grafika se na računalniški monitor iz zgodovinskih razlogov izrisuje od zgoraj navzdol po vrsticah[7]. Kadar se podatki v slikovnem medpomnilniku grafične kartice spremenijo, še predno se je le ta v celoti izrisal na zaslon, pride do tako imenovanega trganja strani, kjer se na zaslon do neke vrstice izriše prejšnja vsebina medpomnilnika, v naslednjih vrsticah pa sedanja. Temu se lahko izognemo tako, da se medpomnilnik osvežuje

samo v vnaprej določenih intervalih. Prednost sinhronizacije je vsekakor v tem, da do teh neželenih pojavov ne prihaja, vendar lahko pomeni počasnejše delovanje aplikacije, saj se v vsaki iteraciji zanke nekaj časa čaka, da se ujame vnaprej določen interval. Vertikalna sinhronizacija se je izkazala za nujno potrebno predvsem pri scenah, kjer je ozadje več ali manj statično, v ospredju pa se premikajo elementi z neko konstantno hitrostjo, na primer večji premikajoč tekst na enobarvni, glede na tekst visoko kontrastni podlagi. V ostalih primerih, na primer pri upodabljanju videa, se je izkazala za nepotrebno in je kvečjemu omejevala aplikacijo.

Število slikovnih medpomnilnikov: uporaba dveh medpomnilnikov (pravzaprav dejansko ne gre za dva različna medpomnilnika, temveč za naslov v slikovnem pomnilniku, ki se periodično spreminja) omogoča, da medtem ko se prvi medpomnilnik izrisuje na zaslon, aplikacija dostopa do drugega medpomnilnika. Ob vnaprej točno določenem intervalu prvi medpomnilnik postane drugi, drugi medpomnilnik pa postane prvi. Uporabimo lahko tudi več kot dva različna medpomnilnika. Uporaba več kot enega medpomnilnika daje vtis gladkejšega gibanja dinamičnih elementov na zaslonu.

Tehnika umazanih pravokotnikov: izkoristimo dejstvo, da je pri nekaterih aplikacijah večji del zaslona statičen in dinamika majhna. V tem primeru nam ni potrebno pisati celotnega medpomnilnika, temveč lahko v medpomnilnik vpišemo samo tiste dele zaslona, ki so se spremenili glede na prejšnjo iteracijo tega medpomnilnika, kar lahko prinese precejšnjo pohitritev. Težava pa se pojavi pri uporabi več kot dveh slikovnih medpomnilnikov, saj moramo voditi evidenco za vsak element upodobljen na zaslonu in upoštevati, kje vse se je element nazadnje nahajal v točno določnem medpomnilniku. Najbolj pomembno je, da se nam vsebina medpomnilnikov med posameznimi iteracijami zanke ohranja.

3.6.1 Iteracija zanke

Glavna zanka v pogonu je sestavljena iz klica štirih metod:

```
while (mRenderSystem->windowMessagePump() == true)
{
	mProfiler->measureFPS();
	step();
	render();
	sleep(20);
}
```

Koda 3: *Glavna zanka*

V prvi metodi najprej izmerimo pretečen čas od prejšnje iteracije v namene profiliranja, kot je opisano v naslednjem razdelku.
Druga metoda je neke vrste logični korak iteracije. Ta metoda je vstopna točka uporabnikove aplikacije. Znotraj uporabniška aplikacija upravlja z viri, kreira objekte, upravlja z scenami in objekti.

Metoda render() kliče izbrani upodobljevalnik. Le ta razvrsti elemente v ustezne vrste in jih upodobi na zaslon. Več o upodabljanju s knjižnico Direct3D je napisano v razdelku 3.7.3.

3.6.2 Profiliranje

Pri samem snovanju pogona in aplikacije potrebujemo pregled nad tem, kakšen vpliv na performance sistema ima sprememba, tako v aplikaciji, kot morebitna optimizacija pogona. V ta namen je bil razvit razred Profiler, ki v vsaki iteraciji zanke na zaslon izriše aktualne performančne podatke v enoti FPS (Frames Per Second) oziroma sličice na sekundo. Prikazani so naslednji podatki:

- Trenutna hitrost upodabljanja izražena z [FPS]
- Največja hitrost upodabljanja [FPS]
- Najnižja hitrost upodabljanja [FPS]

Performančni podatki so sicer nekoliko zakasnjeni, saj se trenutna hitrost upodabljanja računa kot povprečje FPS zadnje sekunde.

Omeniti je potrebno mejni primer in sicer, kadar uporabljamo vertikalno sinhronizacijo. Takrat bo vrednost FPS vedno sinhronizirana s frekvenco osveževanja vertikale. Pri LCD zaslonih je bilo to tipično 60 FPS, oziroma 30 FPS, pri grafično bolj zahtevni aplikaciji, kjer se je zaradi zahtevnosti upodabljalo le na vsak drugi interval osveževanja vertikale.

3.7 Specifična implementacija za grafično knjižnico Direct3D

Direct3D je del Microsoftove DirectX zbirke programskih vmesnikov. DirectX med drugim vsebuje tudi DirectSound (knjižnica za delo z zvočno kartico), DirectPlay (knjižnica za komunikacijo) in v starejših verzijah DirectDraw, ki je bil namenjen izključno delu z 2D grafiko [8].

Namenjen je predvsem upodabljanju elementov v 3D prostoru, uporablja se ga pa tudi za delo v 2D prostoru, saj je začenši z verzijo 8 DirectDraw integriran v Direct3D API in se ga za nove aplikacije ne uporablja več.

Implementiran je kot skupek COM vmesnikov. Microsoft Component Object Model je platformno neodvisen sistem za kreiranje programskih komponent, ki jih lahko uporabimo v celi vrsti programskih jezikov. Omogoča uporabo komponent brez poznavanja natančnega delovanja le teh v notranjosti, zadostuje le poznavanje vmesnika. Programski vmesnik za delo s COM objekti je namreč standardiziran. V nekaterih aplikacijah ga je sedaj nadomestilo okolje .net.

Vsaka COM komponenta je enolično določena z identifikatorjem CLSID (Class Identificator). Svojo funkcionalnost izraža z naborom vmesnikov, ki so tudi enolično določeni z identifikatorjem. V minimalni obliki mora vsaka COM komponenta implementirati vmesnik IUnknown. Ta vmesnik ima tri metode:

- AddRef()
- Release()
- QueryInterface()

Nabor pravil in rezultat, ki ga vrnejo zgoraj uporabljene metode, je strogo določen.

COM objekte tipično lahko uporablja več uporabnikov, lahko se uporabijo tudi med različnimi procesi, zato je pomembno, da je zagotovljen način, kako se objekt izbriše iz pomnilnika šele po končani uporabi vseh uporabnikov. To je rešeno tako, da se vsakič, ko nam objekt preko neke metode vrne referenco vmesnika, poveča interna spremenljivka referenc za ena. Implementirano je tako, da objekt na začetku interno kliče metodo AddRef(). Ko uporabnik konča z uporabo vmesnika, mora eksplicitno klicati metodo Release(). Če interna spremenljivka doseže vrednost nič, pomeni, da vmesnik ni več uporabljan, takrat se objekt sprosti iz pomnilnika. Tu si lahko pomagamo z uporabo pametnih kazalcev, ki nam lahko precej poenostavijo delo z COM objekti. Tak pameten kazalec je CComPtr iz knjižnice ATL (Active Template Library), ki sem ga uporabil tudi sam pri delu z Direct3D COM objekti.

Namen knjižnice Direct3D je abstrakcija komunikacije med grafično aplikacijo in gonilniki za grafično kartico.

Direct3D je Immediate Mode API, kar pomeni, da klici metod knjižnice neposredno prožijo upodabljanje na zaslon, kar omogoča fleksibilnost pri programiranju. Grafična aplikacija je torej sama v celoti odgovorna za upodabljanje elementov.

Direct3D Immediate mode predstavljajo trije tipi abstrakcij:

- *devices*
- *resources*
- *swap chains*

Poznamo 4 različne tipe naprav (devices):

- **Hardware Abstraction Layer (HAL) device:** strojno pospešena grafična kartica
- **Reference (REF) device:** referenčna naprava, ki podpira celotno Direct3D funkcionalnost. Gre za softwersko implementacijo funkcionalnosti, torej ni nikakor pospešena in zato izjemno počasna. Uporablja se le za razhroščevanje in odkrivanje napak, na primer na gonilnikih grafične kartice. Za delovanje potrebujemo Direct3D SDK.
- **Pluggable software device:** uporablja se za programsko rasterizacijo. Uporabimo ga takrat, kadar nam grafična kartica neke funkcionalnosti ne ponuja, le to pa potem lahko implementiramo programsko.
- **Null reference device:** ta naprava je izbrana, kadar Direct3D SDK ni naložen in je zahtevan Reference device.

Vsaka naprava vsebuje tudi zbirko virov. Viri so specifični podatki, ki se uporabijo pri upodabljanju. Vsak vir ima štiri atribute:

- **Type:** opisuje tip vira: surface, volume, texture, cube texture, volume texture, surface texture, index buffer, vertex buffer.
- **Pool:** opisuje, kje je shranjen vir in kako ga naprava obravnava. Default pool pomeni, da se vir nahaja samo v pomnilniku naprave. Managed pool pomeni, da je vir shranjen v delovnem pomnilniku in se ga prenese v pomnilnik naprave po potrebi. System memory pool pomeni, da se vir nahaja samo v delovnem pomnilniku. Scratch pool je podoben prejšnjemu, le da ni omejen s funkcionalnostjo naprave.
- **Format:** opisuje pixel format, v katerem je shranjen vir.
- **Usage:** skupina zastavic, ki definira, na kakšen način bo uporabljen dani vir.

Swap chain je skupek medpomnilnikov, ki se izmenično upodabljajo na zaslonu. Vsak medpomnilnik nam predstavlja množico pikslov, ki vsebujejo atribute, kot sta barva in globina.

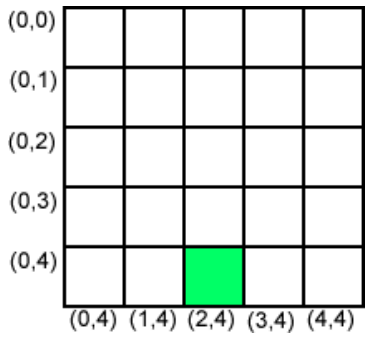
3.7.1 Viri

Vsaka grafična knjiznica ima svoj nabor pravil za upravljanje in operiranje z viri, kot so teksture, kadar pa jih posredno oziroma neposredno podpirajo, tudi s fonti. Direct3D fontov

neposredno ne podpira, podpira jih preko visokonivojske knjižnice D3DX (Direct3D Extension).

Teksture

Planarne teksture so v knjižnici Direct3D predstavljene s svojim koordinatnim sistemom, ki ima izhodišče v levem zgornjem kotu. Način naslavljanja tega prostora je nekoliko drugačen od tistega, ki smo ga navajeni iz, v prvem poglavju omenjenega, kartezičnega koordinatnega sistema. Vsak teksel je naslovljen s parom koordinat, v razponu od 0.0f do 1.0f, torej z realnimi vrednostmi. Na spodnji sliki je poenostavljen primer, kam se preslika naslov (0.5, 1.0):



Slika 10: *Preslikava naslova (0.5, 1.0) v teksel (2, 4)*

Teksture se na zaslon preslikajo z inverzno preslikavo. Za vsak piksel se izračuna ustrezen položaj teksla v teksturi. Ta se potem preslika na način, kot je omenjeno v razdelku 2.2. Podprti so naslednji načini filtriranja tekstur:

- metoda najbližjega soseda
- linearno filtriranje (interpolacija)
- bilinearno filtriranje (interpolacija)
- ansitropično filtriranje
- filtiranje z mipmapi

Vsi načini razen ansitropičnega filtriranja so podrobneje predstavljeni v razdelku 2.2. Ansitropično filtriranje je uporabno predvsem v 3D prostoru, kjer je način filtriranja oziroma vzorčenja odvisen od kota, pod katerim se projecira tekstura na zaslon. V pogonu se ga ne uporablja.

3.7.2 Video vsebine

Video vsebine se na konkretni implentaciji upodabljajo preko programskega vmesnika DirectShow.

DirectShow, oziroma krajše DS, je multimedijsko ogrodje in API za predvajanje in upravljanje medijskih tokov in datotek na operacijskem sistemu Windows [9]. Bazira na prej omenjenem COM ogrodju. Omogoča splošen dostop do multimedijskih vsebin in tokov iz različnih programskih jezikov.

Sprva je bil DS del knjižnice DirectX, kasneje je postal del Platform SDK programskega vmesnika, ki se sedaj imenuje Windows SDK.

V DS se kompleksnejša multimedijska opravila, kot so na primer predvajanje videa oziroma zvoka, izvedejo v več enostavnejših korakih, preko tako imenovanih filtrov. Tak pristop omogoča, da kompleksnejši problem strukturiramo na več enostavnejših. Posamezne filtre lahko povezujemo med seboj. Filtre med seboj povezujemo preko vhodnih in izhodnih vtičev, seveda pod pogojem, da je tok podatkov kompatibilen. DS pozna tri vrste filtrov:

Source filters: predstavljajo nek vir toka podatkov, kot je branje bajtov iz datoteke z video vsebino oziroma tok podatkov iz zajemalnika žive slike.

Transform filters: nad izhodnim tokom nekega drugega filtra izvajajo transformacijo, ki je lahko dekompresija MPEG okvira oziroma neposredna manipulacija vira, kot je spreminjanje barve piksla.

Renderer filters: ti filtri so navadno zadnji v hierarhični verigi filtrov in predstavljajo vez med IO sistemom in verigo. Na primer zapis video datoteke na trdi disk, ali pa predvajanje le te na zaslon.

Za upodabljanje video vsebin v posamezno teksturo je uporabljen VMR9 (Video Mixing Renderer 9) filter za upodabljanje.

VMR9 Direct Show filter je naslednik VMR7 filtra in je bil prvič predstavljen v Direct3D verziji 9. Omogoča enostavno upodabljanje video vsebin in tokov, prav tako tudi mešanje le teh, na primer mešanje toka podnapisov in toka video vsebine.

VMR9 je navadno zadnji člen v verigi filtrov in preko Direct3D upodablja okvir (sliko) videa neposredno na zaslon, lahko pa nadzor nad upodabljanjem prevzamemo sami, kot bo to opisano kasneje.

Obstajajo trije načini delovanja [10]:

- **Windowed mode:** privzet način delovanja. Filter sam ustvari okno, kamor se upodablja video. Oknu lahko določimo starševsko okno. Za upodabljanje skrbi filter sam.
- **Windowless mode:** filter sam okna ne ustvari, temveč moramo to storiti sami. Video se tako lahko upodablja v okno naše aplikacije, določiti moramo le pozicijo. Pri tem moramo biti pozorni na to, da filter obvestimo o dogodkih WM_PAINT, WM_DISPLAYCHANGE in WM_SIZE v oknu aplikacije. Podobno kot pri načinu windowed, je tudi tu za upodabljanje odgovoren filter sam.
- **Renderless mode:** za upodabljanje je odgovorna aplikacija, ki ga uporablja. Tak pristop omogoča aplikaciji, da ima popoln nadzor nad Direct3D napravo in nad upodabljanjem. To je potrebno predvsem takrat, kadar preko Direct3D poleg videa upodabljamo še druge elemente. Pogon uporablja tak pristop.

Za uporabo VMR9 filtra v načinu renderless moramo implementirati svoj *allocator-presenter.* Postopek za implementacijo je sledeč:

1. Implementirati je potrebno razred, ki podpira vmesnika IVMRSurfaceAllocator9 in IVMRImagePresenter9.
2. Naredimo poizvedbo preko VMR-9 za IVMRFilterConfig9 in IVMRSurfaceAllocatorNotify9 vmesnika.
3. Pokličemo metodo IVMRFilterConfig9::SetRenderingMode z zastavico VMR9Mode_Renderless.
4. Pokličemo metodo IVMRSurfaceAllocatorNotify9::AdviseSurfaceAllocator s kazalcem na allocator-presenter.
5. Pokličemo metodo IVMRSurfaceAllocator9::AdviseNotify s kazalcem na VMR-9 IVMRSurfaceAllocatorNotify9 vmesnik. Ta in prejšnji korak vzpostavita povezavo med našim objektom allocator-presenter in med VMR9 filtrom.
6. Dodelimo površine z IVMRSurfaceAllocator9::InitializeDevice callback metodo, ki se kliče samodejno ob inicializaciji:
    - Pokličemo IVMRSurfaceAllocatorNotify9::SetD3DDevice metodo s kazalcem na Direct3D napravo in z oprimkom na zaslon.
    - Kreiramo Direct3D površino, ki ustreza parametrom, ki smo jih zastavili v InitializeDevice metodi. To lahko storimo z metodo IVMRSurfaceAllocatorNotify9::AllocateSurfaceHelper.

- Želimo upodobiti okvir videa na površino teksture, zato moramo v strukturi VMR9AllocationInfo postaviti zastavico VMR9AllocFlag_TextureSurface. V primeru, da naprava ne podpira formata pikslov, moramo kreirati novo teksturo in vanjo kopirati podatke iz površine videa.

7. Filter VMR9 dobi površino z objekta allocator-presenter preko metode IVMRSurfaceAllocator9::GetSurface.
8. Ob vsakem klicu metode IVMRImagePresenter9::PresentImage prenesemo sliko videa v prej inicializirano teksturo.
9. Ob koncu toka podatkov VMR9 filter kliče metodo IVMRSurfaceAllocator9::TerminateDevice. Objekt allocator-presenter mora sprostiti vse inicializirane resurse (površine in teksture).

Očitno je, da se kliče metoda IVMRImagePresenter9::PresentImage asinhrono glede na glavno iteracijo zanke. Zato je pomembno, da se ob inicializaciji same naprave Direct3D omogoči več nitenje. To storimo ob inicializaciji naprave Direct3D z zastavico D3DCREATE_MULTITHREADED.

Video se torej upodablja na neko, v ta namen prej pripravljeno teksturo objekta Sprite, le ta pa se potem upodablja na zaslon. Tak pristop omogoča, da video vsebino tudi dodatno animiramo.

3.7.3 Upodabljanje

Pred uporabo moramo knjižnico najprej inicializirati. Knjižnico inicializiramo s klicem metode:

```
HRESULT CreateDevice(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  HWND hFocusWindow,
  DWORD BehaviorFlags,
  D3DPRESENT_PARAMETERS * pPresentationParameters,
  IDirect3DDevice9 ** ppReturnedDeviceInterface
);
```

Pri inicalizaciji definiramo parametre delovanja knjižnice tekom izvajanja aplikacije. Najpomembnjši parametri so:

- **BehaviorFlags**; zastavice, kjer nastavimo parametre, kot so način procesiranja točk v prostoru, možnost več nitenja, delovanje gonilnikov.
- **D3DPRESENT_PARAMETERS**; struktura, v kateri se hrani predstavitvene parametre, ki vplivajo na prikaz elementov na zaslon. Ti parametri so na primer število

medpomnilnikov, okenski/polno zaslonski način, format pikslov medpomnilnikov, višina ter širina, kadar aplikacija ne deluje v polnozaslonskem načinu.

Postopek za upodabljanje poteka v petih korakih:

1. izbrišemo medpomnilnik
2. pokličemo metodo za začetek upodabljanja na trenutno izbrani medpomnilnik
3. kličemo metode za upodabljanje na trenutno izbrani medpomnilnik
4. pokličemo metodo za konec upodabljanja na trenutno izbrani medpomnilnik
5. prikažemo trenutni medpomnilnik

V pogonu je ta postopek enkapsuliran v treh metodah:

- **void D3D9RenderSystem::_beginFrame()**; v tej metodi sta enkapsulirana koraka 1 in 2, prej omenjenega postopka.
- **void D3D9RenderSystem::renderObject(RenderableObjectPtr renderable)**; upodobi nek objekt na izbrani medpomnilnik, torej 3. korak upodabljanja.
- **void D3D9RenderSystem::_endFrame()**; vsebuje koraka 4 in 5.

Direct3D pozna tri vrste osnovnih gradnikov, ki se izrisujejo na zaslon, tako imenovanih primitivov, to so točka, črta in trikotnik. Med seboj se delijo še naprej, glede na način kako podamo koordinate njihovih točk.

Sama metoda za upodabljanje 2D elementov v 3D okolju je nekoliko drugačna od upodabljanja 2D elementov s konvencionalnimi metodami, saj tu upodabljamo teksturo direktno na nek poligon. Pravokotni poligon določimo kot množico oglišč dveh trikotnikov, ki sestavljata tak poligon. Najbolj intuitiven način, kako definiramo tak poligon je, da določimo vsa oglišča trikotnikov. Pri tem sta dve oglišči trikotnika skupni. Načinov, kako podamo oglišča trikotnikov je pri Direct3D namreč več. Na tako nastali pravokotni poligon prilepimo teksturo, nato pa upodabljanje na medpomnilnik zaključimo, kot je opisano v koraku 4. Pri tem moramo biti pozorni na način preslikave teksla v piksel, ki se v uporabljeni različici Direct3D9 malce razlikuje od načina uporabljenega v kasnejši različici Direct3D10.

4 Zaključek

V diplomskem delu sem predstavil snovanje in implementacijo 2D grafičnega pogona. Zastavljeni pristop k snovanju in načrtovanju se je izkazal za učinkovitega in primernega.

Rezultat dela je pogon, ki omogoča manipulacijo nad 2D objekti, omogoča pa tudi predvajanje video vsebin. S pravilnim pristopom k snovanju je tudi omogočena prenosljivost in neodvisnost od uporabljene nizko nivojske grafične knjižnice.

Najzahtevnejša je bila implementacija specifične implementacije za nizko nivojsko grafično knjižnico in predvajanje video vsebin. Razlog je predvsem v naboru funkcionalnosti, ki jo (ne)podpirajo proizvajalci strojne opreme. Izkazalo se je, da grafične kartice, predvsem tiste v nižjem cenovnem razredu, ne podpirajo nekaterih ključnih funkcij, ki so bile predvidene za uporabo v pogonu. Implementacija je zato morala zajeti nabor funkcionalnosti, ki zadovoljivo deluje tako na strojni opremi nižjega, kot tudi na strojni opremi višjega cenovnega razreda.

Pogon, oziroma aplikacija, ki ga uporablja, zadovoljivo deluje tudi na počasnejših sistemih, seveda z določenimi omejitvami. Možnost nadaljne optimizacije je predvsem v prilagajanju načina upodabljanja objektov z uporabniško aplikacijo, pa tudi z večjim izkoristkom funkcionalnosti, ki jo podpira predvsem strojna oprema višjega cenovnega razreda.

Pri nadaljni optimizaciji je seveda na mestu razmislek o ekonomski upravičenosti le te. Zaradi nenehnega upadanja in večanja zmogljivosti strojne opreme je morda bolj ekonomična uporaba zmogljivejše strojne opreme kot nadaljna optimizacija. Seveda je to popolnoma pogojeno z aplikacijo, ki ta pogon uporablja.

Literatura

[1] Ogre3D graphic engine. Dostopno na: http://www.ogre3d.org/

[2] Simple DirectMedia Layer. Dostopno na: http://www.libsdl.org/

[3] Irrlicht graphic engine. Dostopno na: http://irrlicht.sourceforge.net/

[4] James D. Foley, Andries Van Dam, Steven K. Feiner, John F. Hughes, *Computer graphics: principles and practice,* Lebanon, Indiana, U.S.A.: Addison-Wesley, 1995

[5] Tomas Möller, Tomas Akenine-Moller, Eric Haines, *Real-time Rendering,* Natick, Mass.: AK Peters, Ltd., 2002

[6] Converting Between YUV and RGB . Dostopno na: *http://msdn.microsoft.com/en-us/library/ms893078.aspx*

[7] Lynn Pocock, Judson Rosebush, *The Computer Animator's Technical Handbook,* San Francisco: Morgan Kaufmann Publishers, 2002

[8] Alan Thorn, *DirectX 9 Graphics: The Definitive Guide to Direct3D,* Plano, TX: Wordware Publishing Inc., 2005

[9] DirectShow. Dostopno na: http://msdn.microsoft.com/en-us/library/ms783323(VS.85).aspx

[10] VMR9 modes of operation. Dostopno na: http://msdn.microsoft.com/en-us/library/dd390956(VS.85).aspx

Seznam slik

Seznam tabel

Seznam programske kode

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Luka Čović, z vpisno številko 63010020, sem avtor diplomskega dela z naslovom:

RAZVOJ GRAFIČNEGA POGONA ZA UPODABLJANJE V 2D PROSTORU

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Saše Divjaka
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 13.3.2009 Podpis avtorja:_____