Datavetenskap

Johan Höjskeld

Anders Svensson

Förbättring av texturhantering i tredimensionella datorspel

Examensarbete, C-nivå

2002:18

Förbättring av texturhantering i tredimensionella datorspel

Johan Höjskeld

Anders Svensson

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

_____

Johan Höjskeld

_____

Anders Svensson

Godkänd, 4 Juni 2002

_____

Handledare: Stefan Alfredsson

_____

Examinator: Stefan Alfredsson

Sammanfattning

Detta dokument beskriver det examensarbete som gjorts under vårterminen 2002. Arbetet handlar om att implementera några olika verktyg för att kunna optimera kvaliteten på texturer i tredimensionella datorspel. Här beskrivs en hel del fakta om hur texturerna i ett tredimensionellt datorspel är uppbyggda, till exempel vad mipmapping, bilinjär filtrering, ditrering och texturewrapping är för något. Annat som beskrivs är hur en textur kan representeras i minnet på en dator. Antingen att det lagras helt okomprimerat eller det är komprimerat på ett eller annat sätt. En grundläggande beskrivning av vad bildformatet Targa är samt en beskrivning av DirectX ges också.

Det vi har arbetat fram beskrivs ingående, vilket i huvudsak är en texturhanterare. För att kunna hantera texturerna på ett effektivt sätt krävdes det bland annat att ett filformat för att representera texturer skapades. Med hjälp av detta filformat kan sedan texturerna användas på ett effektivare sätt, med avseende på minnesutnyttjande. Här kan du också läsa om de olika applikationerna som vi har tagit fram för att kunna omvandla traditionella texturer till vårt texturformat. Vi har också tagit fram verktyg för att man i en Windowsmiljö skall kunna granska texturerna i olika format och storlekar på ett enkelt sätt.

Implementationen av några klasser vi gjort beskrivs också översiktligt i detta dokument. De olika implementationerna man kan läsa om här behandlar bland annat den grundläggande klassen som sköter allt som har med texturerna att göra. Till sist beskrivs också hur implementationen av en texturhanterare som egentligen varit huvudmålet med arbetet har skett. I slutet av rapporten diskuteras hur applikationerna skulle kunna ha förbättrats. Allra sist i rapporten dras slutsatser av det arbetet som gjorts under denna tid.

Improving texture management in threedimensional computer games

Abstract

This document describes the bachelor's project which we have worked on during the spring of 2002. The work is about implementing some different tools which can be used to optimize the textures used in three dimensional computer games.

We will give some background facts about how textures are used in a 3d game and also a bit about how they are constructed. We will describe the terms mipmapping, bilinear filtering, dithering and texture wrapping. Other parts of this document describes how a texture can be represented in the computer memory.

We also give a breif explanation of Targa and DirectX.

Our own work is also described in more details. The main part is a texturemanager. To be able to handle the textures in an effective way we had to construct a file format for representing textures. You can also read about the different tools we have constructed to convert traditional textures to our texture format. We also constructed a tool to view the textures in different formats and sizes in a Windows environment.

The implementation of some of our classes is also described in general in this document.

Then we describe how we implemented the texturemanager which has been the main objective for our work. At the end we discuss how we could make our tools better. The improvements we think of is mainly implementational.

Finally we make our conclusions about the work we have done during these four months.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

Från att endast ha varit en maskin som användes för att göra avancerade beräkningar har datorn utvecklats mer och mer mot en underhållningsmaskin. Nu för tiden är det inte ovanligt att en dator inhandlas med enda syftet att det skall spelas spel på den. Framförallt på senare tid har grafiken blivit den dominerande faktorn när man väljer vilket spel man vill köpa. Därför är det viktigt att skapa så bra förutsättningar som möjligt för att grafiken skall hålla så hög kvalitet som möjligt.

När en bild skall visas på skärmen måste den först laddas in i grafikkortets minne, grafikminnet. När sedan, som i ett datorspel, hela världar visas på skärmen måste man ladda in stora mängder bilder, eller texturer som det kallas i 3d spelens värld. Detta leder till att ju större och mer detaljerade världar som skall visas desto mer grafikminne krävs. Och eftersom minne är relativt dyrt ställs det höga krav på att minnet skall utnyttjas så optimalt som möjligt. Man vill helt enkelt inte slösa med det minne man har tillgång till eftersom det knappt räcker till ändå.

Det är idag mycket ovanligt att ett datorspel utnyttjar allt tillgängligt grafikminne och på så sätt ritar upp datorvärldarna med bästa möjliga kvalitet. Anledningen till detta är att det är mycket svårt för grafikerna att skapa texturer till objekt i världarna så att dessa utnyttjar grafikminnet på bästa möjliga sätt.

Det är även vanligt att ett spelföretag utvecklar ett spel för flera olika plattformar, exempelvis Playstation 2, PC med flera. Därför varierar storleken på grafikminnet väldigt mycket, från några Megabyte ända upp till 100-tals Megabyte. När spel utvecklas till PC kan man heller inte på förhand veta hur mycket grafikminne som finns på varje dator som spelet kommer att spelas på. På grund av detta är det därför omöjligt för grafikerna att skapa texturer som passar alla olika hårdvaror perfekt. Grafikerna skapar texturer i väldigt hög upplösning och sedan beroende på vilken hårdvara spelet skall köras på sänks storleken och därmed kvaliteten på texturerna, så att de får plats i grafikminnet.

Det vanligaste sättet att anpassa kvaliteten på texturerna till storleken på minnet är att börja med att försöka ladda in texturerna i högsta kvalitet. Om inte alla texturer skulle få plats sänks storleken på alla texturer till en fjärdedel, halva bredden och halva höjden, och man försöker ladda in dem igen. Denna procedur upprepas till dess att alla texturer får plats i minnet. Detta gör att utnyttjandet av grafikminnet oftast inte blir optimalt. Här följer ett litet exempel för att

illustrera det hela. Om 16 Megabyte (Mb) grafikminne finns ledigt och texturer med en sammanlagd storlek på 17 Mb skall laddas in, kommer dessa inte att få plats i minnet. Då skalas de ner till en fjärdedel, i det här fallet till ca 4 Mb. När dessa sedan har laddats in i minnet är kvaliteten på grafiken mycket sämre men ändå finns 11 Mb outnyttjat grafikminne, vilket inte är tillfredställande.

Detta arbete har gått ut på att lösa detta problem genom att utveckla ett system som anpassar kvaliteten på texturerna till hur mycket ledigt grafikminne man har på ett bättre sätt. Arbetet baserades till största delen på ett internt dokument från företaget Digital Illusions (Dice) som arbetet utfördes åt. Det interna dokumentets namn är ”Texture database design draft” och är författat av Daniel Hansen.

Uppsatsen är organiserad enligt följande:

- Kapitel 2 beskriver en del bakgrundsfakta som behövs för att kunna sätta sig in i det arbete vi har gjort.
- Kapitel 3 behandlar syftet med arbetet och vilka problem vi tänker lösa med vårt arbete.
- Kapitel 4 beskriver designen av vårt system.
- Kapitel 5 behandlar implementationen av vårt system.
- Kapitel 6 tar upp lite om vad vi har lärt oss av vårt arbete samt vad vi hade kunnat göra annorlunda. Sist i detta kapitel sammanfattas vårt arbete och slutsatser dras.

2 Bakgrund

I detta kapitel tar vi upp lite bakgrundsfakta om 3d spel och hur dessa är uppbyggda. Detta för att bättre kunna förstå vad vi har utvecklat och varför vi har gjort det. De olika delarna i detta kapitel kommer också att refereras till när vi senare förklarar vårt arbete.

2.1 Tredimensionella datorspel

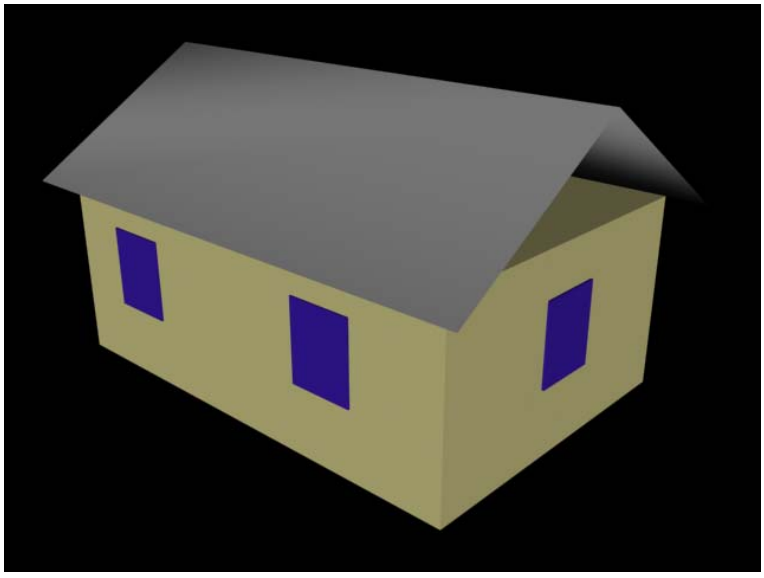
Datorspel som använder sig av tredimensionell grafik har blivit väldigt populära på senare år. Spel som Quake, Counterstrike och Rally Masters använder sig alla av tredimensionell grafik för att rita upp spelplanen. I tredimensionella spel beskrivs miljön och spelfigurerna som objekt i en tredimensionell rymd som vi kan se i Figur 2.1.

Figur 2.1: En typisk 3d modell

När sedan spelaren väljer att flytta sin figur eller när något sker i spelet görs en mängd matematiska beräkningar som rotationer och förflyttningar. Därefter ritas spelplanen upp på skärmen igen. Innan grafikkort med stöd för tredimensionell grafik, eller 3d-grafik, fanns till rimligt pris, gjordes både beräkningar och uppritning av 3d miljöerna i spelen av datorns processor. Med tanke på att processorn måste göra en mängd andra beräkningar och att de dessutom inte var så snabba, kunde inte miljöerna ritas upp speciellt realistiskt och detaljerat för ett tiotal år sedan. Nuförtiden finns stöd för 3d-grafik på de flesta grafikkort som säljs. Dessa grafikkort är utrustade med snabba uppritnings- och beräknings-enheter så datorns processor inte behöver göra dessa beräkningar längre, utan kan ägna sig åt andra uppgifter. Grafikkortens uppritnings- och beräknings-enheter har även stöd för mer avancerade effekter som exempelvis ljussättningsberäkningar och texturmappning, vilket beskrivs nedan.

2.2 Texturer

När något modelleras som ett tredimensionellt objekt i en dator är det svårt att få med alla detaljer. Om exempelvis ett tegelhus med tak av sten skall modelleras kan en modell med gula väggar, blå fönster och grått tak skapas (se Figur 2.2).

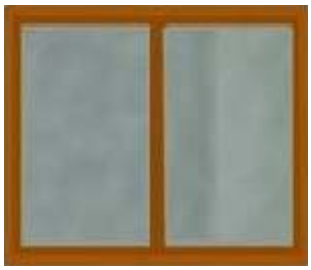


Figur 2.2: En 3d modell utan texturer

Själva modellen påminner om huset men den ser absolut inte ut som tegelhuset ser ut i verkligheten. Ett sätt att få objektet att se mer verklighetstroget ut är att ”lägga på” bilder på

objektets sidor. För detta objekt kan bilder på tegel, fönster och sten användas (Figur 2.3 - Figur 2.5).

Figur 2.3: Tegeltextur



Figur 2.4: Fönstertextur

Figur 2.5: Stentextur

Figur 2.6: 3d modell med texturer

När dessa bilder används som ytor på objektets sidor kallas de för objektets texturer (se Figur 2.6).

Texturer skapas som sagt utifrån bilder som för det mesta ritats i ett ritprogram. Dessa texturer kan vara i färg, i gråskalor, i olika storlekar och helt eller delvis genomskinliga. Denna genomskinlighet används bland annat till att få ett fyrkantigt objekt med en textur på att inte se helt fyrkantigt ut. Genom att välja vissa delar av en textur till att vara genomskinliga uppfattas objektet som att inte vara fyrkantigt fastän det kanske i själva verket är det. I den högra delen av Figur 2.8 ser vi hur denna teknik används för att få en gran som är modellerad som en fyrkantig sida med en textur på att se ”taggig” ut som en gran.

Figur 2.7: Grantextur

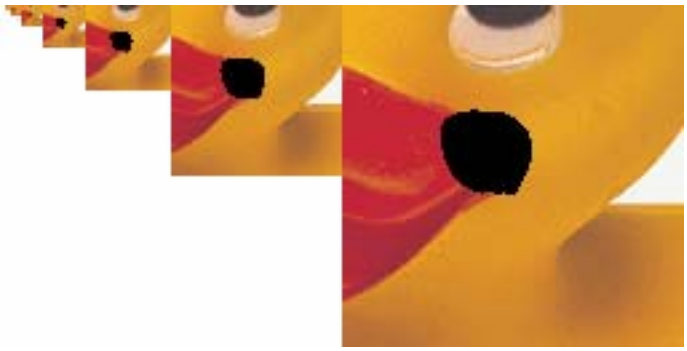
Denna effekt uppnås genom att till grantexturen (Figur 2.7) lägga till en textur med samma dimensioner som anger genomskinligheten för granen. Denna textur kallas för alfatextur och den visas i den vänstra delen av Figur 2.8. Svart betyder helt genomskinligt och vitt betyder helt solitt, alltså inte genomskinligt över huvud taget.

Figur 2.8: Till vänster texturens alfakanal, till höger objektet med texturen

Det vi nyss skrev var egentligen en liten lögn. Texturen som anger genomskinligheten är egentligen inte en separat textur utan tillhör grantexturen och kallas för texturens alfakanal. För att lättare förstå hur det fungerar kan man tänka dem som två olika texturer.

2.2.1 Mipmapping

Mipmapping är en teknik som används för att grafiken i ett 3d spel skall flyta renare och snyggare. Det innebär att man använder sig av flera olika storlekar av samma textur. Allt från den storleken texturen har i originalutförandet till en textur som är 1x1 pixel stor (se Figur 2.9). Alla storlekar som används är i storleken 2^x x 2^x. Det innebär att storlekarna är 1x1, 2x2, 4x4 och så vidare upp till den största storleken. Varje sådan storlek på texturen kallas för en mipmapnivå, där 0 är den lägsta nivån som är 1x1 pixel stor. Detta görs för att olika storlekar på texturen skall kunna användas beroende på hur långt bort texturen är. Ett exempel för att illustrera varför det kan behövas följer här. Om vi har en vägg där det sitter ett fönster på ser det bra ut medan fönstret är så nära så att det är större eller lika stort som originaltexturen. Om man sedan förflyttar sig i miljön så att fönstret försvinner bort från kameran visas det inte med samma antal pixlar som det i själva verket innehåller. Alla pixlar får helt enkelt inte plats. Om vi nu säger att vi flyttar oss ganska långt från fönstret, så att storleken skulle vara hälften i x-led och hälfen i y-led, det vill säga en fjärdedel så stort. Då får endas en av fyra pixlar i originaltexturen plats på skärmen. Vilka pixlar väljs då? Innan tekniken med mipmapping uppfanns valdes en pixel slumpmässigt bland de fyra pixlar som krockade. Och samma pixlar valdes heller inte varje gång. Detta ledde till att flimrande pixlar uppstod i fönstret i vårt tidigare exempel och detta var inte alls snyggt. Med mipmaptekniken sparas istället flera olika storlekar av texturen. Alla storlekar är av dimensionerna 2^x x 2^x alltså 1x1, 2x2, 4x4 och så vidare. På detta sätt byts istället texturen ut till en mindre variant när den är så långt bort att det skulle kunna börja flimra. Dessa mindre versioner av texturen framställs genom att genomsnittet tas av pixlarna i originaltexturen som blir en pixel i den förminskade texturen.



Figur 2.9: En fullständig mipmapkedja av en textur

2.2.2 Bilinjär filtrering

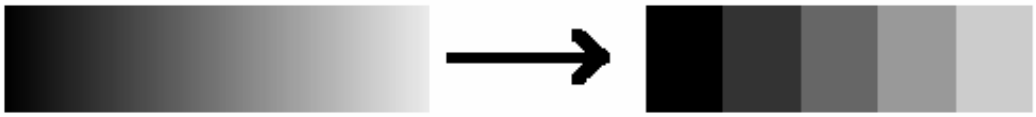
Filtrering är ett sätt att utjämna skarpa konturer. Detta är bra att använda när en bild förstoras och man inte vill att det skall synas så tydligt. I Figur 2.10 visas ett exempel på bilinjär filtrering:

Figur 2.10: Uppförstorad bild utan och med filtrering

När en bild förstoras utan filtrering, upprepas bara pixlarna i originalbilden flera gånger i den uppförstorade bilden. Effekten av detta blir som den övre bilden i Figur 2.10. I detta fall syns tydligt att bilden är uppförstorad. Istället för att bara upprepa pixlarna vid uppförstoringen kan man även låta de angränsande pixlarna vara med och bestämma färgen på den uppförstorade pixeln. Detta kallas att förstora en bild med bilinjär filtrering. Resultatet av denna förstoring visas i den nedre bilden i Figur 2.10.

2.2.3 Ditrering

Ditrering används när en textur konverteras från något format till ett annat format med färre antal färger. I Figur 2.11 ser vi resultatet av konverteringen utan att ditrering används.



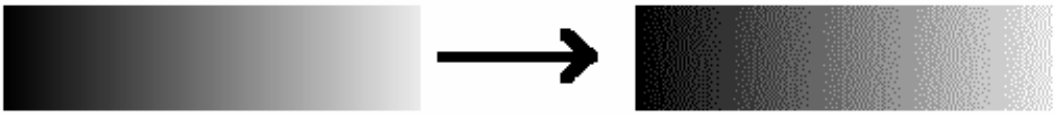
Figur 2.11: Konvertering utan ditrering

När man gör denna konvertering får man i varje pixel en differens i färgvärde mellan originaltexturen och den konverterade texturen. Istället för att bara strunta i detta fel, som vanligtvis görs, kan det fördelas till de kringliggande pixlarna. Det är detta som kallas

ditrering. Det vanligaste sättet att fördela felet på är till pixeln till höger som får 7/16 av felet, och de tre pixlarna nedanför som får del av felet enligt följande:

	Aktuell Pixel	7/16
3/16	5/16	1/16

Denna metod kallas för Floyd Steinberg ditrering. I Figur 2.12 visas resultatet av en konvertering där ditrering används.

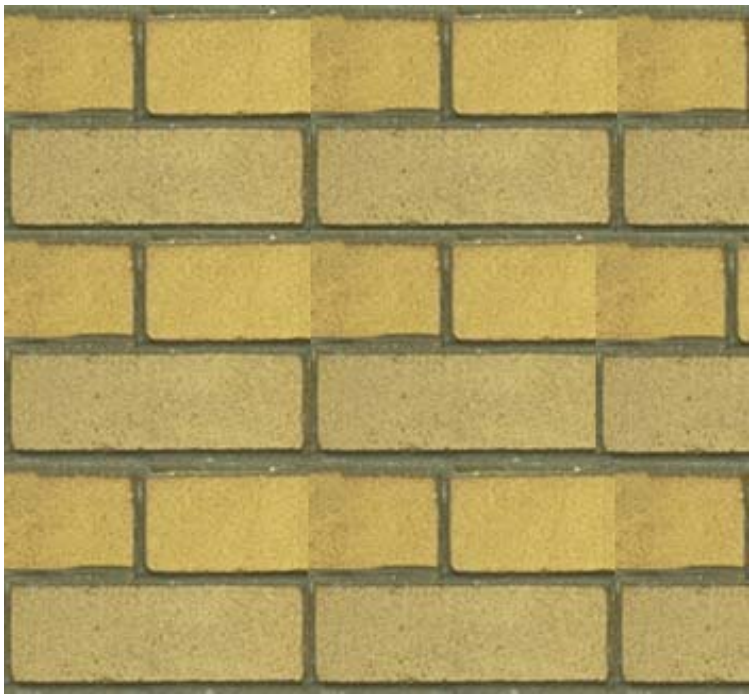


Figur 2.12: Konvertering med ditrering

2.2.4 Wrap

Att en textur är ”wrappad” innebär att den används ett multipelt antal gånger på sidan av ett objekt. Som ett exempel kan vi säga att vi har en vägg som skall vara fylld av en textur med tegelstenar (se Figur 2.14). I stället för att texturen avbildar varje sten består texturen av en liten del av en vägg (se Figur 2.13). Sedan används denna textur upprepade gånger sida vid sida så att det ser ut som om man bara har en stor textur med ett antal tegelstenar. Detta kallas ”wrap texture address mode” i DirectX.

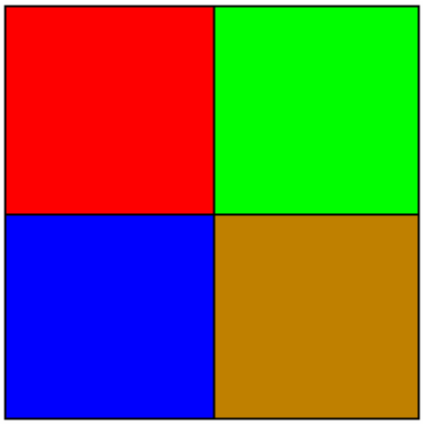
Figur 2.13: Textur lämplig att wrappa



Figur 2.14: Texturen i Figur 2.13 wrappad på en hel vägg

2.3 Texturformat

En tvådimensionell bild representeras vanligtvis i en dators minne som en matris med färgvärden. Antalet rader och kolumner i matrisen bestämmer storleken på bilden. Varje element i matrisen kallas för en pixel. En pixel består vanligtvis av en röd, en grön och en blå del. Varje del eller så kallad kanal anger intensiteten för varje färg i pixeln. Dessa färger kan kombineras med olika intensitet och på detta sätt kan fler än de tre grundfärgerna återskapas. Om exempelvis intensiteten för röd, grön och blå kanal väljs till 75%, 50% respektive 0% fås en brun nyans. I Figur 2.15 visas en bild som är 2 pixlar bred och hög.



Figur 2.15: Fyra pixlar

Pixlarna representeras i datorns minne som matrisen nedan.

100% Röd, 0% Grön, 0% Blå	0% Röd, 100% Grön, 0% Blå
0% Röd, 0% Grön, 100% Blå	75% Röd, 50% Grön, 0% Blå

Tabell 2.1: Färgvärden för pixlarna i Figur 2.15

En textur skapas utifrån en bild. Den kan skapas i en mängd olika format beroende på vad den skall användas till eller vilken kvalitet man vill ha på texturen. Är bilden som texturen skapas utifrån svartvit behövs bara en bit per pixel för att beskriva texturen. Innehåller bilden en mängd olika färger och dessutom information om alfakanal krävs däremot ett flertal bitar per pixel för att beskriva texturen. I många fall gäller att ju mer minne texturen tar i anspråk desto bättre återskapning av bilden sker.

2.3.1 Truecolor texturer

Detta är det vanligaste och enklaste formatet. Varje pixel i detta format är uppdelad i en röd, en grön, en blå och eventuellt en alfadel. Varje del anger intensiteten för respektive kanal i pixeln. Använder man 8 bitar per kanal kan intensiteten för varje kanal anges i $2^8 = 256$ steg. Ju fler bitar som används per kanal desto bättre kvalitet får texturen, men samtidigt krävs mer minne för att rymma texturen.

2.3.2 Dxt komprimerade texturer

Om en 32 bitars textur skall lagras i videominnet krävs 4 bytes per pixel. Åtta bitar, eller en byte, för var och en av de fyra kanalerna röd, grön, blå och alfa. Om vi antar att en textur är 1024 x 1024 pixlar stor så blir det totala minnesutrymmet som bara denna textur upptar ca 4,2 Mb. Sedan kan man betänka att man kan vilja lagra hundratals texturer i videominnet samtidigt, samt att videominnet oftast är begränsat till 32 Mb eller 64 Mb. Då förstår man ganska snabbt att man gärna vill ha något sätt att komprimera dessa texturer så att de tar mindre plats utan att förlora för mycket av sin kvalitet.

Detta problem löstes av företaget S3, som i huvudsak tillverkar grafikkort, då de tog fram en komprimeringsteknik som de kallar för S3TC. När denna sedan såldes till Microsoft för att användas som standard komprimering i deras utvecklingsverktyg DirectX döptes teknologin om till DXT. Eftersom DirectX är ett programmeringsgränssnitt som utan kostnad kan laddas hem via internet och därmed fritt utveckla applikationer med, har DXT-teknologin spridit sig till de flesta grafikkortstillverkare.

DXT är en så kallad förstörande komprimering, vilket innebär att bildkvaliteten försämras, men inte alls lika mycket som storleken på texturerna minskar. När man utvecklade DXT hade man tre huvudsakliga mål:

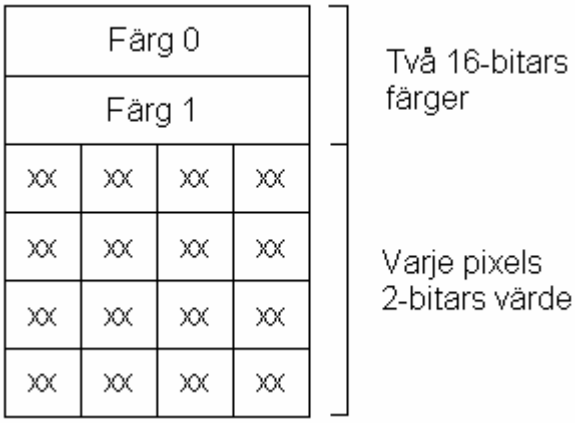
- Det skulle vara snabbt och enkelt att dekomprimera de komprimerade texturerna. Detta mål hade man för det skulle vara relativt billigt att implementera en dekomprimering i hårdvaran, och att det samtidigt skulle vara väldigt snabbt att dekomprimera.
- Texturerna skulle komprimeras relativt mycket så att det var någon mening med att komprimera dem.
- Det tredje målet var att bildkvaliteten på texturerna inte skulle sjunka så mycket att det syntes då komprimerade texturer användes i 3d spel.

DXT finns i fem olika varianter, men vi har valt att endast implementera tre av dem i vårt system och därför kommer vi endast att gå igenom hur de tre fungerar här, dock är det inte några stora skillnader på de andra två varianterna som vi inte går igenom här.

De tre varianter som vi har fördjupat oss i är DXT1, DXT3 och DXT5. Skillnaderna mellan dessa tre ligger endast i hur alfa kanalen i texturerna hanteras och komprimeras.

DXT1:

I samtliga DXT varianter delas texturen upp i s.k. DXT-block. Dessa består av ett block med 4x4 pixlar ur texturen som hanteras samtidigt. Det som sedan genereras vid komprimeringen av ett sådant block är 2 stycken 16 bitars färgvärden samt 2 stycken bitar för varje pixel som beskriver vilken färg de skall ha (se Figur 2.16). Färgerna som sparas är i formatet 5:6:5. Att de är i formatet 5:6:5 innebär att den röda färgen tar upp 5 bitar, den gröna tar upp 6 bitar och den blå tar upp 5 bitar.



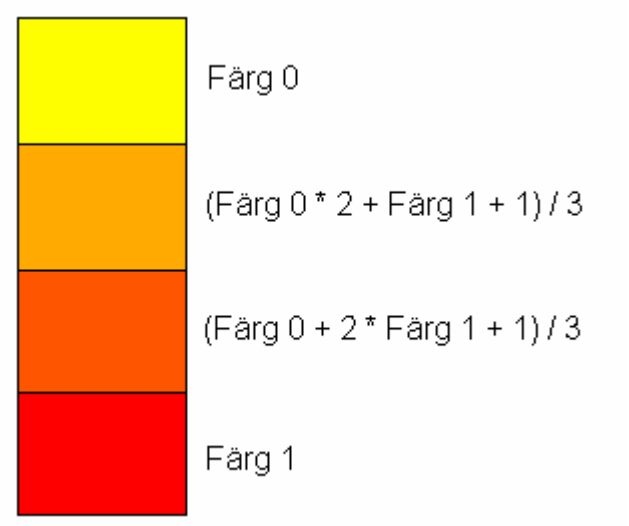
Figur 2.16: Ett DXT1 block

Alltså sparas 8 bytes information för varje block om 16 pixlar. Om man jämför med storleken på blocket innan det var komprimerat är det stor skillnad, här följer ett exempel: Vi antar att varje kanal röd, grön och blå är 8-bitar och ingen alfakanal används, alltså varje pixel är 24 bitar eller 3 bytes. I blocket ingår sedan 16 pixlar, det vill säga 16 pixlar * 3 bytes = 48 bytes. Med andra ord är storleken på det komprimerade DXT-blocket bara en sjättedel så stor som det var innan blocket komprimerades. Då förstår man hur viktig komprimeringen är då minnesutrymmet är begränsat.

Från de två färgerna som genererades vid komprimeringen tar man sedan fram två stycken färger till som är interpolerade från de två första. Det innebär att det är en blandning av de två. Då har man alltså fyra färger och det är endast dessa fyra färger som pixlarna i blocket kan ha efter att de har dekomprimerats igen. Det är också här de två bitarna som genererades för

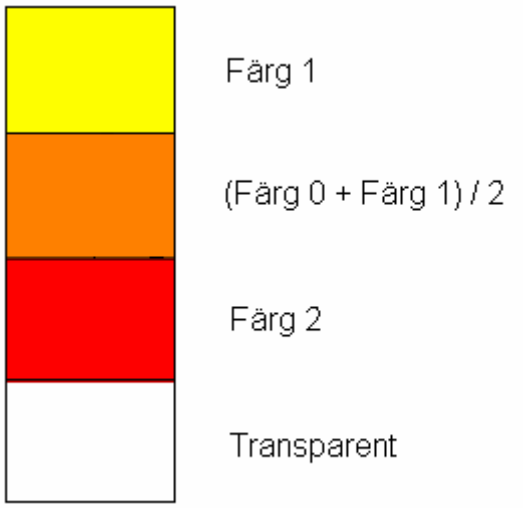
varje pixel kommer till användning. Med två bitar kan man representera fyra olika värden, dessa bitar anger alltså vilken av de fyra färgerna en viss pixel skall ha.

Det finns också möjlighet att använda sig av en 1-bits alfakanal i DXT1. Det går till så att en av de fyra kombinationer man kan få fram från de två bitarna varje pixel i blocket har representerar alfakanalen. När en alfakanal används kan man alltså bara ange vilken av tre olika färger en pixel skall ha, eller om den skall vara helt transparent. De alternativ man har i genomskinlighet är alltså helt transparent eller inte transparent över huvud taget. För att markera om man använder 1-bits alfa eller inte används ordningen i vilken de två färgerna är sparade. Om den första färgen har ett högre heltalsvärde än den andra färgen innebär det att man använder en 1-bits alfakanal samt tre färger för att återskapa texturen. Annars används som sagts tidigare fyra färger, de två sparade färgerna samt två stycken interpolerade färger mellan dessa. Ett exempel på hur interpolering fungerar när ingen alfakanal används ges i Figur 2.17.



Figur 2.17: DXT1 färger interpolerade utan alfa

I Figur 2.18 visas hur samma färgschema skulle se ut om man använde en enbits alfakanal i blocket.



Figur 2.18: DXT1 färger interpolerade med alfa

DXT3:

Skillnaden för DXT3 gentemot DXT1 är endast hur alfakanalen hanteras och komprimeras. I DXT3 används inte en 1-bits alfakanal utan i stället en 4-bits alfakanal. Det innebär att man alltid använder sig av fyra färger i varje block, varav två är interpolerade. Det innebär också att varje DXT-block är dubbelt så stort för DXT3 jämfört med DXT1. Att varje DXT-block är dubbelt så stort beror på att för varje pixel genereras ett alfavärde som är 4-bitar stort (se Figur 2.20), alltså totalt 64-bitar för de 16 pixlarna. Den totala storleken på DXT-blocket blir då 128-bitar eller 16 bytes. Komprimeringen av färgerna i texturen sker på exakt samma sätt som i DXT1.

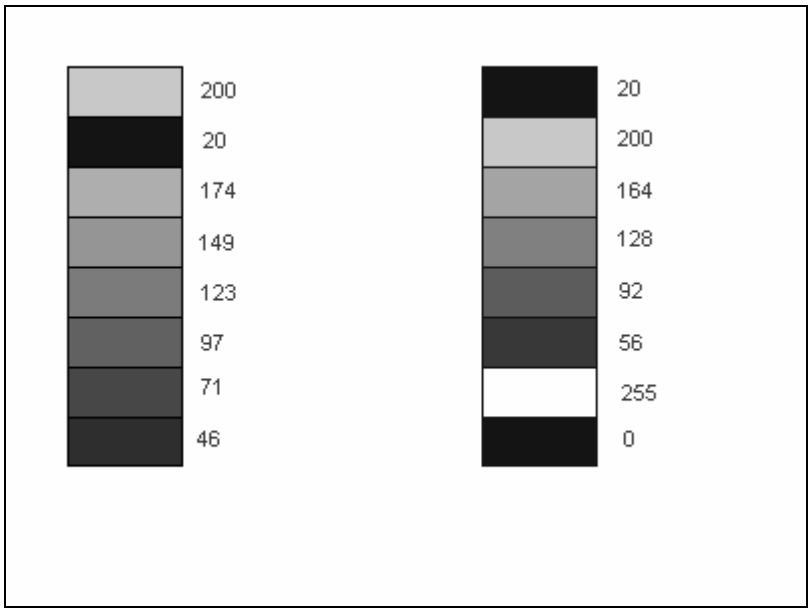
För att återskapa alfakanalen sparas som sagt 4-bitar per pixel. Dessa bitar kan väljas ut på olika sätt. Det enklaste och vanligaste sättet är att man tar de 4 mest signifikanta bitarna i den 8-bitars alfakanal, som den okomprimerade texturen innehåller. Detta innebär att varje textur som komprimeras endast kan innehålla 16 olika alfavärden mot 256 för en okomprimerad bild.

DXT5:

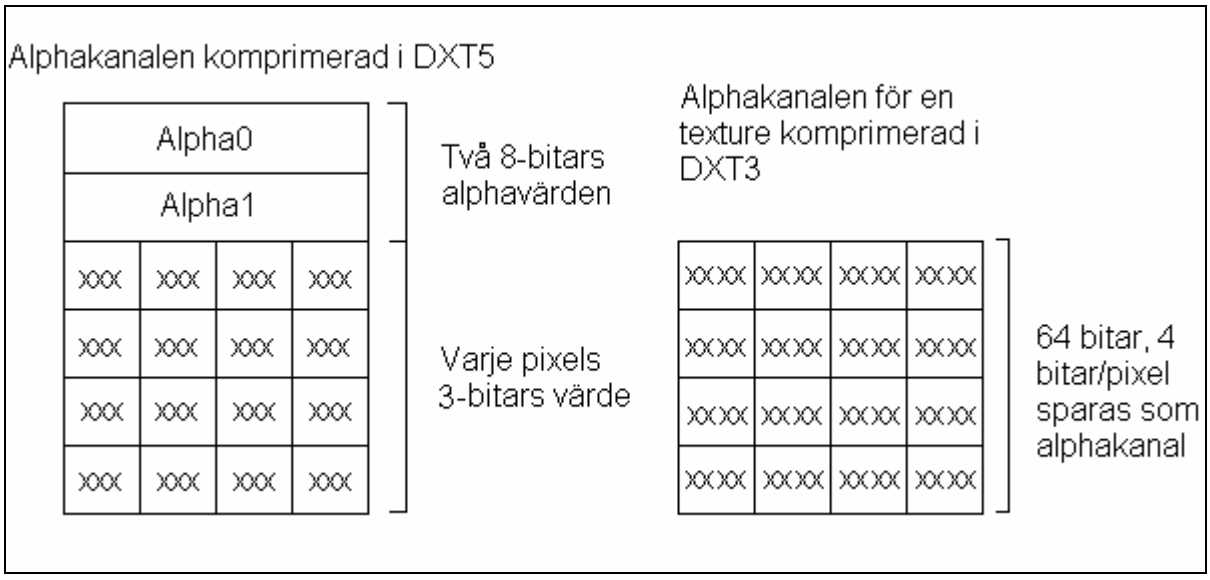
Skillnaden mellan DXT5 och DXT1/DXT3 är precis som skillnaden mellan DXT1 och DXT3 hur alfakanalen komprimeras och sparas. I DXT5 komprimeras färgerna precis som i DXT3. Även här precis som i DXT3 sparas också 8 bytes per DXT-block som beskriver alfakanalen. Precis som när färgerna komprimeras så tas de två 8-bitars alfavärden fram som bäst kan

beskriva alfablocket och sparar dessa. Sedan sparas för varje pixel i blocket 3-bitar som beskriver hur mycket av varje alfavärde som pixeln skall innehålla (se Figur 2.20). På detta sätt kan man för varje DXT-block använda sig av 8 olika alfavärden och för varje textur kan använda sig av 256 stycken. Detta ger förstås bättre resultat än när endast de 4 mest signifikanta bitarna används som oftast görs i DXT3. Eftersom alla alfavärden interpoleras mellan de två alfavärden som bäst beskriver bilden kan det bli problem om någon pixel skall vara helt solid eller helt transparent. Detta problem har dock löst på så sätt i DXT5 att om någon pixel skall vara helt solid eller helt transparent används inte 8 alfavärden, utan man använder sig av de 2 som sparats, 4 som interpoleras mellan de 2 sparade samt ett värde för helt solid och ett för helt transparent. Vilken typ av komprimering som skall användas för ett block, 8 interpolerade eller 6 interpolerade, 1 solid och 1 transparent avgörs genom i vilken ordning de två alfavärdena sparas.

I Figur 2.19 visas ett exempel på hur de olika alfavärdena interpoleras beroende på om någon pixel skall vara helt transparent eller helt solid. I båda fallen är alfavärdena 20 och 200 de som bäst återskapar blocken och det är utifrån dessa övriga alfavärden interpoleras. Siffrorna till höger om blocken anger alfaintensiteten.



Figur 2.19: Interpolation av alfavärden i DXT5



Figur 2.20: Beskrivning av hur alfavärden sparas för DXT3 och DXT5

2.3.3 Färgtabellsmappade texturer

I detta format är inte varje pixels värde en direkt färg utan ett index in i en tabell som innehåller de verkliga värdena på de olika färg- och alfakanalerna. Tabellen väljs vanligtvis till att innehålla 256 olika färgvärden. På detta sätt kan varje pixel i texturen vara exakt en byte stor, eftersom en byte kan anta just 256 olika värden. Om varje pixel skall innehålla det verkliga värdet för alla kanaler, som är fallet i truecolor formatet, krävs 4 bytes per pixel, om 8 bitar = 1 byte per kanal används. Med färgtabellsmappade texturer krävs bara 1 byte per pixel och en tabell med 256 färgvärden.

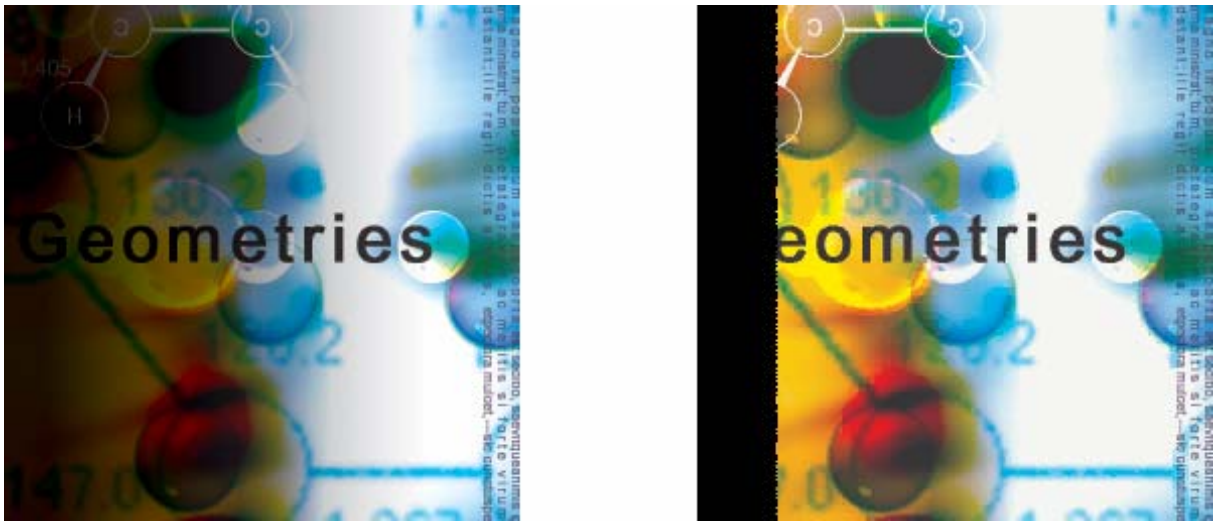
2.3.4 Texturformatens minnesanspråk

Om en textur skall skapas utifrån en bild som innehåller en mängd färger krävs det olika mycket minne beroende på vilket texturformat som väljs. Även formatets precision spelar roll. Till exempel kräver en textur som är i truecolor format mer minne om varje kanal i texturen skall representeras med 8 bitar än om varje kanal bara skall representeras med 5 bitar. Mer minne krävs även om DXT3 formatet väljs istället för DXT1 formatet. I Tabell 2.2 gör vi en sammanställning som visar hur mycket minne en textur som är 256 pixlar hög och bred och även använder sig av en alfakanal tar i anspråk.

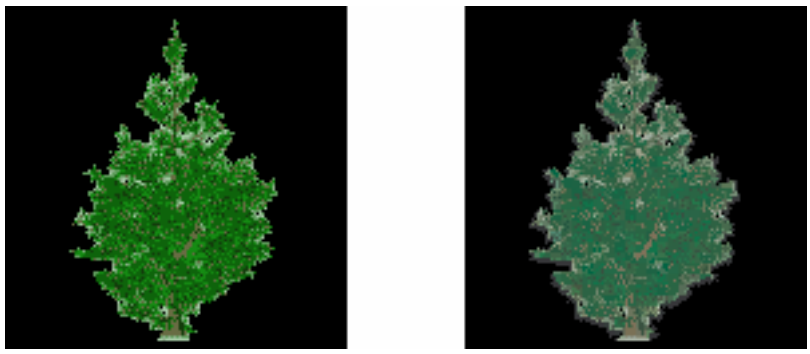
Texturformat	**Antal bitar per kanal**	**Storlek i kilobytes**
Truecolor	R8G8B8A8	256
Truecolor	R5G5B5A1	128
DXT1	R5G6B5A1	32
DXT3	R5G6B5A4	64
DXT5	R5G6B5A4	64
Färgtabellsmappad	R8G8B8A8	65

Tabell 2.2: Storleksjämförelse mellan olika texturformat

Kolumnen för antal bitar per kanal i Tabell 2.2 kräver en förklaring. Exempelvis betyder R5G6B5A4 att fem bitar används för den röda kanalen, sex för den gröna, fem för den blå och slutligen används fyra bitar för alfakanalen. I tabellen ser vi hur stor skillnad det är i minnesanspråk mellan de olika formaten. En textur i truecolor format med 8 bitar per kanal tar 8 gånger så stor plats i minnet som en textur i DXT1-format. Eftersom DXT1-texturen kräver så mycket mindre minne är det rimligt att tro att man fått ge avkall på något. Det är oftast kvaliteten på texturen som har blivit sämre. Används ett stort spektrum av färger inom en begränsad yta av texturen och en alfakanal som varierar i många nivåer syns ofta stor skillnad mellan formaten (se Figur 2.21), men används bara ett fåtal färger och bara helt genomskinlig eller ogenomskinlig textur är skillnaden mellan texturerna knappt märkbar. I texturerna nedan illustrerar vi detta. Lägg märke till att när det finns många olika gröna nyanser som i Figur 2.22 tappar texturen väldigt mycket i kvalitet då den komprimeras. Dock är alfan identisk i både originaltexturen och i den komprimerade. Detta eftersom det endast används två olika alfavärden.



Figur 2.21: Till vänster visas en okomprimerad textur, till höger visas samma textur komprimerad med DXT1. Det svarta på bilderna är alfakanalen.



Figur 2.22: Till vänster originaltextur, till höger textur komprimerad med DXT 1.

2.4 Bildformatet Targa

Targa[1] (TGA) är ett filformat som används för att lagra färgbilder. Formatet definierades av företaget Truevision i mitten av 1980-talet och används nuförtiden av många olika grafiska applikationer. Targaformatet är applikations- och plattforms-oberoende, vilket innebär att man kan rita och spara en bild i TGA format i t.ex. Photoshop på en IBM PC kompatibel dator. Den sparade TGA filen kan sedan öppnas och bearbetas i en applikation som stöder TGA formatet på en Macintosh. Detta underlättar arbetet hos bland annat ett spelutvecklingsföretag där 2d-grafikerna ritar sina bilder i ett bildhanteringsprogram och 3d-grafikerna arbetar i ett helt annat program. Om bägge applikationerna har stöd för TGA formatet kan bilder skapade av 2d-grafikerna smidigt importeras som texturer i 3d-miljöerna som 3d-grafikerna skapat. TGA formatet har stöd för Truecolor-, Färgtabellsmappade- och gråskale-bilder, och man kan

välja att spara bilderna okomprimerade eller runlength komprimerade. Runlength komprimering är en icke-förstörande komprimering som effektivt komprimerar data som är konstant under längre perioder. Detta är ofta fallet i bilder skapade för hand, d.v.s. inte foton tagna av en kamera och överförda till en dator. Även en alfa kanal kan användas, vilket är ett måste för att formatet skall vara användbart i vår applikation.

2.5 DirectX

För att kunna spela de flesta spel som kommer ut på marknaden idag krävs någonting som kallas DirectX [2]. Vad är då detta DirectX som man behöver installera för att kunna spela de nya spelen?

DirectX är ett så kallat API från microsoft. API betyder Application Programming Interface och det är precis som namnet antyder ett gränssnitt för utveckling av applikationer. De applikationer som man främst använder DirectX för att utveckla är grafik- och multimedia-applikationer. I DirectX finns det ett antal funktioner för behandling av olika grafiska objekt och även för hantering av t.ex. ljud och joysticks. Den del av DirectX som vi främst använder oss av i detta arbete är en del som heter Direct3D som innehåller funktioner för hantering av 3d grafik och, framförallt i vårt fall, texturer.

3 Syfte och Problemställning

Det är ovanligt att ett datorspel utnyttjar allt tillgängligt texturminne, och därför ritas datorvärldarna sällan upp med bästa möjliga kvalitet. Anledningen till det är att det är mycket svårt för grafikerna att skapa texturerna till objekten i världarna så att dessa utnyttjar texturminnet på bästa möjliga sätt.

Det är vanligt att ett spelföretag utvecklar ett spel som skall köras på flera olika plattformar, exempelvis playstation, pc med flera. Därför varierar storleken på texturminnet väldigt mycket, från några MB ända upp till 100-tals MB. På grund av detta är det därför näst intill omöjligt för grafikerna att skapa texturerna till spelet som passar alla olika hårdvaror optimalt. Grafikerna skapar texturerna i väldigt hög upplösning och sedan beroende på vilken hårdvara spelet skall köras på får man sedan sänka kvaliteten och därmed storleken på texturerna, så att de får plats i texturminnet.

Det vanligaste sättet att anpassa kvaliteten på texturerna till storleken på minnet är att börja med att försöka ladda in texturerna med högsta kvalitet. Om inte alla texturer skulle få plats sänks storleken på alla texturer till en fjärdedel, och man försöker ladda in dem igen. Denna procedur upprepas till det att alla texturer får plats i minnet. Detta gör att utnyttjandet av de flesta texturminnen inte blir optimalt. Som vi skriver i kapitel 1 kan man i ett olyckligt fall tappa 75% i kvalitet hos texturerna samtidigt som man bara utnyttjar cirka 25% av texturminnet. Och detta är naturligtvis inte tillfredställande.

Målet med detta arbete är att skapa en bättre metod för att anpassa kvaliteten och storleken på texturerna så att texturminnet utnyttjas på ett mer effektivt sätt. Detta skall ske genom att vi skapar en texturhanterare som håller reda på alla texturer i spelet. Denna texturhanterare skall sedan avgöra hur stor försämring i bildkvaliteten olika förminskningar av texturer samt byte av bildformat leder till. Genom detta kan man sedan, om alla texturerna inte får plats i minnet, välja att minska kvaliteten på den textur som påverkas minst negativt av förminskningen.

Det vi skapar är ett filformat för texturer som består av en header där information om texturens kvalitet i olika storlekar och format finns. Filformatet innehåller också en datadel där texturen finns lagrad i den bästa möjliga kvaliteten och några övriga komprimerade format i alla olika storlekar i en mipmap kedja. För att detta filformat skall gå att använda

skapar vi ett konverteringsverktyg som gör om en targa-fil till vårt format. Detta verktyg skapar då informationen i headern som talar om kvaliteten i olika format och storlekar av texturen, samt sparar datan.

Till detta filformat skapar vi även några olika verktyg, för att underlätta för grafikerna att arbeta med formatet. De verktyg vi skall skapa är ett plug-in till Photoshop för att spara och ladda filformatet, samt ett gränssnitt för att se på texturerna i de olika formaten i Windows.

Det är också detta fil-format som läses in och hanteras av vår texturhanterare. Denna är det egentliga huvudmålet med hela vårt arbete. Den läser in informationen om texturerna från vårt filformat, när detta är gjort för alla texturer i spelet är det möjligt att generera den bästa sammansättningen av olika texturers format och storlek för att fylla texturminnet på ett tillfredställande sätt.

4 Design

Samtidigt som texturhanteraren är huvudmålet för vårt arbete är det inte denna del som är den viktigaste i projektet, utan det är filformatet. Att filformatet fungerar tillfredställande är en förutsättning för att texturhanteraren överhuvudtaget skall vara användbar. Därför har vi lagt ner mycket arbete just på att ta fram en klass för att hantera filer av vårt filformat som vi valt att kalla DTF, *Dice Texture Format*. Dessa klasser har också en väldigt central roll både i den bildvisare vi gör för att kunna visa bilder i windowsmiljö och i vår applikation som konverterar en targa bild till en DTF-fil.

4.1 Filformat för texturer

En textur bör vara lika bred som hög och bredden och höjden bör väljas till att vara 2^x där x är ett heltal som är större eller lika med noll. Detta innebär att bredden och höjden bör vara något av talen 1, 2, 4, 8, 16, 32 och så vidare. Dessa tal bör väljas för att de flesta grafikhårdvaror kräver detta, mestadels för att hög effektivitet skall kunna uppnås. Väljs inte något av dessa tal kan hårdvaran allokera minne för en textur som är så stor som nästkommande tal i serien. Detta innebär att om man vill använda sin bild som är 520 pixlar i bredd och i höjd kan minne allokeras för en textur som är 1024 i bredd och i höjd. På detta sätt går mycket minne till spillo och färre texturer får plats i minnet samtidigt.

Några av de truecolor format som vår applikation skall använda sig av är:

- R8G8B8A8
- R4G4B4A4
- R5G5B5A1
- R8G8B8
- R5G6B5
- R5G5B5

Där exempelvis R5G5B5A1 betyder att 5 bitar används för röd, grön och blå kanal och 1 bit används för alfakanalen. Detta format kräver alltså 5+5+5+1=16 bitar, eller med andra ord

2 bytes per pixel. En textur i detta format som är 128 pixels bred och hög kräver alltså 128*128*2=32 kilobytes för att lagras i minnet.

Först i vårt filformat lagras en header som innehåller en del information för att man på ett bra sätt skall kunna läsa ut och använda sig av datan som ligger i filen. I denna header står till att börja med en identifierare som visar att filen är en DTF fil. Efter denna följer vilken version av formatet DTF som filen är i. På detta följer två fält som säger vilket som är den största och vilket som är den minsta mipmapnivån av texturen som är sparad i filen. Efter detta står vilket truecolor format som originaltexturen är sparad i, och efter detta kommer ett fält med några olika flaggor. Dessa flaggor är Wrap-X och Wrap-Y som anger om en textur skall vara ”wrappad” i x-led respektive y-led. Nästa flagga som kan vara satt är ditreringsflaggan, den anger om ditrering skall användas, se kapitel 2.2.3. Nästa flagga man kan sätta är en mipmapflagga, denna anger om mipmapping skall användas eller inte. Mipmapping förklarades i kapitel 2.2.1. Den sista flaggan anger om bi-linjär filtrering skall användas eller inte, detta förklarades i kapitel 2.2.2.

Den sista posten i vår header anger hur många olika ”surfaces” som man kan generera från den aktuella filen. Det innebär hur många olika hårdvaruformat och storlekar man kan få ut från filen.

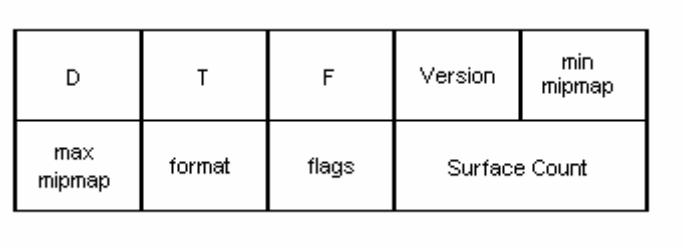
Efter headern följer sedan en lista med information om alla ”surfaces” som man kan generera från filen. I informationen för dessa står hur stora de är i bytes, ett värde på hur stort felet är på dem gentemot original texturen, vilket format de är i samt i vilken mipmapnivå de är.

När en textur sparas i vårt filformat är det inte bara i originalstorleken den sparas. Hela mipmapkedjan för texturen sparas, det innebär att alla möjliga storlekar från texturens originalstorlek till en version av texturen som är 1x1 pixel stor sparas.

Det är dock inte bara de olika truecolor formaten som filformatet innehåller. Det innehåller också data som sparats för de olika komprimerade formaten. Det som sparas för de komprimerade formaten är två färger per block för dekomprimering av DXT1, två färger per block för dekomprimering av DXT3/DXT5. Att färgerna inte är samma för samtliga tre DXT format beror på att DXT1 ibland använder en-bits alfakanal, och alltså bara tre färger per block, medan DXT3 och DXT5 alltid använder fyra färger per block. Det sparas också två stycken alfavärden per block för dekomprimering av alfakanalen i DXT5, man kan läsa mer om DXT i kapitel 2.3.2. Denna information sparas alltså för varje mipmapnivå. Det sparas också en färgtabell för varje mipmapnivå för att kunna återskapa de färgtabellsmappade

formaten utan att det skall ta för lång tid. Den tidskrävande delen av denna färgkvantisering är just att ta fram färgtabellerna.

All denna information ligger sedan varvad i filen. Varvad innebär att truecolor data för den minsta mipmapnivån ligger först. Detta följs av datan för de komprimerade formaten för samma nivå. Sedan kommer truecolor datan för mipmapnivå två följt av den komprimerade datan för denna nivå och så vidare, ända upp till den högsta mipmapnivån. Att datan ligger varvad har vi gjort därför att man inte skall behöva söka igenom hela filen om man inte är ute efter att ha hela kedjan, utan om man bara vill ha de tre minsta nivåerna skall de ligga först. Sedan behöver man inte bry sig om resterande nivåer.



Figur 4.1: Filheader för DTF-formatet

Nedan ses hur filformatets header representeras som en C-struktur:

```
struct DtfHeader {
  char id[3];                          // 'D', 'T', 'F'
  char version;                        //Version
  u8 minMipmapLevel, maxMipmapLevel;
  u8 format;                           //A, RGB eller ARGB
  u8 flags;
  u16 surfaceCount;                    //Antalet surfaces
};
```

4.2 Texturhanteraren

Syftet med texturhanteraren är att den skall ta fram den bästa konfigurationen av olika texturer beroende på hur mycket ledigt texturminne som finns att tillgå. Detta görs genom att man anger för texturhanteraren vilka texturer som skall finnas i spelet. När man har lagt till alla texturer till texturhanteraren kan man låta den generera lämplig kvalitet på texturerna för den minnesstorlek man har tillgänglig.

Metoden för att anpassa texturernas kvalitet till storleken på texturminnet är följande. När alla texturer är inladdade i texturhanteraren skapas en sorterad lista med alla texturer. I listan ligger alltid den textur som får minst kvalitetsförsämring efter en förminskning i minnesanspråk först. För att sedan anpassa texturernas kvalitet görs en kontroll om texturerna får plats i minnet om alla texturer har högsta möjliga kvalitet. Om detta inte är fallet tar man ut den textur som ligger först i den sorterade listan. Denna textur sänker man sedan kvaliteten på ett steg för att sedan sortera in den i listan igen. Efter detta görs en ny kontroll om texturerna får plats i minnet med den nya konfigureringen. Om detta inte heller är fallet upprepas förminskningen av den första texturen i listan ända tills alla texturer får plats i minnet. På detta sätt får alla texturer en jämn kvalitet.

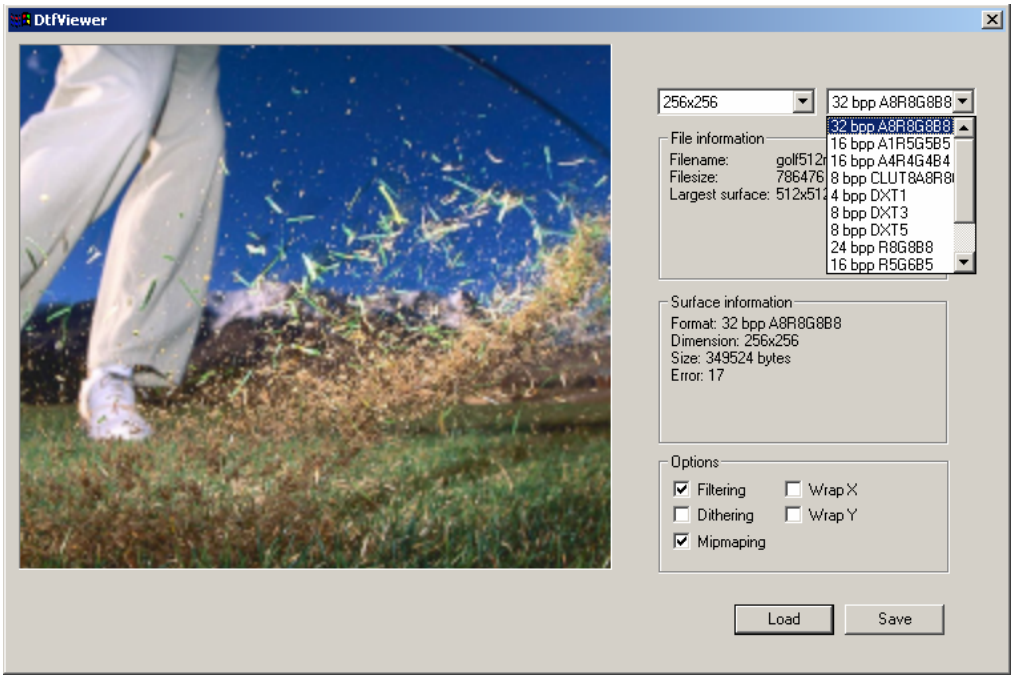
Andra funktioner som texturhanteraren innehåller är ta bort en textur man lagt till i den samt att ta bort alla texturer man lagt till.

4.3 Targa till Dice Texture Format (DTF) applikation

Denna del av projektet är till för att man skall kunna skapa texturer i DTF-format av redan färdiga bilder i targa-format. Det är ett så kallat kommandoradsverktyg, vilket innebär att det körs från kommandoprompten i ett windows/dos system. Funktionaliteten är väldigt enkel och det som sker när man väljer att konvertera en targa fil till en DTF fil är följande. Bilddatan läses ut från targafilen till ett truecolor format som innehåller åtta bitar för varje kanal. Bilddatan kan vara i en, tre eller fyra olika kanaler beroende på om man vill ha en gråskalebild eller en färgbild med alfakanal. Från denna truecolor data skapas sedan en hel mipmapkedja (se Figur 2.9) som sedan komprimeras till de tre olika DXT-formaten. Även en så kallad kvantisering görs på varje textur i mipmapkedjan och en färgtabell sparas för varje. Denna färgtabell används sedan om man vill använda sig av något färgtabellmappat format. All denna information sparas sedan i en DTF-fil.

4.4 Texturvisare för Windows

När grafiker ritar texturer så ritar de dem i en hög upplösning med ett stort antal färger. För att grafikerna skall kunna se hur kvaliteten på deras texturer blir när de konverteras till de olika formaten krävs en texturvisare (Figur 4.2).



Figur 4.2: Texturvisare för windows

I denna visare kan man välja att ladda in en Targa- eller en dtf-fil. Man kan sedan stega igenom de olika formaten och storlekarna som finns tillgängliga. För varje texturformat och storlek kan man sedan se hur mycket minne som krävs och hur stort felet är jämfört med den ursprungliga texturen. Man kan även välja att texturen skall vara ditrerad, bilinjärt filtrerad, om mipmapping skall användas eller om man vill att texturewrapping skall appliceras i x- respektive i y-led. Namnet texturvisare blev lite missvisande eftersom man i denna applikation även kan konvertera Targa-filer till dtf-filer. Detta tillägg till applikationen verkade vettigt eftersom grafikerna här kan se om texturen ser bättre ut om ditrering eller filtrering används och därefter spara texturen med dessa attribut.

4.5 Photoshop insticksprogram för laddning/sparning av dtf filer

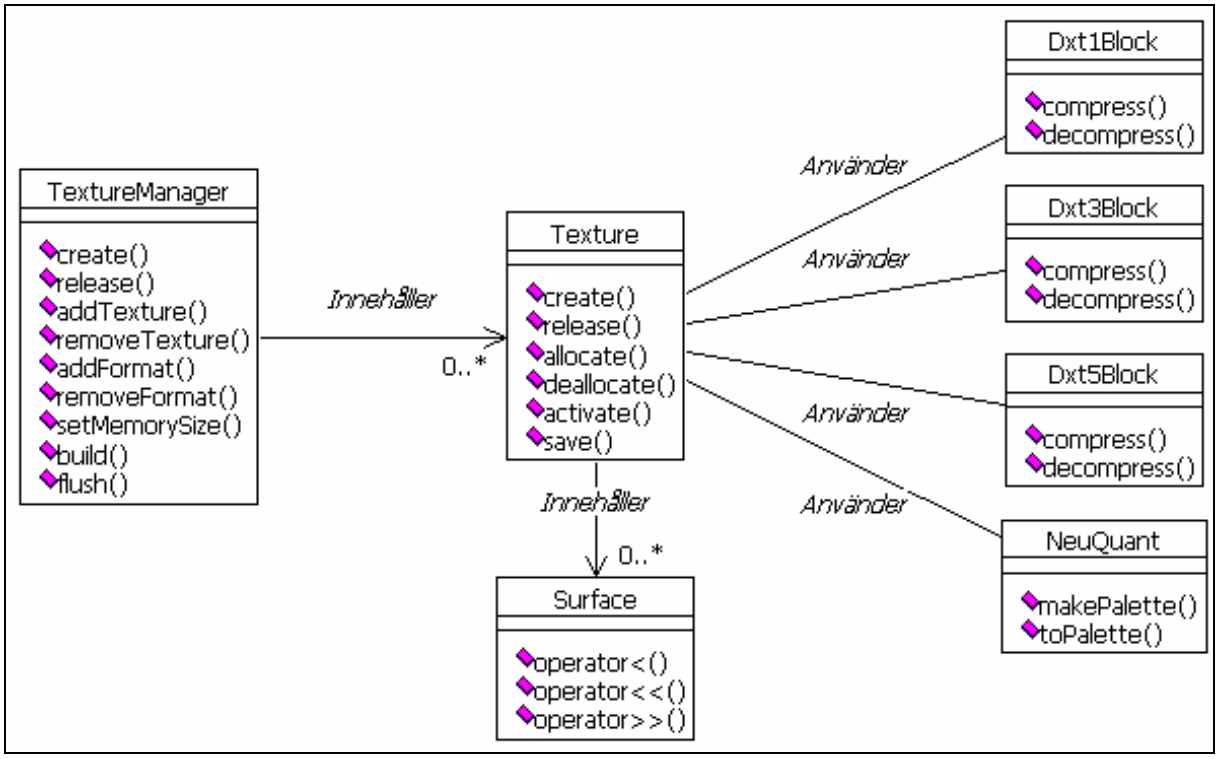
Photoshop är ett bildbehandlingsprogram som tillverkas av företaget Adobe. Det är detta bildbehandlingsprogram som används mycket vid framställning av grafiken till olika spel, därför har vi tänkt skapa ett s.k. insticksprogram till detta för att man skall kunna ladda och spara filer i vårt filformat. Ett insticksprogram är ett litet program som lägger till extra funktionalitet till ett redan befintligt program utan att man behöver några stora omkonfigurationer.

Det vårt insticksprogram är tänkt att göra är att det sparar en bild som man har tagit fram i photoshop genom att ta fram headern och datan som skall sparas. Vid sparning kommer också vissa komprimeringar av den bild man sparar att göras och på grund av detta går det inte så snabbt att spara filerna. Vid laddning av filer kommer inladdning av texturen i det bästa formatet ske för att man sedan skall kunna behandla denna och sedan spara den igen.

På grund av tidsbrist har vi inte hunnit implementera detta insticksprogram. Detta är något vi kommer att göra senare, men någon beskrivning av implementationen kommer inte att finnas med i denna rapport.

5 Implementation

I detta kapitel beskriver vi hur vi har implementerat de olika klasserna i vårt projekt. Mer hur tanken bakom de olika teknikerna vi använder oss av kan man läsa i tidigare kapitel. Här beskrivs till största delen bara hur vi har gjort vår implementation.



Figur 5.1: Klassdiagram som förklarar hur våra klasser relaterar till varandra

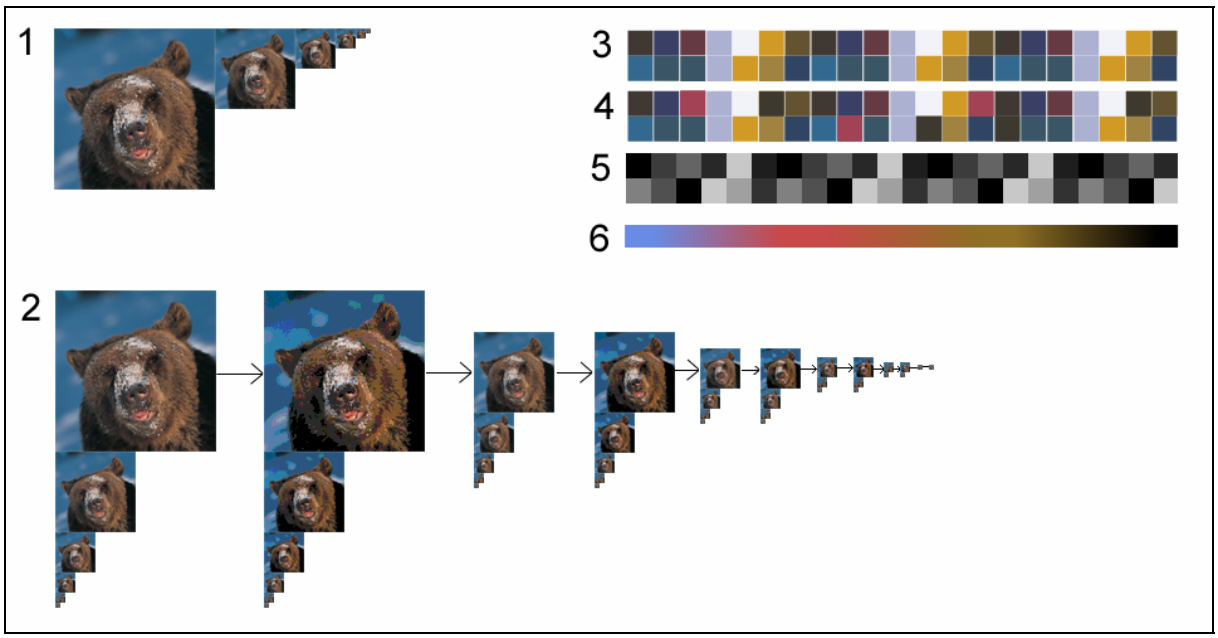
5.1 Klassen Surface

Denna klass är endast till för att hålla reda på olika varianter av texturen. Den har inga egna metoder, utan den innehåller bara data samt operatorer för in- och utströmmar. Den innehåller också en operator för jämförelse, detta därför att man skall kunna sortera Surface objekten. De data objekten innehåller är texturens högsta och lägsta mipmapnivå, texturens format, texturens storlek i bytes från lägsta till högsta mipmapnivå samt texturens felvärde i just denna variant. Klassens huvudsyfte är framförallt att användas av texture klassen, som beskrivs nedan, för att den skall kunna hålla ordning på alla olika varianter av en textur.

5.2 Klassen Texture

Denna klass hanterar allt som har med skapande och hantering av texturer i vårt texturformat att göra. Eftersom denna klass är väldigt komplex tänkte vi beskriva hur den är implementerad genom att visa ett exempel. Vi kommer att beskriva hur en textur hanteras från det att den skapas, till dess att den är färdig att laddas in i texturminnet. Detta eftersom vi tror att det är det enklaste sättet att få överblick över klassen.

Det hela börjar med att ett anrop görs till klassens create metod, i detta anrop anges antingen filnamnet till en Targa-fil eller till en dtf-fil. Alternativt kan texturen skapas direkt från data i minnet, då skickas en pekare till denna data in i metoden. Beroende på vad som skickas in i metoden händer lite olika saker, vi börjar med att beskriva vad som händer då en Targa-fil skickas in. Händelserna då är i princip samma som om ren data hade skickats in, enda skillnaden är att vid inladdning av en Targa-fil läses datan ur en fil istället.



Figur 5.2: Innehållet i ett objekt av klassen texture

När då create metoden anropas med Targa-filen som argument sker följande. Datan läses in från filen och lagras i minnet. Det är denna data som är den största bilden vid (1) i Figur 5.2. Därefter skapas alla mipmapnivåer av texturen med hjälp av metoden downsample, som beskrivs i kapitel 5.2.1. Dessa nivåer är resterande bilder vid (1) i Figur 5.2. Varje mipmapnivå lagras som ett surface vilket syns vid (2) i Figur 5.2. Därefter skapas surfaces för alla olika format som texturen kan konverteras till, till exempel DXT eller något truecolor format. Även detta kan man se vid (2) i Figur 5.2. Vid detta tillfälle framsälls även paletter för

alla olika mipmapnivåer och komprimering av texturerna till DXT görs. Dessa data lagras sedan också i objektet och kan ses i Figur 5.2 vid (3) DXT färger för DXT1, (4) DXT färger för DXT3/DXT5, (5) Alfavärden för DXT5 och (6) paletter för färgtabellsmappade texturer.

De data som finns lagrade i Textur objektet används för att ta fram den slutgiltiga texturen när man vet vilket format och vilken storlek man vill ha på den.

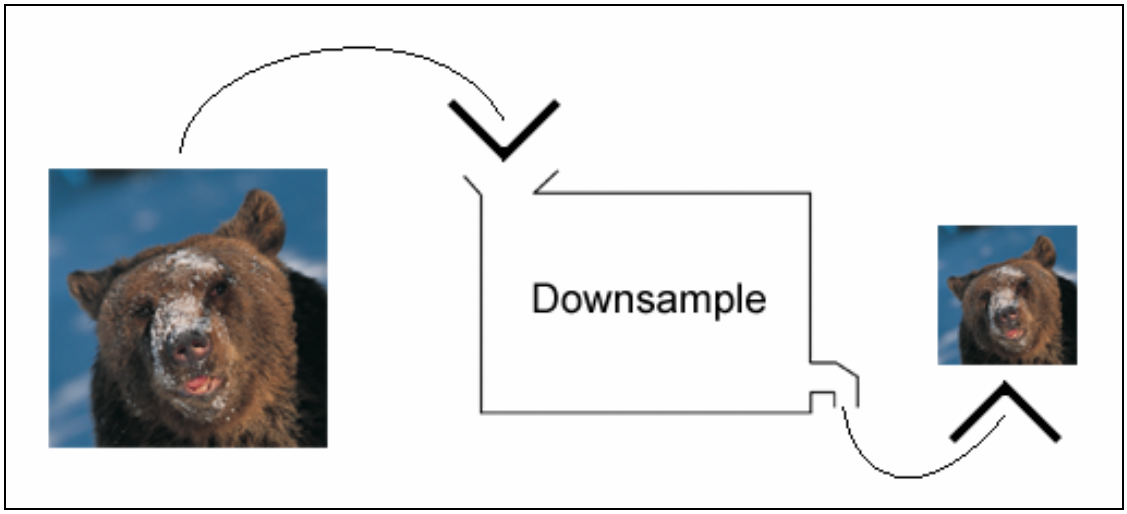
För varje nytt surface som skapas räknas ett felvärde ut, som beskriver felet mellan originaltexturen och texturen i detta surface. Detta görs med hjälp av metoden computeError som beskrivs i kapitel 5.2.3. Denna metod kräver att texturerna som jämförs är lika stora. Detta är inte alltid fallet då olika mipmapnivåer jämförs och därför måste den mindre texturen förstoras. Denna uppförstoring görs av metoden upsample, 5.2.2.

Dessa surfaces lagras i en sorterad lista där det surface som har lägst felvärde lagras först. På detta sätt kan man sedan stega igenom texturens olika surface i en ordning som ger en minskning av kvaliteten för varje steg.

Efter detta är texturobjektet färdigt att användas. Det vanliga sättet som man använder texturobjektet på är att man anropar metoden allocate som reserverar texturminne för det surface som ligger först i den sorterade listan, och flyttar texturdatan till minnet. När sedan texturen skall användas som textur på ett 3d-objekt anropar man metoden activate som säger till grafikkortet att just denna textur skall användas vid uppritningen av objektet. När man vill ta bort en textur man har laddat in i texturminnet använder man sig av metoden release.

Om man vill spara texturen till en fil finns det även en färdig metod för detta. Det den tar som argument är vilket filnamn man vill att textur-filen skall ha då den sparas. Formatet filen sparas i är Dtf, som beskrivs i kapitel 4.1.

5.2.1 Metoden downsample



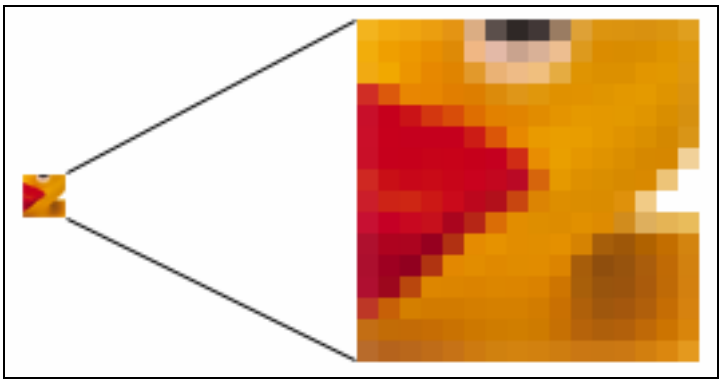
Figur 5.3: Exempel på hur en textur förändras då metoden downsample körs

Metoden downsample används som vi ser i Figur 5.3 för att förminska en textur till en lägre mipmapnivå. Det går till på så sätt att man alltid utgår från texturen i den största mipmapnivån. Man tar genomsnittet av en mängd pixlar från originaltexturen och skapar en ny pixel i en lägre mipmapnivå. Om man exempelvis vill förminska en textur som är 128 x 128 pixlar stor till en textur som blir 64 x 64 pixlar stor, kommer varje pixel i den förminskade texturen att vara genomsnittet av 4 pixlar i originaltexturen. Om man istället skulle vilja förminska texturen till 32 x 32 pixlar skulle varje pixel vara genomsnittet av 16 pixlar i originaltexturen. Att man alltid utgår ifrån den största texturen och inte bara från föregående storlek gör att vi håller felet nere på en minimal nivå. Om man däremot hade utgått från föregående mipmapnivå skulle felet bli större och större för varje förminskning.

5.2.2 Metoden upsample

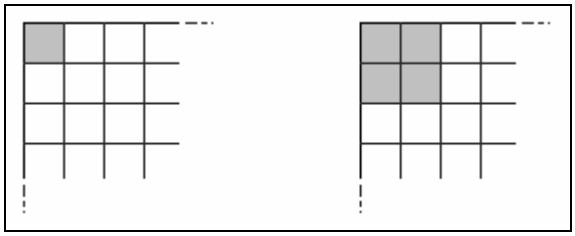
Denna metod förstorar en textur från en mipmapnivå till en annan. Metoden behövs för att kunna förstora alla mipmapnivåer av texturen till den ursprungliga storleken på texturen. Detta krävs för att kunna använda metoden computeError, som måste ha två lika stora texturer för att kunna utföra jämförelsen. Vi har skrivit två varianter av metoden, en med bilinjär filtrering och en helt utan filtrering.

Metoden utan filtrering drar bara ut texturen i x- och y-led och den uppförstorade texturen får ett kantigt utseende (se Figur 5.4).



Figur 5.4: Förstoring utan filtrering

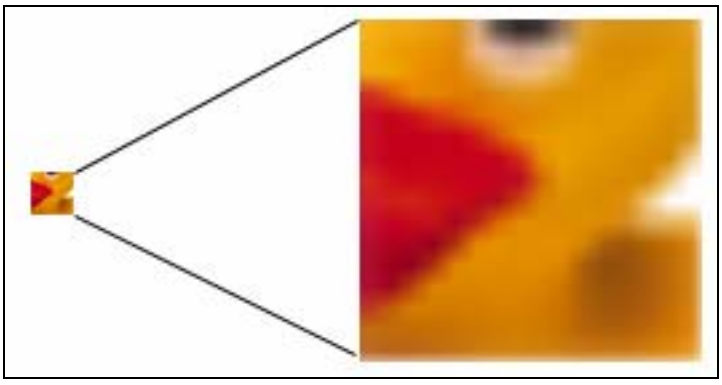
Om texturen exempelvis skall göras dubbelt så stor läses en pixel in från texturen och lagras fyra gånger i den uppförstorade texturen (Figur 5.5).



Figur 5.5: Uppförstoring av pixel utan filtrering

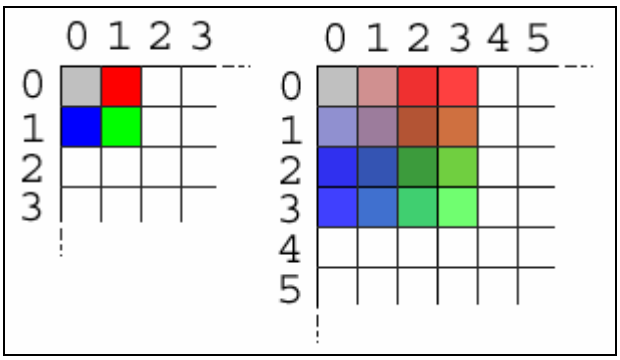
Detta upprepas för alla pixlar i hela texturen och på så sätt har texturen blivit dubbelt så bred och hög.

Metoden med bilinjär filtrering tar även hänsyn till angränsande pixlars färgvärden när den skall förstora upp en pixel. Därför fås en mjukare färgövergång mellan pixlarna än utan filtrering (Figur 5.6).



Figur 5.6: Förstoring med bilinjär filtrering

När en pixel skall förstoras med filtrering blir pixlarna i resultattexturen en blandning av pixeln som förstoras och dess grannar i texturen (Figur 5.7).



Figur 5.7: Uppförstorade pixlar med bilinjär filtrering

I Figur 5.7 kan vi exempelvis se att färgerna i pixlarna (1,0) och (2,0) i den högra texturen är en blandning av färgerna i pixlarna (0,0) och (1,0) i den vänstra texturen. När vi skriver pixel (1,0) menar vi pixel nummer ett i sidled och pixel nummer 0 i höjdled. Det är på detta sätt den jämna övergången mellan pixlarna framställs när man förstorar med filtrering.

5.2.3 Metoden computeError

När en textur förminskas eller förändras till något annat format försämras den för det mesta. För att få ett numeriskt värde på hur mycket den försämrats jämfört med originalbilden använder vi oss av en algoritm som jämför bilderna. Denna algoritm fungerar på så sätt att man ser varje pixel som en koordinat i ett 4-dimensionellt koordinatsystem. Där röd, grön, blå och alfa utgör varsin axel. För att sedan jämföra två texturer sätter vi in varje pixel i koordinatsystemet och räknar ut hur stor skillnad det är mellan pixeln för originalbilden och den försämrade bilden. Slutligen summerar vi alla fel och dividerar med antalet pixlar för att få fram ett medelvärde på hur stort felet är.

Denna metod att jämföra texturer är inte på något sätt optimal, men den är lätt att implementera och ger ett användbart värde. Detta värde är dock inte speciellt bra. Aspekter som måste tas hänsyn till om denna algoritm skall förbättras är bland annat hur det mänskliga ögat uppfattar saker, vilka färger vi ser skillnad på samt hur skarpa kontraster vi kan se och så vidare. Det går dock enkelt att byta ut denna metod mot en bättre om man vid ett senare tillfälle har möjlighet till det.

5.3 Klassen TextureManager

Denna klass har som vi beskrivit ovan som uppgift att sänka storleken på de texturer som är laddade i den, till den nivå där alla texturer får plats i det tillgängliga texturminnet. Hur detta principiellt går till beskrivs översiktligt i kapitel 4.2, texturhanterare. Det som beskrivs här är endast hur vi har implementerat detta.

Några funktioner som klassen innehåller förutom konstruktorer och destruktorer är metoder för att lägga till/ta bort texturer ur hanteraren och lägga till/ta bort format. Den innehåller också en metod som vi kallar för build. Denna metod är hjärtat i klassen skulle man kunna säga. Det finns även en funktion för att ange hur mycket ledigt minne man har tillgång till samt en funktion som vi kallar för flush som tömmer texturhanteraren på de inladdade texturerna.

Metoderna lägg till/ta bort texturer är inte så komplicerade. Deras uppgift är precis som det låter att lägga till en redan skapad textur av klassen Texture som beskrivs ovan. Det som är lite speciellt då man lägger till en textur är att den sorteras in i en lista där alla inladdade texturer finns. Denna lista är sorterad på det sätt att den textur som tappar minst i kvalitet av att man sänker storleken på den är först i listan. När man tar bort en textur ur texturhanteraren är det inga konstigheter alls, utan den plockas bara ut ur listan av texturer.

Man måste också ange för texturhanteraren vilka format som man vill skall vara med i urvalet då storleken skall sänkas. Att man skulle vilja välja bort något format beror främst på att alla grafikkort inte stöder alla format, så detta får man ange när man vet vilka format grafikkortet kan hantera. Vilka format som stöds anger man med metoderna lägg till/ta bort format.

Den metod som är viktigast i hela klassen är som vi sa tidigare build-funktionen. Denna fungerar på så sätt att den börjar med att räkna ut hur mycket minne som skulle behövas för att alla de texturer som man laddat in i texturhanteraren skulle kunna laddas med högsta kvalitet. Om det visar sig att det krävs mer minne för att ladda in texturerna än vad som är tillgängligt börjar förminskningen av texturer. Det går till på så sätt att den textur som ligger först i listan av texturer plockas ut. Därefter sänks kvaliteten på denna ett snäpp. Om storleken på texturen minskade då kvaliteten sänktes sorteras den in i listan igen. Därefter görs en ny beräkning om texturerna skulle få plats i minnet med den nuvarande kvalitetsuppsättningen. Om detta inte är fallet görs en ny förminskning av den textur som ligger först i listan. Detta upprepas sedan tills dess att alla texturer får plats i minnet. Om nu alla texturer inte får plats i

minnet trots att deras storlek har minskats till ett minimum returnerar metoden värdet false. Om texturen som ligger först i listan inte går att minska storleken på mer eftersom den redan står på det minsta tillgängliga texturformat i den lägsta mipmapnivån, läggs denna till sist i texturlistan och en flagga sätts i den som säger att den inte går att minska storleken på mer. När man har gjort beräkningen så att alla skall få plats i minnet är det dags att börja allokera texturminne åt alla texturer. Det är inte helt säkert att alla texturer får plats i minnet trots att de enligt tidigare beräkningar skall få det. Anledningen till detta är att det är näst intill omöjligt att beräkna hur mycket ledigt minne det finns att tillgå. På grund av detta måste man även testa om det går att allokera minnet till texturerna med den konfigurering som de har efter de första beräkningarna. Om det går bra att allokera minne till alla texturer flyttas de in i grafikminnet från datorns internminne, om det däremot visar sig att alla texturer inte får plats i texturminnet får man göra precis som man gjorde från början och börja sänka kvaliteten på den som ligger först i listan igen. Detta upprepas till dess att alla får plats i minnet eller alla har minskats till en nivå där det inte går att förminska dem mer.

Tilläggas kan att man endast kan ladda in en textur en gång i texturhanteraren. Detta har vi implementerat med hjälp av ett ternärt sökträd. Här läggs filnamnet in på varje textur då texturen läggs till i texturhanteraren. När man försöker lägga till en textur söks först trädet igenom efter filnamnet man försöker lägga till. Om filnamnet redan finns i trädet returneras endast en referens till den redan tillagda texturen. Om filnamnet inte finns läggs det in och ett nytt texturobjekt skapas.

5.4 Klassen NeuQuant

Som vi tidigare har förklarat lagras texturerna internt i Textureklassen i ett 32-bitars truecolor format som alltså tar upp fyra bytes per pixel. Ett format som är mindre minneskrävande är färgtabellsmappade texturer, där varianten med 256 färgvärden i tabellen är vanligast. Denna variant kräver endast en byte per pixel, eftersom en byte kan hålla värden från 0 till 255. Varje pixelvärde är då ett index in i färgtabellen och pixeln får det färgvärde som återfinns på aktuell position i färgtabellen. Det är denna konvertering mellan truecolor texturer och färgtabellsmappade texturer som hanteras av denna klass.

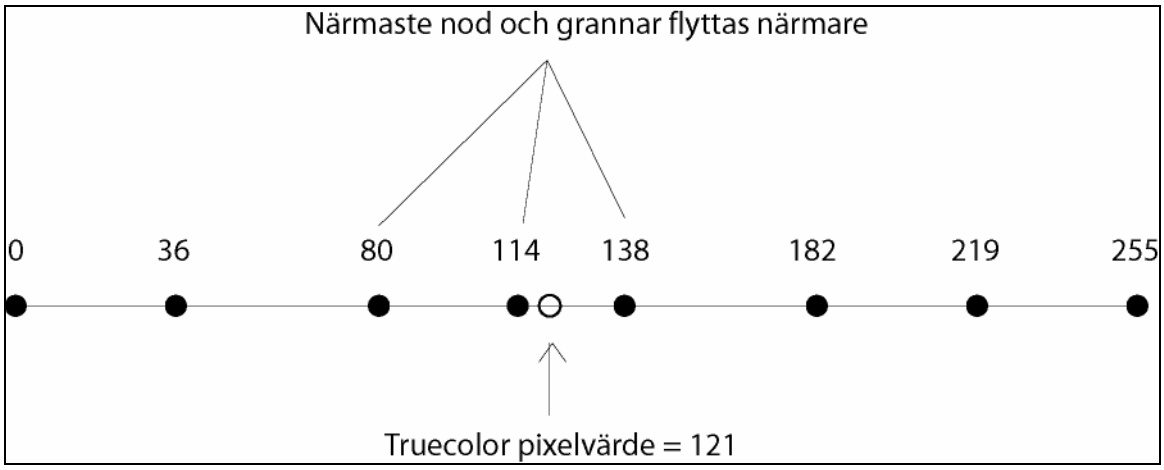
Det finns en mängd olika algoritmer för att göra denna konvertering eller färgkvantisering som det också kallas. Gemensamt för alla är att de från en bild med en stor mängd färger skall ta fram ett visst antal färger (vanligtvis 256) som man bäst kan representera den ursprungliga bilden med. De olika algoritmerna ger olika bra kvalitet på resultatet och är mer eller mindre

processorkrävande. En snabb algoritm som även ger ett bra resultat är den s.k. Neuquant-algoritmen [3]. Denna algoritm bygger på Kohonens nätverk av neuraler. Detta nätverk består av ett antal noder, *neuraler*, som är sammankopplade med varandra. I vårt fall finns 256 noder där varje nod innehåller ett färgvärde. Det är dessa färgvärden som sedan kommer att finnas i färgtabellen. För att lättare kunna förklara hur algoritmen fungerar kommer vi att visa ett nätverk med åtta noder där enbart bilder i en gråskalekanal kan kvantiseras.

Vi börjar med att initiera alla noders färgvärden jämnt över hela färgskalan (Figur 5.8).

Figur 5.8: Nätverkets utseende efter initiering

Sedan läser vi in pixel för pixel från truecolor bilden och hittar den nod vars färgvärde bäst stämmer överens med pixelns. Denna nods färgvärde ändras sedan för att bättre stämma överens med pixelns färgvärde. Även noderna i dess närhet ändras mot pixelvärdet (Figur 5.9).



Figur 5.9: Ändring av noder

Denna procedur upprepas tills alla pixlar har lästs in och noderna korrigerats. När detta är färdigt har vi i noderna de åtta färgvärden som bäst kan användas för att representera truecolor bilden. Dessa värden lagras i färgtabellen som vi sedan använder för att skapa den färgtabellsmappade bilden. För att skapa denna bild går vi ännu en gång igenom truecolor

bilden pixel för pixel och hittar det färgvärde i färgtabellen som bäst stämmer överens med pixelvärdet. Indexet till denna färg i tabellen lagras som det nya pixelvärdet.

Algoritmen vi beskrivit ovan fungerar liknande med fler färgkanaler som med bara en gråskalekanal, och i vårt fall använder vi tre färgkanaler och en alfakanal. Skillnaden blir att vi arbetar i fyra dimensioner istället för en dimension. Istället för att vi söker den närmaste noden i en dimension får vi söka den närmaste noden i fyra dimensioner och så vidare. Detta är mer tidskrävande men inte mycket svårare att implementera.

5.5 Klasserna för hantering av Dxt-block

Om intresse finns för hur dxt-komprimering går till rent teoretiskt hänvisar vi här till 2.3.2. I detta avsnitt tänker vi endast ta upp hur vi har löst de olika delarna rent implementationsmässigt och inte så mycket om tankarna bakom själva komprimeringen.

Vid dxt-komprimering av en textur delas som vi tidigare har sagt texturen upp i block om 16 pixlar. Klasserna Dxt1Block, Dxt3Block och Dxt5Block används för att komprimera just ett sådant block. Det vi vill ha ut från komprimeringen är för Dxt1Block endast två stycken 16 bitars färger. Då kan man ju undra varför vi inte tar reda på det bitmönster som senare används för att dekomprimera blocket. Svaret på denna fråga är att det skulle ta för mycket plats i vår texturfil om man skulle spara bitmönstret för 3 olika dxt-komprimeringar. Det är även så att det som är tidskrävande vid komprimering av dxt-block är att ta fram de två färger som bäst återskapar bilden. Att sedan ta fram bitmönstret går väldigt snabbt och är lätt gjort så detta kan man lika gärna göra i realtid om man vill använda dxt-formatet för en textur. Detta är naturligtvis en avvägning vi gjort med tanke på hur vår texturhanterare fungerar och hur vårt filformat ser ut.

Vid komprimering av dxt1 block går vi till väga på följande sätt. Vi går först igenom alla pixlar i blocket vi har fått in för att komprimera för att ta reda på om alfakanalen används eller inte, beroende på detta tas färgerna fram lite annorlunda. Samtidigt som vi går igenom om alfakanalen används eller inte kontrollerar vi också hur många olika färger det finns i blocket. Om det endast finns en eller två färger är det inte några problem att välja ut vilka färger som bäst återskapar blocket. Då det ju naturligtvis är de som finns i blocket. Om det däremot finns fler färger i blocket blir det något mer komplicerat. Då går vi tillväga så att vi skapar alla möjliga kombinationer av de olika färgerna som finns i blocket. Därefter testar vi helt enkelt alla kombinationer genom att anta att det är de färgerna som vi tänker använda och sedan

räknar ut hur stort felet blir om vi använder dessa färger. De färger vi till slut kommer att använda oss av är de som hade det minsta felet i vårt test.

Vid komprimeringen av dxt3 och dxt5 block går vi till väga på ungefär samma sätt. Skillnaden här är att vi inte gör någon kontroll på om alfakanalen används eller inte, för här tar man alltid med alfakanalen i komprimeringen oavsett om den används eller inte. För att ta fram färgerna för dessa dxt block gör vi på precis samma sätt som när vi tar fram färgerna för dxt1 blocket. Enda skillnaden är som sagt att man inte blandar in alfakanalen i färgkomprimeringen som man gör för dxt1, utan här komprimeras alfakanalen separat.

Den enda skillnaden för hur dxt3 och dxt5 komprimeras är hur man tar fram alfakanalen. Vi gör dock inte helt enligt specifikationen på dxt här heller utan väljer att inte komprimera någon alfakanal för dxt3 som vi sparar i vår fil utan denna kommer vi sedan ta fram under körning av vår texturhanterare, detta också för att spara på utrymmet i filen. Även för dxt5 väljer vi att spara på utrymmet i filen, så när vi komprimerar dxt5s alfakanal sparar vi endast två stycken 8-bitars alfavärden som bäst återskapar alfakanalen i originalbilden. Bitmönstret som används vid dekomprimeringen sparas inte här heller utan tas fram först då vi vill använda dxt5 formatet i vår texturhanterare.

Hur tar vi då fram de två olika alfavärden för att återskapa alfakanalen på bästa sätt? Jo, då gör vi precis som vi gjorde när vi tog fram färgerna vid komprimeringen av dxt, det vill säga testade alla möjliga alternativ och använde oss av det som gav minst fel.

6 Slutsatser och kommentarer

I detta kapitel tas upp en del kommentarer på hur i har genomfört vårt arbete. Här tas också upp hur saker hade kunnat göras annorlunda. Sist i kapitlet dras slutsatser av det arbete som har genomförts.

6.1 Erfarenheter och rekommendationer

När vi hade slutfört vårt arbete och tittade tillbaka på det märkte vi att vi hade lärt oss väldigt mycket. Detta samt hur vi skulle kunna ha gjort saker annorlunda kommer vi att ta upp och diskutera lite i detta kapitel.

Vi börjar med det positiva; alla de erfarenheter och kunskaper detta examensarbete har givit oss. Till att börja med var det väldigt nyttigt att få göra ett större projekt, till skillnad från de allra flesta laborationer vi har gjort under studietiden. Det är mycket mer stimulerande att veta att man gör någonting som kommer att kunna användas i något verkligt sammanhang än att göra något bara gör för att man ska göra det.

Själva arbetet gav oss mycket förbättrade kunskaper i framförallt programmering i c++. Vi har sysslat relativt mycket med detta även tidigare, men då på en något lägre nivå. Vi har även fått en del kunskaper i hur man planerar ett sådant här projekt. När det gäller just planeringen tycker vi att vi har lyckats bra. De allra flesta delarna av våra program har blivit precis som vi planerat dem, så vi har sluppit gå tillbaka efter en tid och göra några stora förändringar. Rent tidsmässigt har vi också lyckats bra. Vi visste redan från början att det kunde bli stressigt att implementera ett insticksprogram för Photoshop, och när vi började med detta och det krånglade ordentligt mycket var vi tyvärr tvungna att skjuta detta på framtiden. Det är dock det enda stora bakslag vi har stött på. Annars har arbetet flutit på i ett för oss högt tempo.

Över till det som vi kanske kunde ha gjort annorlunda och på så sätt bättre, men inte haft tid att fördjupa oss mer i.

Det som skulle kunna förbättra funktionen på framförallt vår texturhanterare är en bättre jämförelsealgoritm. Alltså den algoritm som bedömer hur stort felet är på en textur. Hur vi gör vår jämförelse nu kan du läsa om i kapitel 5.1, implementation av texturklassen. Vad vi istället kunde ha gjort är att använda oss av någon algoritm som även tog hänsyn till hur ögat uppfattar olika skillnader i texturerna. Om till exempel någon färgförändring uppfattas mindre

än någon annan av det mänskliga ögat. Sådana här saker har vi inte alls haft någon tid att sätta oss in i, men det skulle definitivt göra vår texturhanterare bättre. Algoritmen går dock enkelt att byta ut om man vill det.

En annan del av vårt system som skulle kunna förbättras är snabbheten. Då menar vi framförallt snabbheten när det gäller att skapa en textur. Eftersom fyra olika komprimeringar görs då en textur skapas tar detta i de flesta fall ganska lång tid. Tiden det tar att skapa en textur skulle förmodligen kunna förminskas om man implementerar DXT-komprimeringen på ett annat sätt. Men det i sin tur skulle förmodligen försämra kvaliteten på den komprimerade texturen. Meningen med hela vårt system är ju just att alla beräkningar och komprimeringar skall göras just när texturen skapas och inte då texturen skall användas. Detta därför att det skall vara väldigt snabbt ta fram rätt textur i runtime när ett spel körs. Eftersom det är först när man har startat spelet man kan veta exakt hur mycket tillgängligt grafikminne man har så är det naturligtvis inte förrän här man vet vilken storlek man har råd att ha på varje textur. Den tid det tar att skapa en textur av vårt texturformat varierar väldigt mycket beroende på storleken på texturen samt hur många kanaler texturen innehåller. Exempelvis kan sägas att det tar cirka 1 minut att skapa en textur i storleken 512 x 512 pixlar på en 700Mhz processor.

6.2 Slutsatser

Arbetet som helhet har gått väldigt bra. Vi har inte stött på några stora problem utan det allra mesta har fungerat näst intill helt felfritt. Programvaran som vi hade behov av för att genomföra arbetet har vi hela tiden haft tillgång till så det har inte varit något problem. Det enda lilla irritationsmomentet var att Visual C++ vid ett tillfälle inte ville kompilera vårt projekt. Detta berodde på att vi saknade det servicepack som krävdes. De dokument vi har varit i behov av har vi också haft tillgång till hela tiden. Internet har varit till stor hjälp om det varit någon information som har saknats. Där har vi tagit hem mycket av det material vi har behövt för att ta fram bakgrundsfakta och även vissa av de algoritmer som vi har implementerat. Även företaget vi har jobbat åt, Digital Illusions, har varit väl tillgängliga och har gett oss den hjälp vi har behövt när vi behövt den.

Vi är väldigt nöjda med den produkt vi har tagit fram. Hela vårt system fungerar så långt vi har kunnat testa det på ett bra sätt. Det skall dock tilläggas att vi inte har kunna testa det i den miljö där det slutgiltigt kommer att användas. Tanken är att det skall användas i ett datorspel, men eftersom spelet inte är färdigt ännu har vi som sagt inte kunnat testa vår produkt.

Vi har arbetat på lite olika sätt under olika faser av arbetet. I början när det var mycket planering satt vi hela tiden tillsammans och diskuterade hur vi ville att vårt system skulle se ut. När vi sedan började implementera det vi hade planerat delade vi upp arbetet och var och en satt på sin egen arbetsplats och arbetade relativt självständigt. I slutet när alla olika delar i projektet skulle sammanfogas arbetade vi återigen tillsammans. Dokumentationen har vi arbetat med under nästan hela arbetet, men bara någon timma i veckan. Detta har vi gjort ganska mycket tillsammans även om var och en har skrivit om det den var mest insatt i. Den arbetsfördelning och det arbetssätt som vi har jobbat med tycker vi har fungerat väldigt bra. Arbetet har gått snabbare framåt när vi har suttit var och en för sig, men samtidigt har det blivit mer genomtänkt när vi har arbetat tillsammans. Så båda arbetssätten visade sig ha fördelar som vi behövde i de olika faserna av projektet.

Referenser

[1] Truevision, *Targa File Format Spec*, ftp://ftp.truevision.com/pub/TGA.File.Format.Spec/PC.Version/TGA_FSS.EXE, 2002-05-15

[2] Microsoft, *DirectX 8.1 SDK*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dx8_c/directx_cpp/dx8start_c_81.asp, 2002-05-15

[3] Anthony Dekker, *Neuquant: Neural Image Quantization*, http://members.ozemail.com.au/~dekker/NEUQUANT.HTML, 2002-05-15

[4] Computer Games Online, *DXTn Texture Compression*, http://www.cdmag.com/Home/home.html?article=/articles/021/068/dxt_feature.html, 2002-05-15