godk.datum vitsord

bedömare

Triangelbaserad och voxelbaserad rendering inom realtidsgrafik

Peter Hedman

Helsingfors 17 december 2011

HELSINGFORS UNIVERSITET
Institutionen för datavetenskap

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matematisk-naturvetenskapliga fakulteten	Institutionen för datavetenskap

Tekijä — Författare — Author
Peter Hedman

Työn nimi — Arbetets titel — Title
Triangelbaserad och voxelbaserad rendering inom realtidsgrafik

Oppiaine — Läroämne — Subject
Datavetenskap

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	17 december 2011	22 sidor

Tiivistelmä — Referat — Abstract

Den ständigt ökande hårdvarukapaciteten för dagens datorer har möjliggjort unika och obegränsade detaljer också inom realtidsgrafik. Unika detaljer är helt oberoende av varandra och följer inga återupprepningsmönster. Med obegränsade detaljer menas att objekten inte har en övre gräns på deras detaljrikedom eller upplösning. Spelföretaget id Softwares nyaste titel, Rage, är ett exempel på hur långt man kan komma med nuvarande renderingsmetoder. I Rage är hela världen unikt och obegränsat färglagd. Uppsatsen analyserar möjligheterna att ta ett steg vidare och också erbjuda unika och obegränsade geometriska detaljer. Något som är möjligt, men komplicerat, med den etablerade triangelbaserade renderingsmetoden. I det etablerade sättet att rendera tredimensionell geometri sparas geometrin som listor av trianglar. För att rendera geometrin till skärmen projiceras trianglarna till skärmplanet. Triangelbaserad rendering kan erbjuda unika och obegränsade geometriska detaljer med hjälp av tessellering och höjdkartor.

*Voxel*baserad (eng. *voxel*, en tredimensionell pixel) datorgrafik har funnits vid sidan om den traditionella triangelbaserade renderingen som ett elegant, men utrymmeskrävande alternativ. För att kringgå utrymmeskravet förespråkar John Carmack, id Softwares chefsutveckare, och hans kollega John Olick strålkastning av voxlar med hjälp av accelerationsstrukturer och kan således erbjuda både unika och obegränsade detaljer. I uppsatsen presenteras och jämförs båda metoderna. Det framkommer det att voxelbaserad grafik kan erbjuda imponerande grafiska resultat med hög detaljnivå, animerade objekt och indirekt belysning. Trots det kommer ändå högupplöst realtidsvoxelgrafik att vara för hårdvarukrävande för grafikteknologin i åtminstone sex år till. Carmack förutspår ändå att en del av nästa generations videospel kommer att experimentera med voxelgrafik som grafikteknologi, även om merparten kommer att använda triangelbaserad grafik.

ACM Computing Classifcation System (CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism–Color, shading, shadowing and texture–Raytracing;

Avainsanat — Nyckelord — Keywords
Voxel, datorgrafik, rendering, oktalträd, BSP-träd,strålkastning, tessellering

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Innehåll

1 Introduktion

Den ständigt ökande hårdvarukapaciteten för dagens datorer har möjliggjort *unika detaljer* och *obegränsade detaljer* också inom *realtidsgrafik*. Med realtidsgrafik menas datorgrafik som inte är genererad på förhand och har tillräckligt hög uppdateringsfrekvens för att användaren skall se en jämn animation istället för enskilda bilder. Med unika detaljer avses färgmässiga och geometriska detaljer som inte följer återupprepningsmönster och är helt oberoende av varandra. Obegränsade detaljer har ingen övre gräns på sin detaljnivå eller upplösning. Med hjälp av unika och obegränsade detaljer lyfts begränsningarna på konstnärerna som skapar och färglägger geometrin och detaljrikedomen begränsas endast av deras förmågor [Oli08, s. 13].

Främst i användningen av unika och obegränsade detaljer står spelföretaget id Softwares[1] nya titel Rage, där hela världen i spelet är unikt färglagd. Laine och Karras [LK10] gör antagandet att trenden mot unika och obegränsade detaljer kommer att bli starkare med geometrin som nästa utvecklingsfas. För att effektivt rendera geometri med obegränsade detaljer måste man försäkra sig om att geometrins detaljnivå anpassas utgående från dess storlek på skärmen [LK10, kap. 1]. Ett problem som uppstår med unika detaljer är att utrymmeskravet ökar markant jämfört med tidigare mönsterbaserade detaljer. För att hantera det ökade utrymmeskravet måste man kunna både spara och ladda in detaljerna effektivt [LK10, kap. 1].

Den etablerade triangelbaserade renderingsmetodens huvudsakliga fördel är att den effektivt kan rendera geometri som lätt kan representeras som en samling plan [LK10, kap. 1]. Den här fördelen minskar dock om de geometriska detaljerna är unika och obegränsade. Det ger en orsak att undersöka alternativa renderingsmetoder som lämpar sig bättre för unika geometriska och färgmässiga detaljer.

I det etablerade sättet att rendera tredimensionell geometri sparas geometrin som listor av trianglar. För att rendera geometrin till skärmen projiceras först trianglarnas tredimensionella koordinater till skärmplanet. Sedan bestäms färgvärdet för varje pixel på skärmen utgående från den närmaste triangeln som täcker pixeln.

Trots sin relativt enkla struktur medför triangelbaserad rendering problem som kräver komplicerade lösningar. Man måste effektivt hitta den närmaste triangeln som täcker en pixel. Ett annat problem som uppstår är *pixelering* (eng. *aliasing)* av trianglarna [AMHH08, kap. 5.6]. Pixelering är den trappstegsliknande effekt som

[1] Id Software har bland annat producerat spelen Wolfenstein 3D (1992), Doom (1993) och Quake (1996).

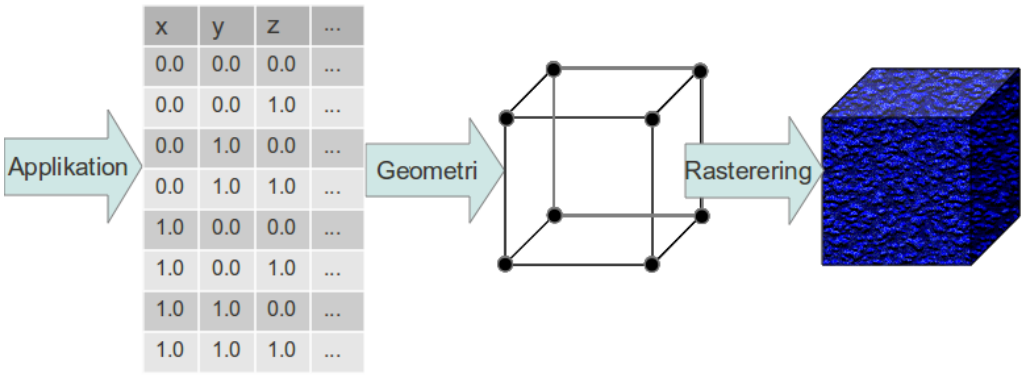
uppkommer när objekten representeras som pixlar. För att undvika pixelering måste man utföra arbetskrävande *kantutjämning* (eng. *antialiasing*). Kantutjämning är ett täcknamn för olika metoder som blandar ut färgvärdena för de pixlar som utgör kanter i geometrin. För att anpassa objektens detaljnivå byts de vanligtvis ut mot stegvis mera förenklade versioner utgående från avståndet till betraktaren [AMHH08, kap. 14.7]. Dessutom är det svårt att effektivt parallellisera den triangelbaserade renderingen [Oli08, s. 19].

John Carmack, id Softwares chefsutvecklare, förespråkar *strålkastning* (eng. *ray casting*) av *voxlar* (eng. *voxels*, tredimensionella pixlar) med hjälp accelerationsstrukturer för att erbjuda unika och obegränsade detaljer [Shr08]. Strålkastning är en renderingsmetod där man följer strålar från varje pixel på skärmen in i scenen. Det första objektet i geometrin som strålen träffar bestämmer pixelns färg. Detaljproblemet löses delvis av strålkastning. Mängden arbete beror på hur komplicerade rutter strålarna följer, inte på scenens komplexitet som vid triangelbaserad rendering [AMHH08, s. 415].

Voxelbaserad datorgrafik har funnits vid sidan om den traditionella triangelbaserade renderingen som ett elegant, men utrymmeskrävande alternativ. Voxelgrafiken har fått fotfäste inom medicinska tredimensionella scanningar och inom filmindustrin [CNLE09, kap. 1], men har haft begränsad genomslagskraft inom realtidsgrafik. Under de senaste sju åren har dock flera forskningsresultat publicerats där voxlar används inom realtidsgrafik. Gobetti et al. [GM05] använder voxlar som approximationer av geometri på långt avstånd. Crassin et al. [CNLE09] samt Laine och Karras [LK10] representerar hela geometrin med hjälp av voxlar och kan således erbjuda unika och obegränsade färgmässiga och geometriska detaljer. Med hjälp av strålkastning kan voxelgrafiken dra bättre nytta av den ständigt ökande parallellismen inom hårdvaran för grafikacceleration än den triangelbaserade renderingen [Oli08, s. 6-38].

Uppsatsen beskriver både den etablerade triangelbaserade renderingen och voxelbaserad grafik. Både för- och nackdelar med den voxelbaserade renderingen jämförs med den triangelbaserade renderingsmetoden. Speciell tonvikt läggs på strålkastning i voxeloktalträd som Carmack och hans kollega Jon Olick föreslår som framtidens renderingsteknologi [Shr08, Oli08].

I det andra kapitlet beskrivs hur den etablerade triangelbaserade renderingen fungerar. För att erbjuda en rättvis jämförelsegrund för de voxelbaserade renderingsmetoderna beskrivs i slutet av kapitlet hur tessellering kan användas för att uppnå



Figur 1: Schematisk överblick av *pipeline*-strukturen för triangelbaserad rendering.

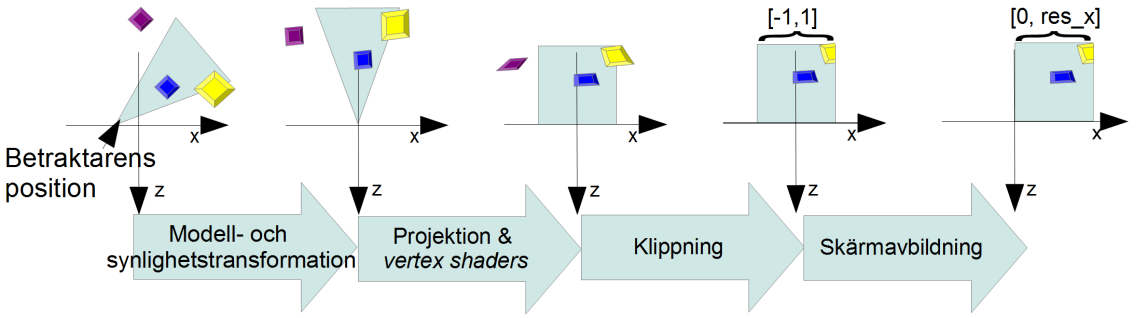
obegränsade geometriska detaljer med triangelbaserad rendering. För att voxelbaserade renderingsmetoder ska kunna beskrivas introduceras och förklaras i det tredje kapitlet olika datastrukturer för rymduppdelning. I kapitel fyra beskrivs olika renderingsmetoder som använder voxlar med tonvikt på strålkastning av voxlar inom oktalträdsstrukturer. Det femte kapitlet är en jämförelse av triangelbaserad rendering och voxelbaserad rendering där båda metodernas svagheter och styrkor analyseras. I sammanfattningen framhävs den triangelbaserade renderingen som den hittills starkare renderingsmetoden trots att voxelrenderingen erbjuder intressanta fördelar.

2 Triangelbaserad rendering

2.1 *Pipeline*-strukturen

Pipeline-strukturer (eng. *pipeline structure*) används överallt inom datatekniken för att snabba upp sekventiella processer. För att skapa en *pipeline*-struktur delar man upp den sekventiella processen i självständiga delar. Eftersom delarna är oberoende av varandra kan man behandla data på samma sätt som ett löpande band, så att varje del hela tiden får nya data att arbeta med. Med en n-stegs *pipeline*-struktur kan man öka dataflödet (eng. *throughput*), det vill säga medeltalsprestandan, med en faktor på n. *Pipeline*-strukturer förekommer ofta inom både processor- [Sta09, kap. 12.4] och grafikprocessorarkitekturer [AMHH08, s. 29].

Pipeline-strukturen för datorgrafik kan i grova drag delas upp i tre distinkta skeden: applikationsskedet, geometriskedet och rastereringsskedet, se figur 1. Alla dessa skeden kan ha egna *pipeline*-strukturer för att ytterligare snabba upp renderingsprocessen.



Figur 2: Överblick av en vanlig *pipeline*-struktur för geometriskedet, baserad på [AMHH08, kap. 2.3]. I figuren antas y-axeln vara riktad ut från pappret. Skärmbufferten motsvaras av rektangeln $R = \{(x, y, z) \mid 0 \leq x \leq res\_x, 0 \leq y \leq res\_y, z = 0\}$, skärmbuffertens horisontella upplösning betecknas res_x och dess vertikala upplösning beteknas res_y.

2.2 Applikation

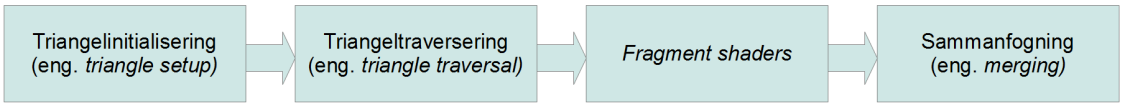
I applikationsskedet genereras *grafikprimitiven* (eng. *graphics primitive*) som resten av *pipeline*-strukturen sedan renderar till *skärmbufferten* (eng. *frame buffer*). Skärmbufferten är en tvådimensionell tabell som innehåller färgvärden som motsvarar de pixlar som syns på skärmen. Med grafikprimitiv menas grundläggande geometriska former som bygger upp geometrin i datorgrafiken. Processerna som utförs här varierar från en applikation till en annan. Exempelvis kan ett videospel utföra spellogiks- och fysikberäkningar i detta skede.

2.3 Geometri

Det är geometriskedets uppgift att projicera grafikprimitiven till skärmbufferten. Akenine-Möller et al. [AMHH08, kap. 2.3] beskriver en indelning av geometriskedet i följande skeden: Modell- och synlighetstransformation, projektion och *vertex shaders*, klippning och skärmavbildning, se figur 2.

En transformation är en avbildning som arbetar på koordinater och utför en förflyttning, förminskning, förstorning, skjuvning och/eller en rotation. Se [AMHH08, kap. 4] för ingående beskrivningar av olika transformationer som används inom datorgrafiken. Modell- och synlighetstransformationerna anpassar koordinatsystemet så att betraktaren befinner sig i origo och tittar längs med den negativa z-axeln[2].

[2]Förutsatt att koordinatsystemet är högerhänt. Transformationerna i detta skede varierar från en implementation till en annan. Metoden som beskrivs här stämmer överrens med den



Figur 3: Överblick av en vanlig *pipeline*-struktur för rastereringsskedet, baserad på [AMHH08, kap. 2.4].

I projektionsskedet och *vertex shader*-skedet anpassas koordinatsystemet ytterligare för att beakta perspektivet. *Vertex shaders* är program som körs på grafikkortet för att modifiera grafikprimitivens positioner. *Vertex shader*-programmen ger också programmeraren möjlighet att spara och modifiera andra data, till exempel data för belysning och *normaler* (eng. *normal*). En normal är en enhetsvektor som representerar en triangels ytnormal. Normaler används inom beräkningar för ljussättning.

Under klippningsskedet filtrerar man bort geometri som inte kommer att synas på skärmen. Geometri som är bara delvis synlig klipps i delar och de nya delarna behandlas på nytt.

I skärmavbildningsskedet anpassas koordinatsystemet så att x- och y-axlarna motsvarar skärmbuffertens resolution.

2.4 Rasterering

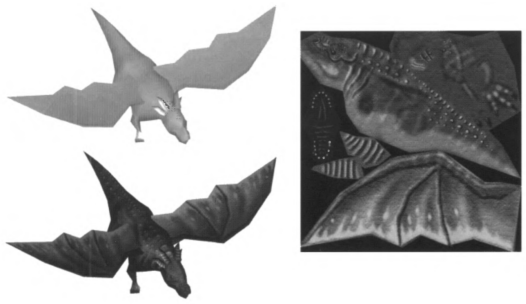
Rastereringsskedet kombinerar den färdigbehandlade geometrin från geometriskedet med de data som sparades av *vertex shader*-programmen för att bestämma ett färgvärde för varje pixel i skärmbufferten. Akenine-Möller et al. [AMHH08, kap. 2.4] delar in rastereringsskedet i fyra delskeden: Triangelinitialisering, triangeltraversering, *fragment shaders* och sammanfogning, se figur 3.

Under initialiseringsskedet kalkyleras derivator för trianglarnas ytor och annan data som behövs under de senare skedena.

Traverseringsskedet besöker varje triangel och räknar ut koordinaterna för varje pixel som triangeln överlappar i skärmbufferten. Här interpoleras också de data som *vertex shader*-programmen har sparat för trianglarna.

Akenine-Möller et al. beskriver [AMHH08, kap. 2.3.1], men till exempel använder sig DirectX-biblioteket sig av ett vänsterhänt koordinatsystem och betraktaren kommer således att titta längs med den positiva z-axeln [AMHH08, s. 96]

Fragment shaders är program som körs på grafikkortet för att bestämma färgvärdet för en pixel. Under *fragment shader*-skedet körs ett *fragment shader*-program för varje pixelposition som traverseringsskedet genererar. Programmen använder också de interpolerade data som traverseringsskedet generar för att bestämma färgvärdet för pixlarna. Under detta skede används ibland *texturer* (eng. *texture*) för att kunna ge unika färgvärden åt pixlarna, se figur 4. En textur är en en-, två- eller tredimensionell tabell som innehåller färgvärden eller annan data. Vanligtvis utgörs texturer av tvådimensionella bilder.



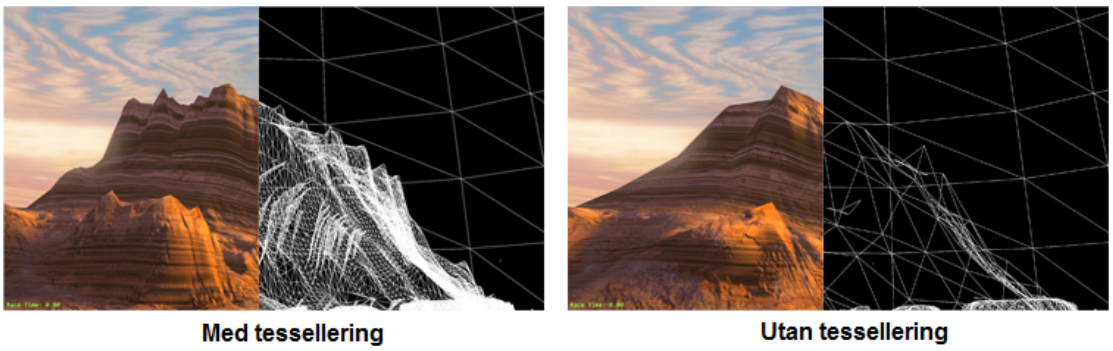
Figur 4: Exempel på texturering under *fragment shader*-skedet [AMHH08, kap. 2.4].

Sammanfogningsskedet tar hand om situationer när flera färgvärden täcker samma pixel. Om färgerna inte är genomskinliga väljs det värde som är närmast betraktaren på z-axeln. Å andra sidan, om en del färgvärden är genomskinliga genereras en medeltalsfärg utgående från de färgvärden som är synliga.

2.5 Tessellering

Tessellering (eng. *tessellation*) är en process som delar in geometriska ytor i nät av geometriska primitiv. Tessellering används aktivt för rendering av datoranimerade filmer: den är en central del *Reyes*-arkitekturen (*Renders Everything You Ever Saw*) som bland annat används i filmstudion Pixars RenderMan-program. I Reyes-arkitekturen tesselleras geometrin in i *mikropolygoner* (eng. *micropolygons*), det vill säga fyrhörningar vilkas storlek på skärmen är mindre än en pixel [CCC87, kap. 2.3]. Detta är inte fallet för tessellering inom triangelbaserad rendering. Geometrin tesselleras in i trianglar och deras storlek på skärmen behöver nödvändigtvis inte vara mindre än en pixel.

Rendering av högdetaljerade objekt är med triangelbaserad rendering lika tidskrävande oberoende av avståndet till betraktaren. I kapitel 5.1 beskrivs olika metoder för att minska den arbetsmängd som krävs för objekt som befinner sig långt borta. På senare tid har hårdvarustödet för tessellering introducerat ett intressant lösningsalternativ till den här problematiken.



Figur 5: Exempel på tessellering med höjdkartor. Bilden är hämtad i november 2011 från NVIDIAs hemsida:
http://www.nvidia.com/object/GTX_400_architecture.html.

Med den nya lösningsmetoden drar man nytta av ett nytt skede i *pipeline*-strukturen för moderna grafikkort, tesselleringsskedet [AMHH08, s. 630]. Tesselleringskedet är en del av geometriskedet och utförs före projektions- och *vertex shader*-skedet[3].

Under tesselleringsskedet kan trianglarna delas upp beroende på deras avstånd till betraktaren. *Vertex shader*-programmen ansvarar sedan för att de nya trianglarnas hörnpunkter motsvarar geometrin de representerar.

I metoden används tessellering i samband med *höjdkartor* (eng. *height maps*) för att dynamiskt anpassa detaljnivån på objekten som renderas, se figur 5 [AMHH08, kap. 13.5.6]. En höjdkarta är en tvådimensionell tabell vars värden representerar olika höjder och tabellens indexvärden används för att beskriva det horisontella läget för höjdvärdena.

Objekten får under tesselleringsskedet olika uppdelningsgrader utgående från avståndet till betraktaren. Sedan förflyttar *vertex shader*-programmen de nya hörnpunkterna längs med den ursprungliga triangelns ytnormal utgående från värdena i höjdkartorna.

[3] *Pipeline*-strukturen som beskrivs här är förenklad och stämmer överrens med strukturen som Akenine-Möller et al. beskriver [AMHH08, s. 630]. *Pipeline*-strukturen för tessellering som används i dagens grafikkort lägger till tre skeden efter *vertex shader*-skedet, se [MK10, kap. 1] för mera detaljer.

3 Datastrukturer för rymduppdelning

För många tillämpningsområden av tredimensionella data är naiva representationer av datan inte tillräckligt effektiva. Objekten kan tänkas sparas i tredimensionella tabeller som direkt motsvarar scenen. Utrymmeskravet för sådana tabeller är $O(N^3)$, där N är resolutionen för en dimension i tabellen, och kan därför snabbt bli för stort för centralminnet. Om objekten i stället sparas tillsammans med sina positioner i listor minskar utrymmeskravet. Tyvärr kan inte listorna bibehålla objektens relativa positioner och det enda sättet att hitta vilka objekt som befinner sig i en viss region är att gå igenom hela listan.

Många processer inom datorgrafiken kan snabbas upp med hjälp av datastrukturer för rymduppdelning. Strukturerna organiserar geometrin i en scen på olika sätt. Akenine-Möller et al. [AMHH08, kap. 14.1] påpekar att användandet av en rymduppdelningsstruktur i många fall kan göra linjära processer till logaritmiska.

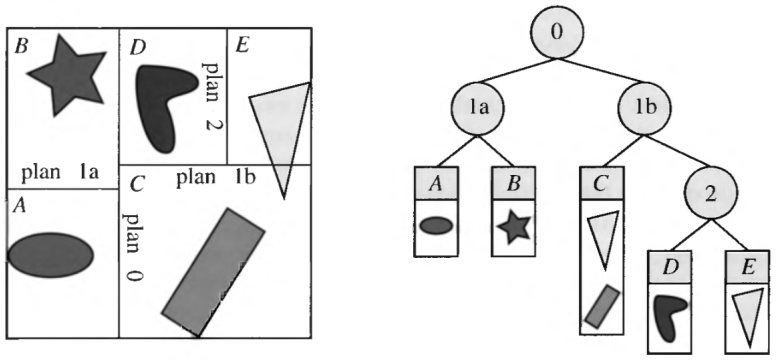
Vanligtvis är strukturerna hierarkiska, det vill säga indelade i nivåer så att de övre nivåerna helt och hållet täcker in de undre nivåerna. Den hierarkiska indelningsprocessen fortsätter tills något kriterium uppfylls. Sådana kriterier kan till exempel vara att höjden på strukturen överskrider ett förutbestämt maximum eller att mängden objekt i varje del av strukturen är mindre än ett förutbestämt minimum.

Rymduppdelningsstrukturer kan användas direkt inom renderingsalgoritmer för att ta reda på i vilken ordning objekten i scenen bör renderas [4] [AMHH08, s. 653–654]. De kan också användas för att gallra bort objekt som inte kommer att synas i den slutgiltiga bilden [AMHH08, s. 666]. Dessa strukturer används bland annat också inom fysiksimuleringar för att snabbt ta reda på vilka objekt som kolliderar med varandra [AMHH08, kap. 17.2].

3.1 BSP-träd

BSP-träd (eng. *binary space partitioning trees*) använder plan för att dela upp geometrin i två delar. Uppdelningen fortsätter rekursivt tills ett bestämt kriterum uppfylls. Inom datorgrafiken används ofta två distinkta versioner av BSP-träd: *Axelorienterade BSP-träd* och *polygonorienterade BSP-träd*. Versionerna skiljer sig från

[4] Till exempel kan man öka prestandan med dagens grafikkort genom att först rendera den geometri som är närmast betraktaren. Dagens grafikkort utför nämligen inte *fragment shader*-skedet för de delar av geometrin som täcks av tidigare uppritad geometri [AMHH08, kap. 15.4.5].



Figur 6: Exempel på indelning med hjälp av axelorienterade BSP-träd [AMHH08, kap. 14.1.2].

varandra utgående från hur de väljer delningsplanen.

Polygonorienterade (eng. *polygon aligned*) BSP-träd användes av tidiga tredimensionella spel[5] för att se till att de främre objekten döljer objekten längre bort [AMHH08, kap. 14.1.2]. De används mera sällan inom dagens datorgrafik och kommer därför inte att behandlas närmare i uppsatsen.

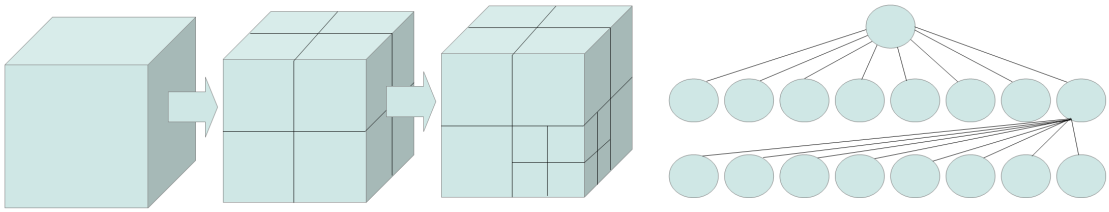
Axelorienterade (eng. *axis-aligned*) BSP-träd är en begränsad form av BSP-träd. I axelorienterade BSP-träd spjälks geometrin endast upp längs med plan som är ortogonala till en av koordinataxlarna, se figur 6. Denna begränsning gör att de går snabbt att skapa och klarar därför till en viss mån av att hantera dynamisk geometri. Om indelningsplanen följer ett förutbestämt mönster kallas träden för *k-d träd* (k-dimensionella träd) [AMHH08, kap. 14.1.2].

3.2 Oktalträd

Oktalträd (eng. *octree*) använder sig av lådor istället för spjälkningsplan för att dela upp geometrin, se figur 7. Rotnoden i oktalträd är en låda som täcker in hela scenen. Strukturen konstrueras genom att rekursivt spjälka upp geometrin i $2*2*2 = 8$ lika stora underlådor tills ett visst kriterum uppfylls. Namnet oktalträd kommer från uppdelningen i åtta delar [AMHH08, kap. 14.1.3].

Axelorienterade BSP-träd kan dela upp geometrin på liknande sätt om alla delningsplanen som väljs skär mittpunkten av lådan. Oktalträd är dock enklare att konstruera än BSP-träd och lämpar sig därför bra för dynamiska scener som ofta förändras.

[5]Spelen Doom (1993), Quake (1996) m.fl.



Figur 7: Överblick av ett oktalträd och dess minnesrepresentation.

Brönniman och Glisse [BG04] visar att oktalträd är den teoretiskt optimala strukturen för strålföljning om man kan garantera att geometrin inte är överlappande och är jämnt fördelad. Eftersom oktalträden inte är lika flexibla som till exempel axelorienterade BSP-träd används enligt Knoll et al. [KWPH06, kap. 2] vanligtvis k-d träd för det allmänna fallet av strålföljning.

4 Voxlar

4.1 Definition och egenskaper

Ordet voxel är en förkortning av uttrycket *volumetrisk pixel*. En voxel är ett grafikprimitiv och definieras som en kub som är förknippad med data. Inom datorgrafiken sparar man vanligtvis färgvärden i samband med voxlar, men det är också vanligt att spara övriga data vid sidan om färgvärdet, exempelvis normaler och genomskinlighetsgrad.

Med en naiv implementation sparas voxlar i tredimensionella tabeller där indexvärdena för en voxel motsvarar direkt dess tredimensionella position. Eftersom implementationen är så enkel används voxlar ofta som geometriuppskattningar för att förenkla uträkningar också utanför datorgrafiken, t.ex. inom fysiksimulationer för att approximera hur vätskor interagerar med kroppar [CLT07].

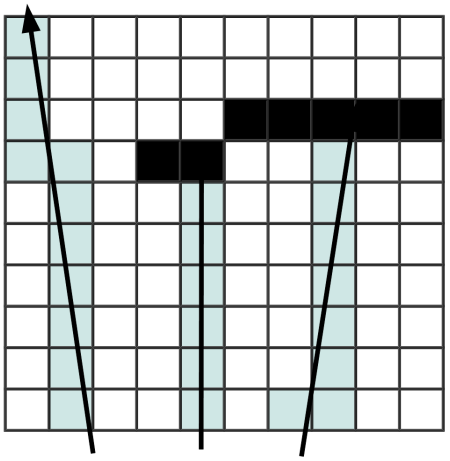
Inom medicinen används rendering av voxeldata för att visualisera diagonstisk data från tredimensionella skanningar, till exempel från datortomografier eller magnetiska resonanstomografier (MRT) [AMHH08, s. 504].

Voxlar används även inom specialeffekter i filmer, vanligtvis för att rendera dynamiska effekter som rök, vatten och skum. Crassin et al. [CNLE09, kap. 1] påpekar att voxlar också används för högdetaljerade statiska objekt.

4.2 Strålkastning av voxeldata

Akenine-Möller et al. [AMHH08, kap. 10.16] beskriver ett sätt att rendera voxeldata genom att använda sig av strålkastning.

Strålkastningsprocessen är betydligt enklare än traditionell triangelrendering. För varje pixel i skärmbufferten följer man en stråle in i voxeltabellen. Den första icke-genomskinliga voxeln som strålen träffar bestämmer färgvärdet för pixeln, se figur 8.



Figur 8: Tvådimensionellt exempel på strålkastning i en voxeltabell.

Stråkastning medför den fördelen att tidskravet för renderingsprocessen är oberoende av detaljrikedomen i geometrin. Å andra sidan är strålkastningen ytterst beroende av scenens storlek. Tidskravet för en strålkastning är $O(N)$, där N är upplösningen på en axel i voxeltabellen. Utrymmeskravet är $O(N^3)$ och överskrider snabbt storleken på grafikminnet för dagens grafikkort[6].

4.3 Voxelterräng

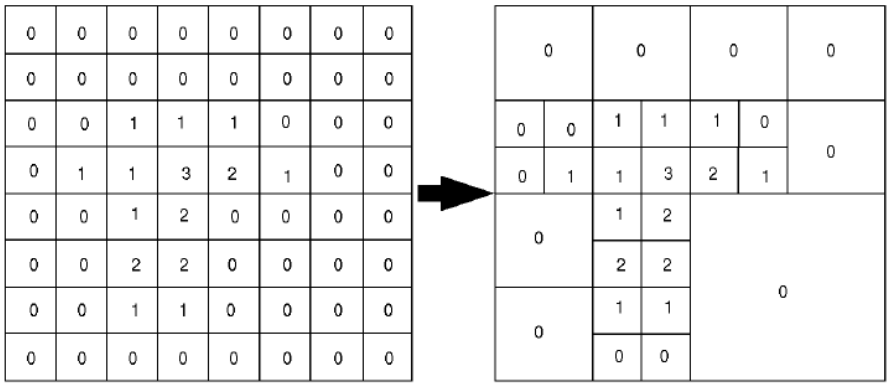
Under 1990-talet användes voxlar inom dataspelsbranschen för att representera terräng[7], se figur 9. LaMothe [LaM95, kap. 16] beskriver en metod där terrängen sparas som tvådimensionella höjdkartor och renderas genom att utföra en strålkastning per pixel på skärmen.

Figur 9: Exempel på voxelterräng i spelet Comanche: Maximum Overkill. Bilden är hämtad i november 2011 från Wikipedia: http://en.wikipedia.org/wiki/Comanche_series

I metoden som LaMothe beskriver optimeras strålkastningsprocessen genom att pixlarna behandlas kolumnvis. LaMothe utför strålkastningen mot planet som höjdkartas lägsta nivå definierar. För varje pixel ritar han upp den projicerade höjden för voxeln som befinner sig i skär-

[6]En relativt lågupplöst scen bestående av 2048^3 voxlar med 32 bitar färgdata per voxel kräver som okomprimerad 32GB utrymme.

[7]Spelen Comanche: Maximum Overkill (1992), Delta Force (1998) m.fl.

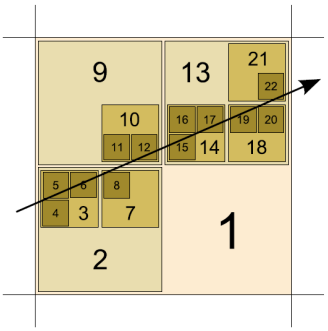


Figur 10: Tvådimensionellt exempel på komprimering av voxeldata med hjälp av oktalträd [KWPH06, kap. 3].

ningspunkten med planet. Terrängen längst bak ritas upp först för att terrängen längre fram skall förbli synlig. Metoden går att optimera ytterligare. Exempelvis kan man utnyttja att de övre pixlarna i kolumen befinner sig högre upp och kan således omöjligt kollidera med den terräng som de undre pixlarna undvikit.

4.4 Voxlar i oktalträd

I kapitel 3.2 konstateras det att oktalträd är den optimala för strålföljning om man kan garantera att geometrin inte är överlappande. Oktalträdstrukturen lämpar sig därför bra som rymduppdelningsstruktur för att effektivera strålföljning i voxeldata, eftersom voxlar är bundna till ett tredimensionellt rutnät och därför inte kan vara överlappande.



Figur 11: Tvådimensionellt exempel på strålkastning i ett oktalträd bestående av voxlar [LK10, s. 13].

Knoll et al. [KWPH06, kap. 3.1] beskriver hur man kan komprimera tredimensionella voxeltabeller med hjälp av oktalträd. Genom att bygga oktalträdsstrukturen nerifrån upp kan regioner med samma färgvärde, så kallade *homogena* regioner, sparas som färre noder högre upp i oktalträdet, se figur 10.

Genom att låta oktalträdet fungera både som rymduppdelningsstruktur och som direkt representation av geometrin drar Knoll et al. [KWPH06, kap. 3.2] nytta av båda fördelarna med oktalträd. Förutom utrymmesbesparingen som oktalträden erbjuder

kan strålkastningssprocessen effektiveras genom att bara besöka de högre nivåerna i oktalträdet för homogena regioner (exempelvis tomrum). Nivåer djupare ner i trädet besöks bara om de verkligen innehåller varierande geometri som måste beaktas, se figur 11.

Crassin et al. [CNLE09] samt Laine och Karras [LK10] utnyttjar ytterligare denna oktalträdsrepresentation genom att låta innernoderna spara approximeringar av sina barn. På så sätt kan detaljnivån enkelt väljas utgående från avståndet till betraktaren. Om noderna befinner sig på ett tillräckligt långt avstånd från betraktaren besöker de helt enkelt inte nivåer djupare ner i trädet, utan nöjer sig med de approximationsvärden som är sparade i de övre nivåerna.

I sin presentation beskriver Olick [Oli08, s. 6-38] hur detaljnivån i framtidens videospel kommer att öka med hänsyn till både unika geometriska och unika färgmässiga detaljer. Han påpekar att sådana effekter är möjliga med triangelbaserad rendering, men kräver två skilda komplicerade lösningar. Enligt Olick kan man med strålkastning in i voxeloktalträd erbjuda båda effekterna (både färgmässig och geometrisk detaljrikedom) med en och samma metod.

5 Jämförelse

5.1 Dynamisk detaljnivå

Inom datorgrafiken kan man uppnå bättre prestanda genom att dynamiskt sänka detaljnivån på objekt som befinner sig på långt avstånd från betraktaren. Akenine-Möller et al. [AMHH08, kap. 14.7] beskriver olika metoder för att anpassa detaljnivån för triangelbaserad rendering.

Metoderna de presenterar har dock begränsningar. De beskriver system där objekten blir stegvis mera genomskinliga ju längre bort de befinner sig från åskådaren. Tyvärr effektiverar inte halvgenomskinliga objekt renderingen förrän de har försvunnit helt. De beskriver också metoder där man helt enkelt byter ut modellen mot mindre detaljerade versioner, vilket leder till att märkbara ryck uppstår när objektets modell byts ut. Till sist presenterar de en algoritm som stegvis sammanslår hörnpunkter för polygonerna som modellen består av, men de hörnpunkter som skall slås ihop måste väljas för hand för att undvika att modellens utseende ändras radikalt.

I kapitel 2.5 beskrevs hur tessellering med höjdkartor kan användas för att dynamiskt

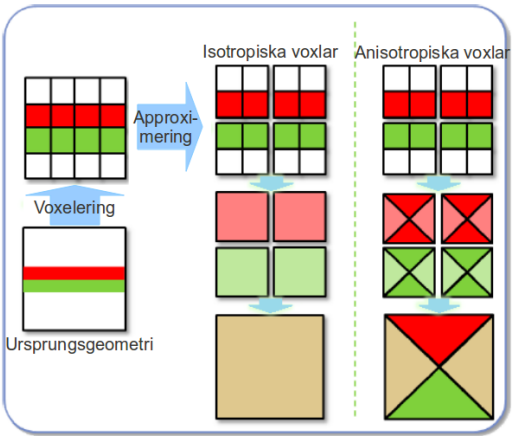
anpassa detaljnivån för triangelbaserade objekt. Med hjälp av tessellering kan man alltså undvika alla de ovannämnda bristerna.

Voxlarnas regelbundna struktur gör dem till utmärkta verktyg för approximering av geometri på långt avstånd.

Gobbetti et al. [GM05] använder sig av triangelbaserad rendering med voxlar som approximationer av geometrin på långt avstånd. I metoden de presenterar sparas geometrin i ett axelorienterat BSP-träd där alla löv är grupper av trianglar. Innernoderna i trädet representeras av voxlar som erbjuder tillräckligt bra approximationer av geometrin de representerar. Voxlarnas färgvärden är beroende av vinkeln de betraktas från. I deras renderingsalgoritm ritar Gobbetti et al. ut voxlarna i stället för geometrin de representerar i de fall där de inte täcker mera än en pixel i skärmbufferten.

Crassin et al. [CNLE09] samt Laine och Karras [LK10] använder samma sorts heuristik i sina lösningar. Eftersom de sparar hela geometrin som voxeloktalträd tillämpar de en liknande metod för all geometri: För varje pixel i skärmbufferten kastar de strålar in i oktalträdet och besöker bara de nivåer i oktalträdet vilkas voxlar nätt och jämnt blir mindre än en pixel om de projiceras till skärmbufferten.

Den här metoden möjliggörs av att låta innernoderna i oktalträdet innehålla approximationer, vanligtvis ett medelvärde, av datan i sina barn. Alltså innehåller rotnoden en approximation av hela geometrin. Om approximationerna räknas ut på ett naivt sätt så medför de ett allvarligt problem: Om man skall generera en approximation för en vägg vars båda sidor har olika färger kommer den att innehålla en blandning av väggens båda färger. Detta är ett problem som Gobetti et al. inte stöter på med sin metod. Crassin et al. [CNS+11] löser det här problemet för geometri som är parallell med koordinataxlarna genom att introducera anisotropiska voxlar, det vill säga voxlar som har olika färgvärden för varje yta. I approximationerna blandas bara de ytor ihop som är riktade åt samma håll, se figur 12.



Figur 12: Exempel på oönskad färgblandning vid approximering av voxeldata och hur det löses med anisotropiska voxlar. Bilden är baserad på [CNS+11, kap. 9].

5.2 Bildkvalitet

I kapitel 4.4 konstateras det att voxlar i oktalträd tillåter både komplicerad geometri och färgläggning. Laine och Karras [LK10, kap. 1] beskriver hur id Softwares spelmotor, id Tech 5, använder en enda stor textur för att erbjuda unik färgläggning för triangelbaserad rendering. De skriver att unika geometriska detaljer skulle vara möjliga med hjälp av tessellering (se kapitel 2.5), om också höjdkartorna skulle sparas i en liknande stor textur.

Crassin et al. [CNLE09] sparar de sista nivåerna i voxeloktalträdet som tredimensionella texturer. Därför kan de effektivt utföra kantutjämning genom texturfiltreringsfunktionaliteten i dagens grafikkort för att blanda ihop datan av olika approximationsnivåer för voxlarna. Kantutjämningsmetoderna som Akenine-Möller et al. [AMHH08, kap. 5.6.2] beskriver för triangelbaserad rendering är mera tidskrävande. Metoderna baserar sig på att rendera flera färgvärden per pixel och spara deras medelvärde i skärmbufferten. Till skillnad från de triangelbaserade kantutjämningsmetoderna så suddar metoden som Crassin et al. beskriver dock ut alla kanter i geometrin. Detta medför att även fullständigt vertikala och horisontella kanter suddas ut, trots att de kan representeras exakt i skärmbufferten.

Laine och Karras [LK10, kap. 5.2] använder sig av konturer för att öka detaljnivån av voxelrepresentationen. Varje voxel kan förknippas med konturer, två parallella plan, som begränsar voxelns geometriska utsträckning. Med hjälp av konturer kan Laine och Karras också exakt representera ytor som inte står vinkelrätt mot koordinataxlarna i voxeloktalträd.

På nära håll lider voxlar av samma pixeleringsproblem som digitala fotografier. Med tillräckligt stor förstoringsgrad ser de kantiga ut. Triangelbaserad rendering lider av samma problem endast om de är texturerade och således är beroende av bilddata. Pixeleringen av texturer undviks genom att låta texturfiltreringsfunktionaliteten i grafikkorten tillsätta *oskärpa* (eng. *blur*) till texturerna under rastereringsskedet. Laine och Karras [LK10, kap. 6.3] beskriver en metod för att uppnå liknande effekter med voxelrendering genom att tillsätta oskärpa vid voxlarnas kanter i efterskott.

5.3 Dynamiska scener

Dynamiska scener och animerade objekt kan enkelt representeras med triangelbaserad grafik. Trianglarna deformeras genom att att anpassa transformationer på deras hörnpunkter, vilket enligt Laine och Karras [LK10, kap. 3.4] anses vara de facto me-

toden för triangelanimation idag. Eftersom trianglarna sparas i listor behöver inte datastrukturen förändras när trianglarnas relativa positioner gör det.

Om voxlarna sparas i något av de ovannämnda formaten (tredimensionella tabeller eller oktalträd) leder samma animationsmetod till problem eftersom datastrukturen också måste förändras varje gång en voxel ändrar position.

Laine och Karras [LK10, kap. 3.4] föreslår olika metoder för att hantera dynamisk geometri i voxeloktalträd.

De påpekar att en metod som liknar komprimeringsmetoderna hos dagens digitala videoformat kunde användas för animering av voxelgrafik. Nackdelen med en sådan metod är att den stödjer sig på förhandsinspelade ögonblicksbilder för animering, vilket omöjliggör växelverkan med resten av geometrin.

De beskriver också ett sätt att spara de dynamiska objekten i *vattentäta* (eng. *watertight*) skal av trianglar, där triangelskalen kan animeras med konventionell triangelanimering. Ett skal anses vara vattentätt om varje sammanhängande objekt som inte skär skalet antingen är fullständigt innanför skalet eller fullständigt utanför skalet [ED08, kap. 5.1]. De strålar som träffar skalet förflyttas till ett annat voxeloktalträd där den dynamiska modellen står stilla. Modellen animeras genom att strålarna i det nya voxeloktalträdet transformeras enligt triangelskalet och således träffar olika voxlar beroende på triangelskalets läge. Triangelskalen verkar vara ett lovande, men invecklat lösningsalternativ och saknar ytterligare forskningsresultat.

Crassin et al. [CNS+11, kap. 4.2] använder i sin artikel realtidsvoxelering av triangelgrafik för att representera dynamiska objekt. Voxeleringsprocessen använder *pipeline*-strukturen för grafik för att effektivt konstruera en voxelering av triangelbaserade modeller. Crassin et al. stänger av *djuptestet* (eng. *depth test*) och renderar modellen från alla tre axlar för att garantera att ett *fragment shader*-program startas för varje potentiell voxel. Djuptestet är den del av sammanfogningsskedet som garanterar att bara de trianglar som är längst fram ritas till skärmbufferten. Crassin et al. utnyttjar triangelmodellernas voxelrepresentationer både för rendering av modellerna och för att förenkla uträkningarna för scenens ljussättning.

Laine och Karras [LK10, kap. 3.5] påpekar att i renderingssyften lider voxelering av triangelgrafik av både triangelrenderingens och voxelrenderingens svagheter. För att representera dynamisk geometri måste modellen renderas två gånger: Först måste den voxeleras och sedan måste voxelrerpesentationen renderas till skärmbufferten. Till skillnad från de voxelbaserade renderingsmetoder som beskrivits ovan beror

voxeleringens arbetsmängd på detaljnivån av triangelmodellen. Dessutom kommer den voxelerade representationen att vara mindre detaljerad än den ursprungliga triangelmodellen.

John Carmack [Shr08] föreslår en hybridmodell som använder voxeloktalträd för statisk geometri och triangelbaserad rendering för dynamiska objekt.

5.4 Utrymmeskrav

I kapitel 4.2 konstaterades det att tredimensionella voxeltabeller snabbt kan bli ohanterligt stora. Voxeloktalträd fungerar som kompressionsmetod för voxeldata, se avsnitt 4.4. Knoll et al. [KWPH06, kap. 5.1] rapporterar att voxeloktalträd kan vara upp till fem gånger mindre än den ursprungliga voxeltabellen de representerar.

Laine och Karras [LK10, kap. 4] presenterar en utrymmesanalys där de konstaterar att *ytbaserade data* i voxeloktalträd har ett utrymmeksrav på $O(N^2)$, där N är resolutionen för en dimension i voxeltabellen[8]. Att geometrin är ytbaserad innebär att den kan representeras som tunna ytor eller skal: Volymerna i geometrin fylls alltså inte ut med data. De visar också att voxeloktalträd bara är en konstant faktor mera utrymmeskrävande än trianglar, ifall båda representerar unika detaljer [LK10, kap. 3.8]. Detta följer av att man för varje triangel måste spara både dess hörnpunkter och två texturer: En för färgläggning och en höjdkarta för de geometriska detaljerna. Trots detta kräver voxeloktalträd ändå för mycket utrymme för att en hel scen skulle kunna sparas i grafikkortsminnet, speciellt om betraktaren förflyttar sig mellan renderingarna.

För att minska utrymmeskravet för sina voxeloktalträd komprimerar Laine och Karras [LK10, kap. 5.5] både voxlarnas färgvärden och ytnormalerna med *destruktiva komprimeringsmetoder* (eng. *lossy compression*), det vill säga kompressionsmetoder som minskar den ursprungliga datans precision.

Gobbetti et al. [GM05, kap. 5], Crassin et al. [CNLE09, kap. 6] samt Laine och Karras [LK10, kap. 7] utnyttjar dataströmmande (eng. *data streaming*) för att kunna hantera större scener med begränsade minnestillgångar. Dataströmningen underlättas av att scenerna kan renderas utan att alla data finns på plats, eftersom nivåerna högre upp i träden innehåller grövre uppskattningar av geometrin vilka kan användas i stället för den detaljnivå som behövs.

[8] I sin analys behandlar Laine och Karras kubiska scener, det vill säga att resolutionen längs med alla axlar är samma [LK10, kap. 4].

Laine och Karras [LK10, kap. 7] strömmar in data utgående från avståndet till betraktaren. Med hjälp av avståndet till betraktaren kan de uppskatta storleken i pixlar för en voxel i oktalträdets löv. I deras lösning strävar de efter hålla lövnodernas storlek konstant i skärmbufferten.

Crassin et al. [CNLE09, kap. 6.1] låter grafikminnet agera som ett cache-minne med LRU-heuristik (*last recently used*), det vill säga de data som inte har använts på längst tid skrivs över. De voxlar som senast besökts under strålkastningen skrivs alltså inte över när nya data måste laddas in för strålkastningen. I deras lösning bestämmer alltså själva renderingsprocessen vilka data som bör finnas i grafikminnet.

Gobbetti et al. [GM05, kap. 5] använder två skilda cachenivåer: Centralminnet och grafikminnet. De upprätthåller dessa cacheminnen med en liknande sorts LRU-heuristik som Crassin et al. I grafikminnet finns de data som nyligen har renderats. Centralminnet innehåller de data som nyligen har besökts under renderingen. Om data saknas och grövre geometriuppskattningar måste användas under renderingsprocessen laddas nya data stegvis in i centralminnet. Under inladdningsprocessen laddas de data in först som tar upp störst area i skärmbufferten.

5.5 Renderingsprestanda

Den triangelbaserade renderingsmetoden är mycket effektiv. Den hårdvaruaccelererade *pipelinen* för datorgrafik innebär att realtidsuppdateringsfrekvenser lätt går att uppnå. Spelet Rage är ett exempel på hur unik texturering är möjlig med triangelbaserad rendering i realtidsgrafik. Många nya spel[9] erbjuder obegränsad detaljnivå med hjälp av tessellering. Det saknas dock ännu en produkt som kombinerar unika och obegränsade detaljer, något som skulle ge en rättvis jämförelsegrund med oktalträden.

Strålkastning i voxeloktalträd är helt tydligt inte lika effektivt som triangelrendering. Knoll et al. [KWPH06, kap. 5.3–5.4] kör sin algoritm på centralprocessorn. De klarar därför inte av att uppnå tillräckligt höga uppdateringsfrekvenser för realtidsgrafik utan att sänka upplösningen för både voxeldatan och skärmbufferten.

Crassin et al. [CNLE09, kap. 7] renderar också till en lågupplöst skärmbuffert. Eftersom de har byggt sin algoritm kring *pipelinen* för datorgrafik och utnyttjar *fragment shaders* för själva strålkastningen klarar de av att rendera mera högupplösta scener än Knoll et al.

[9]Spelen Crysis 2, Battlefield 3, Metro 2033 med flera.

(a) Unigine Corporations prestandatest som använder sig av tessellering.

(b) Indirekt ljussättning av en scen med mycket hög detaljnivå med hjälp av voxeloktalträd [CNS+11].

Figur 13: Exempel på obegränsade detaljer med hjälp av tessellering och voxeloktalträd.

Laine och Karras [LK10, kap. 9.1] använder GPU-programmeringsspråket CUDA för att få full kontroll över grafikprocessorn. De uppnår 25 bilder per sekund i en högupplöst skärmbuffert (med upplösningen $1920 * 1080$). Renderingen saknar dock avancerad ljus- och skuggsättning och kantutjämning. Med liknande kantutjämning som för den triangelbaserade renderingen (se avsnitt 5.2) och avancerad skuggsättning genom ytterligare strålkastning från geometrin till ljuskällorna minskar dock prestandan drastiskt. De uppskattar att det kommer att ta uppskattningsvis sex år innan man ser högupplöst realtidsgrafik som baserar sig på strålkastning in i voxeloktalträd.

Enligt Olick [Oli08, s. 6-38] kommer dock den triangelbaserade renderingens prestandaövertag att minska med tiden. Han påpekar att trenden för hårdvaruaccelererad grafik går mot högre parallellism. Något som kommer att leda till problem för triangelbaserad rendering, eftersom initaliseringsskedet av rastereringen (se kapitel 2.4) inte går att parallellisera i en lika hög grad som resten av *pipeline*-strukturen. Strålkastning kan dra betydligt större nytta av parallell hårdvara eftersom uträkningarna för varje stråle är helt oberoende av varandra.

6 Sammanfattning

I den här uppsatsen presenterades först den konventionella triangelbaserade renderingen varefter strålkastning in i voxeloktalträd presenterades som en alternativ renderingsmetod.

Som det konstaterades i kapitel 4 är arbetsmängden för strålkastning oberoende av geometrins detaljnivå. I kapitel 5.5 beskrivs det hur strålkastningen erbjuder en mycket högre grad av parallellism än triangelbaserad rendering. I kapitel 5 beskrivs också hur voxeloktalträd lättare kan hantera dynamiska detaljnivåer jämfört med triangelbaserad rendering.

I kapitel 5.3 framkom även de nackdelar som voxelbaserad strålkastning medför. Dynamiska scener är svåra att representera med hjälp av voxlar. Den mest lovande lösningen verkar vara att använda triangelbaserad rendering för att rendera de dynamiska objekten efter att den statiska geometrin har renderats med strålkastning.

Trots att analysen i kapitel 5.4 visar att voxeloktalträdens utrymmeskrav är betydligt mindre än förväntat krävs fortsättningsvis olika dataströmningslösningar för att med dagens minneskapacitet rendera scenarion där betraktaren rör på sig.

Med hjälp av triangelbaserad rendering har man skilt för sig uppnått både obegränsad detaljnivå och unika färgmässiga detaljer, se kapitel 5.5. Båda effekterna finns ändå inte i samma produkt ännu. Unigine Corporations prestandatest Heaven är ett bra exempel på hur man åstadkommer obegränsade geometriska detaljer med tessellering, se figur 13a.

Crassin et al. [CNS+11] demonstrerar i deras artikel imponerande grafiska resultat med både animerade objekt och interaktiv indirekt belysning med hjälp av voxeloktalträd, se figur 13b. Indirekt belysning innebär att ytorna inte endast belyses av ljuskällorna i scenen, utan också beaktar reflekterat ljus. Att beräkna indirekt belysning är en tidskrävande process. Därför beräknas belysningen inom realtidsgrafiken vanligtvis på förhand [AMHH08, s. 327–333]. Detta försvårar möjligheterna att interagera med den indirekta belysningen.

Som det konstateras i kapitel 5.5 kommer högupplöst voxelgrafik att vara för hårdvarukrävande för grafikteknologin i åtminstone sex år till [LK10, kap. 9.1]. Carmack [Shr11] förutspår ändå att en del av nästa generations videospel kommer att experimentera med voxeloktalträd som grafikteknologi, även om merparten kommer att använda tesselleringsbaserad grafik.

Referenser

AMHH08 Akenine-Möller, T., Haines, E. och Hoffman, N., *Real-Time Rendering.* Tredje utgåvan. A. K. Peters, Ltd., Natick, MA, USA, 2008.

BG04 Brönnimann, H. och Glisse, M., Cost-optimal trees for ray shooting. *Proceedings of the Latin American Symposium on Theoretical Informatics*. Springer, 2004, sidorna 349–358.

CCC87 Cook, R. L., Carpenter, L. och Catmull, E., The Reyes image rendering architecture. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA, 1987, ACM, sidorna 95–102.

CLT07 Crane, K., Llamas, I. och Tariq, S., Real-time simulation and rendering of 3D fluids. *GPU Gems 3*, New York, NY, USA, 2007, Addison-Wesley Professional, sidorna 633–675.

CNLE09 Crassin, C., Neyret, F., Lefebvre, S. och Eisemann, E., Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, New York, NY, USA, 2009, ACM, sidorna 15–22.

CNS+11 Crassin, C., Neyret, F., Sainz, M., Green, S. och Eisemann, E., Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, 30,7(2011).

ED08 Eisemann, E. och Décoret, X., Single-pass GPU solid voxelization for real-time applications. *Proceedings of Graphics Interface 2008*, GI '08, Toronto, Ont., Kanada, 2008, Canadian Information Processing Society, sidorna 73–80.

GM05 Gobbetti, E. och Marton, F., Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, New York, NY, USA, 2005, ACM, sidorna 878–885.

KWPH06 Knoll, A., Wald, I., Parker, S. G. och Hansen, C. D., Interactive isosurface ray tracing of large octree volumes. *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006, sidorna 115–124.

LaM95 LaMothe, A., *Black Art of 3D Game Programming*. Waite Group Press, Corte Madera, Kalifornien, USA, 1995.

LK10 Laine, S. och Karras, T., Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, februari 2010.

MK10 McDonald, J. och Kilgard, M., Crack-Free Point-Normal Triangles Using Adjacent Edge Normals. NVIDIA Whitepaper WP-05621-001_v01, NVIDIA Corporation, december 2010.

Oli08 Olick, J., Next generation parallelism in games. *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008, ACM, sidorna 21:1–21:89.

Shr08 Shrout, R., John Carmack on id Tech 6, ray tracing, consoles, physics and more. *PC Perspective*. http://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more [Hämtad i oktober 2011].

Shr11 Shrout, R., John Carmack interview: GPU race, Intel graphics, ray tracing, voxels and more. *PC Perspective*. http://www.pcper.com/reviews/Editorial/John-Carmack-Interview-GPU-Race-Intel-Graphics-Ray-Tracing-Voxels-and-more [Hämtad i oktober 2011].

Sta09 Stallings, W., *Computer Organization and Architecture: Designing for Performance*. Åttonde utgåvan. Pearson Prentice Hall, Upper Saddle River, New Jersey, USA, 2009.