Raytracing i Realtid

Miika Heikura Matr. 34201
Kandidatavhandling i Informationsteknologi
Handledare: Ville Timonen
Institutionen för Informationsteknologi
Åbo Akademi
2013

Innehållsförteckning

1. Referat

Ray tracing simulerar ljuspartiklars, eller -strålars, flöde genom den virtuella rymden, vilket leder till mycket mera realistiska bilder än med tekniker som bara använder geometrin. Metoden är dock mycket långsammare p.g.a. den extra komplexiteten som följer då man simulerar ljusets kollisioner med virtuella objekt. Fördelen med att använda ray tracing är att många av ljusets egenskaper är väldigt enkla att simulera med denna metod, medan de är betydligt mera komplicerade att implementera med geometri-baserade renderingsmetoder. En annan fördel är att ray tracing är mera effektivt i väldigt stora geometrier, eftersom man kan indela geometrin, och på så sätt få en logaritmisk komplexitet.
I detta arbete ägnar jag diskutera och jämföra metoder som används för att få renderingsmetoden ray tracing att fungera i realtid, och samla resultat om hur snabbt man fått den att fungera hittills.
Ray tracing har dock undersökts mycket mindre jämfört med rasteriseringstekniker, som nu används i de flesta grafikkort, så det är ännu svårt att veta hur metoden kommer att utvecklas. Dessutom växer den kapacitet med vilken vi kan utföra beräkningar hela tiden, så det är möjligt att ray tracing i realtid en dag blir mera än bara en dröm.

2. Inledning

Rendering betyder att skapa en grafisk bild av en tredimensionell modell, utgående från en virtuell kameras position. Ray tracing är en renderingsmetod som simulerar ljusets strålar för att generera fotorealistiska bilder. Ray tracing simulerar geometriskt ljusstrålars färd i 3D-rymden, men den behöver inte nödvändigtvis simulera ljusets vågegenskaper. Vissa implementationer tar ljusets våglängd i beaktande, vilket möjliggör simulation av t.ex. effekter som dispersion i ett prisma.
Metoden härstammar från Arthur Appels ray casting metod [1], som sköt ljusstrålar från skärmen in i 3D-modellen, så att strålarna stannade genast då de träffade något objekt. Metoden utvidgades av Turner Whitted [2] så att strålarna fick studsa vidare vid kontakt med objekt. Då en stråle träffade en yta kunde den bilda tre olika slags strålar, refraktions-, reflektions- och skuggstrålar [3].
Ordet ’trace’ betyder på Engelska att spåra. I praktiken följer man ljusets strålar från kameran, till objekten i modellen, som ljuset kan studsa ifrån, absorberas av eller färdas i, och till ljuskällor i modellen. På grund av sin beräkningsintensitet är metoden avsevärt långsammare än rasterisering, som nu används i alla grafikkort.
Ett problem med rasterisering är att global belysning är svårt att implementera. Med global

belysning menas flera av ljusets egenskaper som får en bild att se mera realistisk ut, med bidrag utöver dem som kommer från direkt belysning. T.ex. blir delar av ett rum som inte direkt nås av en ljuskälla ändå inte helt och hållet mörka, eftersom ljuset studsar av från andra ytor och når ytor genom indirekt belysning. När rasterisering används simuleras global belysning genom t.ex. ambient ocklusion, vilket innebär att områden i modellen som har mycket geometri runt sig renderas som mörkare men andra områden lämnas ljusa. Global belysning innefattar även reflektioner och kaustik, som bidrar till ljus från strålar som färdats inuti ett medium. Dessa är mycket svåra att simulera med rasterisering. Ray tracing har alltså fördelen att m.h.a. den kan man skapa bilder med mycket högre kvalitet, och det är lätt att implementera flera av ljusets naturliga egenskaper. Ray tracing är som metod även väldigt enkel och lätt att förstå, och leder ofta till väldigt enkel och koncis kod. Wald et al. har skrivit om global belysning och metoder för att implementera det med rasterisering eller ray tracing [4].

Dessa egenskaper gör ray tracing mycket attraktivt för vem som helst som producerar datorgrafik. Ray tracing används ofta i filmindustrin, och det kan ta från timmar till flera dagar att rendera en enda bild. Detta beskriver väl problemet att använda metoden i realtid.

Implementationer av rendering indelas som off-line, interaktiv eller realtid beroende på deras snabbhet. Dessa beskrivs i boken Real-Time Rendering av Akenine-Möller et al [5]. Implementationer i realtid innebär att en ny bild kan renderas 15 gånger i sekunden eller snabbare. En fart på 6 bilder per sekund ger redan känslan av realtid, men vid 72 bilder per sekund är de enskilda bilderna helt oskiljbara, och användaren kan uppleva att det som sker är helt och hållet i realtid. Om det tar en sekund eller mera att rita en bild på skärmen, lider användarens känsla av interaktivitet, men detta är ändå tillräckligt snabbt då man t.ex. vill se hur en 3D-modell ser ut med en viss belysning. Då renderingen tar för lång tid för att användaren skall kunna agera med den talas det om off-line rendering. Då kan det ta från minuter till dagar att rendera en enda bildruta.

Tredimensionella modeller består av primitiv. Dessa kan vara t.ex. trianglar eller polygoner. Mera om 3D-modellering finns att läsa t.ex. i Alan Watt's bok 3D Computer Graphics [6].

Den naiva lösningen på problemet är att skapa en stråle från varje pixel i skärmen och testa dess skärning med varje primitiv i scenen. Detta leder dock till en komplexitet på O(p x s), där p står för antal primitiv och s för antal strålar. Detta leder dessutom till onödiga beräkningar, eftersom en stråle inte kommer att skära de flesta primitiv i scenen. Benjamin Mora påpekar dock att en naiv lösning kan delas upp i mindre, fortfarande naiva lösningar, och på så sätt minska komplexiteten [7]. Denna metod och dess resultat tas upp i avsnitt 3.2.2.

För att snabba upp skärningstesten måste man ha ett effektivt sätt att dela upp geometrin enligt läge i rymden eller enligt objekt. Då man har delat upp rymden på något sätt, kan

man testa skärningen mellan strålar och större helheter än primitiv, och på så sätt kan man dramatiskt minska antalet skärningstest.
Ett sätt är att bygga ett kd-träd som beskriver geometrin. Kd-träd (k-dimensionella träd) är binära träd som delar rymden enligt en av koordinataxlarna med varje förgrening [8]. I 3D-grafiska användningar har de dimensionen 3. De är väldigt lätta att bestiga, men de har en lång konstruktionstid. Kd-träd lämpar sig bra för statiska scenografier, eftersom man bara behöver bygga trädet en gång, men lämpar sig mindre bra för dynamiska scenografier, där trädet skulle behöva återbyggas för varje bildruta. Användningen av kd-träd har dock försnabbats till en interaktiv nivå av t.ex. Shevtsov et al [9].
En annan metod är användningen av Bounding Volume Hierarchies (BVH). Dessa delar in rymden hierarkiskt i volymer, så att de innesluter ett antal objekt. Sedan kan en volym delas i flera mindre volymer och så vidare, och på detta sätt kan man välja större eller mindre grupperingar av objekt. Sedan kan skärning testas med strålar, och om en stråle inte korsar volymen, så kommer den inte heller att korsa något av objekten inne i volymen. Mera om BVH kan läsas av Rubin et al [10]. BVH är även lämpliga för dynamisk geometri, eftersom de kan uppdateras, istället för att bilda dem från början för varje ny bild. Det är dock långsammare än interaktivt att bygga upp trädet för moderat stora geometrier [7]. Resultat gällande BVH diskuteras i avsnitt 3.2.1.
En till accelerationsstruktur som kan användas är gitter. Gitter är snabbare att bygga upp, och har en lägre komplexitet, så de har föreslagits som ett alternativ [11] [12]. Jag kommer inte att fokusera på gitterimplementationer i detta arbete.
En renderingsmetod där strålar även modelleras är path tracing. Path tracing uppfanns av Kajiya, och beskrivs i samma arbete som renderingsekvationen [13]. Då path tracing används, integreras allt ljus som kommer in till varje punkt på varje yta i bilden. I praktiken används dock olika metoder för att försnabba processen, eftersom det är beräkningsintensivt att beräkna det inkommande ljuset från varje riktning från varje punkt. Det är möjligt att skicka strålar i slumpmässiga riktningar, och sedan ta genomsnittet av dem för att bestämma hur en punkt på en yta kommer att se ut. För detta används s.k. Monte Carlo algoritmer, som är slumpmässiga funktioner med en deterministisk exekveringstid [14]. Slumpmässigheten leder till brus i bilden, eftersom två bredvidliggande punkter kan sampla ljus från helt olika riktningar. Då man låter algoritmen skjuta iväg tillräckligt många strålar, kommer den dock slutligen att konvergera mot en fotorealistisk bild utan brus. Denna metod leder även till fotorealistiska bilder. I praktiken kan man försnabba denna metod genom att istället för att enbart välja riktningar slumpmässigt, välja att undersöka riktningar med stor sannolikhet till att bidra mycket till belysningen. T.ex. kan det löna sig att undersöka belysningen från en ljuskälla, och att kolla om det finns någon geometri som är i vägen för ljuset. Mera om path tracing, och resultat som fåtts gällande snabbhet, kommer jag att behandla i avsnitt 3.1.1. En annan möjlighet är att använda path

tracing så att strålar från både kameran och från ljuskällorna bildar vägar i 3D-världen. Dessa kombineras då ihop för att bilda stigar från kameran till ljuskällorna. Detta kallas för bidirectional path tracing, och mera om det läsas av Fortune et al [15].
Renderingstiden påverkas av bl.a. geometrins storlek, mängden strålar som skjuts, den datastruktur som används för att accelerera korsningstest mellan strålar och geometri.
Ett självklart sätt att försnabba renderingen skulle vara att använda väldigt enkel geometri. Detta skulle t.ex. möjliggöra nya versioner av gamla eller annars simpla spel med ny, fotorealistisk grafik. Det finns en sådan implementation av datorspelet Quake Wars [16]. Sfera är ett simpelt spel där alla objekt är sfär-formade, men spelet är trots det grafiskt imponerande, med realistiska reflektioner. Spelets skapare skriver om spelet på Luxrenders forum och källkoden finns även tillgänglig [17].
Mängden strålar som skjuts påverkas av skärmens storlek, då det skjuts en stråle för varje pixel. Då probabilistiska metoder används, kan man välja hur många strålar som skjuts, flera strålar, innebär långsammare rendering, men bättre grafisk kvalitet.
I detta papper ägnar jag samla resultat om hur snabbt ett urval metoder kan rendera 3D-modeller med dagens algoritmer och hårdvara. Jag kommer att fokusera på snabbhet, trotts att det finns andra egenskaper som kan evalueras. Olika metoder har olika egenskaper när det gäller enkelhet, kvalitet, minnesbruk m.m. Minnesbruk kommer jag endast att ta upp då det kan påverka snabbheten. Kvalitet är också relevant för snabbhet, då mindre kvalitetskrav ofta gör renderingen snabbare, men jag kommer inte att jämföra kvaliteten hos olika metoder eftersom jag anser det vara utanför detta pappers omfång.
Hårdvaran som används är även något som påverkar snabbheten. För att snabba upp de elementära ray tracing operationerna, att besöka noder i accelerationsstrukturen och att testa om strålar och primitiv korsar varandra, används GPU-implementationer. Användning av GPUn försnabbar beräkningar då de kan utföras parallellt, men detta förutsätter någon form av koherens av det data som används för beräkningarna. Inkoherenta strålar är ett centralt problem i ray tracing, eftersom sekundära strålar som studsar i en modells geometri kommer att ha väldigt olika riktningar och kollisionslägen. För att uppnå koherens, kan man t.ex. omgruppera strålar vars vinkel skiljer sig litet från en utvald riktning, i en konform [7]. Det har även visats att val av scheduleringsalgoritm påverkar snabbheten [18]. Jag kommer inte att fokusera på GPU-implementationer i mitt arbete, men mera om de aspekter som påverkar prestanda i GPUn kan läsas i konferenspubliceringar av Aila et al [19] [18].

2.1 Teori

I detta avsnitt beskrivs teori gällande ljusets egenskaper som är relevanta för att modellera ljus. Ljuset från olika källor omkring oss gör det möjligt för oss att se vår omgivning. Ljuset

har egenskaper som t.ex. emission, reflektion, transport inuti ämnen och indirekt belysning som får en scen att se mera intressant ut. Ljuset verkar vara en så naturlig del av vår värld att vi sällan tänker på hur komplicerat det skulle vara att modellera med en dator.
Ljuset består i verkligheten av elektromagnetiska vågor, vilket syns praktiskt i vissa av ljusets egenskaper, t.ex. Doppler-effekten som får ljus som studsar från ytor med hög hastighet att byta våglängd. Ljusets vågegenskaper är inte viktiga att implementera vid rendering, eftersom de bidrar väldigt litet till en färdig bild. Istället modelleras ljuset som strålar, vars färd simuleras geometriskt.
Ljus från en ljuskälla färdas rakt ut i alla riktningar, tills det når något material. Från en yta kan en del av ljuset reflekteras med en vinkel som beror på infallsvinkeln, medan en del kan fortsätta inne i mediet med en ny vinkel beroende på materialet.

2.1.1 Renderingsekvationen

Hur ljuset reagerar till ytor i vakuum beskrivs av renderingsekvationen, som uppfanns av David Immel och James Kajiya år 1986 [20] [13]. Ekvationen beskriver hur ljuset beter sig då den träffar ytor, och implementation av den leder till realistiska bilder. Ekvationen grundar sig på energins bevarande.
Enligt renderingsekvationen är den utgående radiansen från en yta, i position x och vinkel ω, lika med summan av det reflekterade ljuset och det emitterade ljuset från ytan. Funktionen beror på läget $\boldsymbol{x}$, tiden $\boldsymbol{t}$ och ljusets vinkel utåt från ytan, $\boldsymbol{\omega}$.

$$L_O(\boldsymbol{x}, \boldsymbol{\omega}\ ) = L_r(\boldsymbol{x}, \boldsymbol{\omega}\ ) + L_e(\boldsymbol{x}, \boldsymbol{\omega}\ ) \tag{1.1}$$

Det reflekterade ljuset beräknas som en integral av den inkommande radiansen, den bi-direktionella reflektans-distributionsfunktionen och infallsvinkeln från alla riktningar i det övre halvklotet runt ytans normal.

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) = \int_{\Omega+} L_i(\boldsymbol{x}, \boldsymbol{\omega}_i) f_r(\boldsymbol{x}, \boldsymbol{\omega}_i \rightarrow \boldsymbol{\omega})(\boldsymbol{\omega}_i \cdot \boldsymbol{n}) d\boldsymbol{\omega}_i \tag{1.2}$$

Här betyder Ω+ det övre halvklotet runt ytans normal, $f_r(\boldsymbol{x}, \boldsymbol{\omega}_i \rightarrow \boldsymbol{\omega})$ betyder den bi-direktionella reflektans-distributionsfunktionen (BRDF) och $(\boldsymbol{\omega}_i \cdot \boldsymbol{n})$ är en faktor som försvagar ljusstyrkan beroende på vinkeln mellan infallsvinkeln och normalen. I ekvation (1.2) har jag använt $\boldsymbol{\omega}$ för den utgående vinkeln och $\boldsymbol{\omega}_i$ för den inkommande vinkeln som integreras.
F. E. Nicodemus beskrev år 1965 hur ljuset beter sig vid kollisioner med ytor [21]. Han föreslog en funktion BRDF, som bestämmer hur ljuset reflekteras från en yta. Den bestämmer hur mycket av det inkommande ljuset som reflekteras ut i en given vinkel, med en given infallsvinkel. Diffusa ytor, som t.ex. papper, reflekterar ljus effektivt åt alla håll, och för ideala diffusa ytor, s.k. Lambertiella ytor, är BRDF bara en konstant. BRDF som ägnar vara realistiska har de egenskaper att de är symmetriska, dvs. $f_r(\boldsymbol{\omega}_i \rightarrow \boldsymbol{\omega}_o) = f_r(\boldsymbol{\omega}_o \rightarrow \boldsymbol{\omega}_i)$ och bevarar energin, dvs. integralen över det reflekterade ljuset i ekvation (1.2) inte är större än

1.
BRDF har dock den nackdelen att de inte tar i beaktande ljusets transmission inuti ämnen. I verkligheten reflekteras en del av ljuset från ytan, medan en del kan brytas och färdas inuti ämnet, och möjligtvis interagera med flera nivåer av olika ämnen förrän den slutligen kommer ut. Ljuset kommer inte heller nödvändigtvis att komma ut från samma ställe där det gick in. För att implementera ljusets refraktion och transmission inuti olika ämnen kan BRDF utvidgas till BSSRDF, dvs. Bidirectional Surface Scattering Reflectance Distribution Function. Kalle Koutajoki beskriver i sin raport år 2002 egenskaperna hos BSSRDFs som utvecklades av Henrik Wann Jensen [22]. Simulering av ljusets spridning och transmission inuti ämnen bidrar mycket till den renderade bildens trovärdighet. För att rendera människans hy på ett realistiskt sett, måste man ta i beaktande ljusets interaktion med hyn, men också nivåerna under den, som blodådrarna, musklerna och benen. Donner et al. har beskrivit en metod för att modellera människans hy på ett realistiskt sätt [23].

6. Bibliography

[1] A. Appel, ”Some techniques for shading machine renderings of solids,” i *AFIPS '68 (Spring) Proceedings of the April 30--May 2, 1968, spring joint computer conference* , New York, 1968.

[2] T. Whitted, ”An improved illumination model for shaded display,” i *SIGGRAPH '05 ACM SIGGRAPH 2005 Courses*, New York, 2005.

[3] T. Nikodym, ”Ray Tracing Algorithm For Interactive Applications,” Czech Technical University, FEE., Prag, 2010.

[4] T. K. C. B. A. K. P. S. Ingo Wald, ”Interactive global illumination using fast ray tracing,” i *EGRW '02 Proceedings of the 13th Eurographics workshop on Rendering* , Aire-la-Ville, Schweitz, 2002.

[5] E. H. a. N. H. Tomas Akenine-Möller, ”Real-Time Rendering, Third edition,” A.K. Peters Ltd, 2008.

[6] A. Watt, 3D Computer Graphics, 3rd edition, Addison-Wesley, 1999.

[7] B. Mora, "Naive Ray-Tracing: A Divide-And-Conquer Approach," *ACM Transactions on Graphics (TOG),* vol. 30, no. 5, 2011.

[8] J. L. Bentley, ”Multidimensional binary search trees used for associative searching,” *Communications of the ACM,* vol. 18, nr 9, pp. 509-517, 1975.

[9] A. S. A. K. Maxim Shevtsov, ”Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes,” Computer Graphics Forum, 2007.

[10] T. W. Steven M. Rubin, ”A 3-dimensional representation for fast rendering of complex scenes,” i *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, New York, 1980.

[11] T. I. A. K. A. K. S. G. P. Ingo Wald, ”Ray tracing animated scenes using coherent grid traversal,” i *SIGGRAPH '06 ACM SIGGRAPH 2006 Papers*, New York, 2006.

[12] G. W. John G. Cleary, ”Analysis of an algorithm for fast ray tracing using uniform space subdivision,” *The VisualComputer,* vol. 4, nr 2, pp. 65-83, 1988.

[13] J. T. Kajiya, ”The rendering equation,” i *ACM SIGGRAPH Computer Graphics*, New York, 1986.

[14] G. Fishman, Monte Carlo: Concepts, Algorithms, and Applications, Springer, 2011.

[15] Y. D. W. Eric P. Lafortune, ”Bi-directional path tracing,” i *CompuGraphics*, Leuven, 1993.

[16] D. Pohl, ”Light It Up! Quake Wars Gets Ray Traced,” Intel Visual Adrenaline, 2009.

[17] D. Bucciarelli, ”Luxrender.net,” 10 December 2011. [Online]. Available:

http://www.luxrender.net/forum/viewtopic.php?f=17&t=7539. [Använd 5 April 2013].

[18] T. K. Timo Aila, ”Architecture considerations for tracing incoherent rays,” i *HPG '10 Proceedings of the Conference on High Performance Graphics*, Aire-la-Ville, Schweitz, 2010.

[19] S. L. Timo Aila, ”Understanding the Efficiency of Ray Traversal on GPUs,” i *HPG '09 Proceedings of the Conference on High Performance Graphics 2009*, New York, 2009.

[20] M. F. C. D. P. G. David S. Immel, ”A radiosity method for non-diffuse environments,” i *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques* , New York, 1986.

[21] F. E. Nicodemus, ”Directional Reflectance and Emissivity of an Opaque Surface,” *Applied Optics,* vol. 4, nr 7, pp. 767-773, 1965.

[22] K. Koutajoki, ”BSSRDF (Bidirectional Surface Scattering Distribution Function),” Helsinki University of Technology, Helsinki, 2002.

[23] H. W. J. Craig Donner, ”A spectral BSSRDF for shading human skin,” i *EGSR'06 Proceedings of the 17th Eurographics conference on Rendering Techniques* , Aire-la-Ville, Schweitz, 2006.

[24] D. L. J. Pantaleoni, ”HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry,” i *Conference on High Performance Graphics*, Saarbrücken, Germany, 2010.

[25] H. W. J. Toshiya Hachisuka, ”Stochastic progressive photon mapping,” i *SIGGRAPH Asia '09 ACM SIGGRAPH Asia 2009 papers*, New York, 2009.