Linköpings tekniska högskola
10 januari 2006

Volymrendering av moln

ett projekt i kursen
TNM022 av
Anders Fjeldstad

Inledning

Detta är en kort rapport som beskriver mitt arbete med ett projekt i kursen TNM047 Procedurella metoder för bilder. Mitt projekt gick ut på att implementera en volymrendering och modellering av moln. Syftet med projektet var att fördjupa mig i hur man använder OpenGL och GLSL för att programmera shaders på grafikkortet och utnyttja procedurella metoder för att rendera scener med komplexa objekt som inte går att representera som polygonmodeller på något effektivt sätt – moln är ett typiskt sådant objekt.

Implementering

Jag bestämde mig tidigt för att i första hand se till att få upp en fungerande rendering av ett tredimensionellt moln, gärna med animering, innan jag skulle bekymra mig för prestanda och ljusberäkningar. Detta innebar att jag först och främst ville implementera någon slags volymrendering, sedan lägga in en implementation av simplex noise och sedan konstruera själva molnmodellen. Sist tänkte jag i mån av tid undersöka alternativa metoder för accelerering och ljusberäkning.

Ett hjälpbibliotek för GLSL-programmering: libglsl

För att det inte skulle bli så mycket kod som rör själva inladdningen av OpenGL-extensions, kompilering och länkning av shaders och så vidare i huvudprogrammet så valde jag att använda *libglsl* som är ett öppet litet klassbibliotek som en man vid namn Martin Christen har skrivit för att göra livet lättare för GLSL-programmerare. Jag har angivit adressen till sidan där man kan hämta biblioteket i referenslistan.

Lånad simplex noise-implementering

I det här projektet har jag lånat Stefan Gustavsons implementering av simplex noise från 2005. Beräkningarna görs i fragmentshadern, med stöd av en perturberingsmatris och en gradientmatris som skickas från huvudprogrammet som texturer. Webbadressen till implementationen finns i referenslistan.

Volymrendering

Det finns många sätt att implementera volymrendering. Jag valde att använda en uppsättning transparenta plan som är parallella med betraktarens bildplan hela tiden. Varje punkt på planen får sitt färgvärde genom att sampla volymen. Resultatet blir ett antal tvådimensionella bilder i en "stack" vänd mot betraktaren, som får intrycket av en tredimensionell volym. Kvaliteten på renderingen kan justeras genom att variera antalet plan som skär volymen.

Användaren kan undersöka volymen från alla möjliga vinklar genom att rotera på den med hjälp av musen. Rent tekniskt går det till så att rotationsmatrisen skickas in till fragmentshadern som applicerar den på sampelpunkternas koordinater i texturerings-steget.

I mitt fall är volymen rent procedurell, men metoden kan naturligtvis även användas för rendering av en tredimensionell textur som man läst in från disk.

Figur 1. Molnets makrostruktur.

Figur 2. Den slutgiltiga renderingen av molnet.

Molnens struktur

När jag har konstruerat själva molnens struktur så har jag till stor del använt mig av den metod som David Ebert (2003) beskriver. Molnens makrostruktur – alltså deras huvudsakliga form – definieras av ett godtyckligt antal implicita primitiver. Det kan vara vilka primitiver som helst, men jag har mest använt mig av sfärer och plan i mina exempelscener eftersom de är så lätta att definiera. Opaciteten avtar mot kanterna på primitiverna så att inga skarpa kanter ska synas. Figur 1 demonstrerar principen. Det är väldigt enkelt att få till i princip vilka former man vill.

Molnens mikrostruktur – alltså hur de ser ut på detaljnivå – bestäms sedan av en serie perturberingar av varje punkt. Det börjar med att litet simplex noise adderas till punktens position. Sedan läggs ytterligare turbulens till genom att fyra oktaver av litet mer simplex noise läggs ihop och blandas med de befintliga koordinatvärdena. Det är dessa perturberade punkter som kontrolleras mot de implicita primitiverna, vilket via ytterligare något mellansteg ger det slutgiltiga opacitetsvärdet. Den resulterande renderingen av grundformen i figur 1 visas i figur 2.

Om man vill veta i detalj hur beräkningarna av mikrostrukturen ser ut så rekommenderar jag att man antingen läser Ebert (2003) eller tittar i källkoden för fragmentshadern från mitt projekt. I stora drag kan nedanstående pseudokod beskriva det hela:

```
dist = simplexnoise(3.0*pointPosition);
pointPosition = pointPosition + 0.1*dist;
turb = turbulence(2.5*pointPosition);
pointPosition.x = pointPosition.x + 0.3*turb;
pointPosition.y = pointPosition.y - 0.3*turb;
pointPosition.z = pointPosition.z + 0.3*turb;
metaDensity = metaballDensity(pointPosition.xyz);
cloudDensity = 0.3*(0.4*metaDensity +
                    (1.0-0.4)*turb*metaDensity);
```

Interaktion

Som jag nämnt ovan så är det i mitt program möjligt att titta på de renderade molnen från vilket håll man vill. Användaren håller helt enkelt vänster musknapp nedtryckt och flyttar sedan på musen för att rotera scenen. För att det hela ska flyta lite bättre så visas endast molnens makrostruktur under interaktionen. Dessutom visas koordinataxlarna som olikfärgade streck för att det ska bli lättare att orientera sig.

Ett litet "tjuvknep" som jag själv använt när jag renderat mina exempelbilder är att minska själva programfönstrets storlek medan jag söker en bra vinkel, så att det blir färre pixlar för shadern att jobba med. På det sättet kan jag på min nuvarande dator (en AMD Athlon XP 1800+ med ett Gigabyte GeForce 6600GT – inte det kraftfullaste systemet alltså) få interaktiva renderingstider per bildruta. När man sedan vill titta närmare på detaljerna i bilden förstorar man fönstret igen.

Ljusberäkningar

En viktig del av rendernigen är förstås ljusberäkningarna. När jag hade strukturmodellen för molnen klar beslöt jag mig för att inrikta mig på att fixa åtminstone någon form av skuggning, eftersom det ger så mycket till slutintrycket av en bild.

Jag funderade på hur jag skulle kunna få in självskuggning på ett hyfsat effektivt sätt och kom efter ett tag fram till en metod som fungerade bra. Innan jag beskriver hur den funkar tänkte jag dock förklara den bakomliggande principen.

I och med att molnet är halvtransparent kan ljus färdas inuti volymen och intensiteten avtar med sträckan. Det handlar alltså om att för varje punkt i molnet beräkna den totala intensiteten, vilket kräver att man kan få ut transportsträckan på något sätt. Naturligtvis är detta en stor förenkling – om man skulle göra det ordentligt så skulle man bli tvungen att ställa upp en modell för hur ljuset sprids också, men det är helt klart att man måste dra gränsen någonstans om man vill att det hela ska gå relativt snabbt att räkna ut.

Ett problem med att implementera ovanstående princip direkt är förstås att det finns en ruskig massa punkter inuti molnet och det är bara ett fåtal som överhuvudtaget syns, så större delen av den tid man skulle lägga på beräkningarna skulle man inte få igen i bildkvalitet.

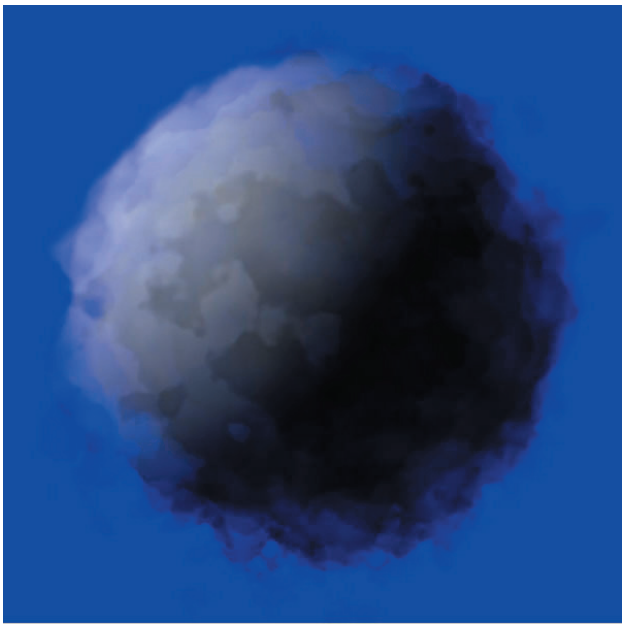
Jag kom på att man på ett enkelt sätt kan få ut en approximativ självskuggning där man även utnyttjar att grafikkortet är bra på att interpolera värden. Jag såg helt enkelt till att ge varje plan en viss upplösning av hörnpunkter, något jag tidigare hade struntat i eftersom alla betydelsefulla beräkningar av det visuella gjordes i fragmentshadern, vilken inte känner till något om hörnpunkter.

Varje hörnpunkt på varje plan kontrolleras mot de implicita primitiverna som definierar molnets huvudsakliga form. Om punkten i fråga ligger inuti volymen räknas sträckan som ljuset färdats i molnet ut (jag summerar steg av fast längd längs en stråle från punkten till ljuskällan) och läggs till grund för beräkningen av ett skuggvärde för hörnpunkten. I fragmentshadern kan man sedan få ut skuggvärden för godtyckliga punkter på planen i och med att interpoleringen sker automatiskt, och dessa värden används för de perturberade punkter som syns i den slutgiltiga konstruktionen av molnet. I figur 3 visas skuggningen av en implicit sfär, och i figur 4 visas samma skuggade sfär fast med fullständiga strukturberäkningar (i dessa figurer har jag medvetet överdrivit molnets ljusabsorbtion för att förtydliga principen).

Jag tycker att detta var en bra lösning eftersom det är lätt att variera upplösningen på "skuggvolymen" och eftersom självskuggningen (även om den bara är approximativ) på ett effektivt sätt hjälper till att dölja det faktum att volymen samplas ett begränsat antal plan.

En fin sak med min skuggningsmetod är att molnen automatiskt kastar skuggor på andra moln – det får man med på köpet i och med att varje molnpunkts intensitet baseras på hur mycket ljus den "ser" med hänsyn taget till alla molnmassor mellan punkten och ljuskällan. Detta illustreras i figur 5.

Figur 3. Skuggning av implicit primitiv.



Figur 4. Skuggning av fullständig molnstruktur.

Figur 5. Slagskugga från moln på annat moln.

Diskussion

I slutändan hade jag utvecklat ett program som faktiskt renderar ganska fina bilder av moln med en enkel men effektiv skuggningsalgoritm. Jag är ruskigt nöjd med resultatet, även om det finns en del punkter som skulle kunna förbättras.

Det första som slår en när man kör programmet på en dator som motsvarar den jag äger i skrivande stund är att renderingen är långsam – jag kommer i mina exempelscener inte ens upp i en bild per sekund när mikrostruktursberäkningen är påslagen. Anledningen till att det går segt är att det görs flera simplex noise-beräkningar i varje punkt på planen som ingår i molnet, oavsett om punkten kommer att synas i slutresultatet eller inte. Detta är förstås dumt, men det är så OpenGL gör; det tar ingen hänsyn till att opaciteten kan ha nått 1.0 när blending-beräkningarna görs för bakomliggande punkter.

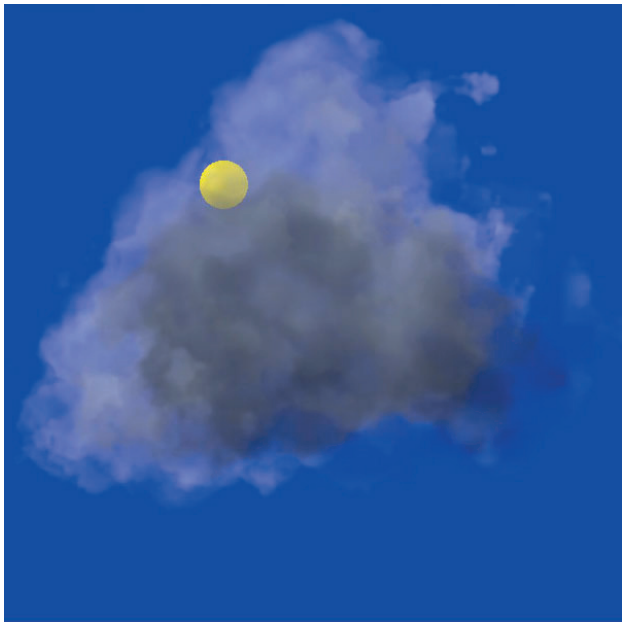
En faktor som skulle snabba upp det hela är förstås att inte beräkna simplex noise och turbulens på riktigt i varje punkt. Man skulle kunna använda någon slags lågupplöst perturberingsvolym som inte uppdateras så ofta för att "fuska", till exempel. Ebert (2003) nämner denna teknik. Jag kände dock att jag hellre fokuserade på att få till ljusberäkningarna än att accelerera en kod som faktiskt fungerar.

Ytterligare en möjlig förbättring av renderingen vore att ta hänsyn till punkter som ligger mellan planen för att få en mer korrekt bild. I min implementation så beror den renderade bilden kraftigt på antalet plan som skär volymen – om man har få plan så blir molnen helt enkelt tunnare vilket får dem att se ut som rök snarare än riktiga moln. Figurerna 6 och 7 illustrerar detta. För att lösa det hela skulle man behöva låta intensitetsvärdet i varje punkt på planen även innefatta någon slags integrering av intensiteterna mellan planen.

En faktor som jag initialt siktade på att ta med i projektet men sedan utelämnade är animering. Det hade varit fint att låta molnen driva med vinden, smälta samman och så vidare. I och med att min fullständiga rendering inte direkt går i realtid så kändes det dock oväsentligt för mig att fördjupa mig i denna nisch.

Att låta användaren bygga sina egna moln hade varit läckert, och inte särskilt svårt. I och med att molnen bygger på enkla former så skulle man kunna användaren modellera på ett liknande sätt som i till exempel 3D Studio och sedan låta programmet rendera molnen med full detaljrikedom. Detta var inte heller något som jag riktigt kände att jag hade tid med, och dessutom föll det inte inom ämnesområdet.

Slutligen vill jag säga att jag är riktigt nöjd med mitt arbete och jag är stolt över resultatet! Jag hoppas att någon som läser detta kan ha nytta av mina resonemang, och sedan själv producera något ännu bättre.



Figur 6. Rendering med 16 plan.

Figur 7. Rendering med 32 plan.

Referenser

libglsl 0.9.4. http://www.clockworkcoders.com/oglsl/downloads.html. (kontrollerad 2006-01-10)

Stefan Gustavsons implementation av simplex noise (2005). http://www.itn.liu.se/~stegu/TNM022-2005/lab3/GLSL-noise.zip. (kontrollerad 2006-01-10)

David Ebert (2003). *Texturing & Modeling, a Procedural Approach* (kap. 10). Morgan Kaufmann Publishers.