Simulering av blixt i realtid

Olof Landahl
TNM022 Procedurella Bilder
Linköpings Universitet, 2007

Inledning

Denna rapporten beskriver hur jag gick tillväga för att slutföra ett projekt i kursen Procedurella Bilder. Projektet gick ut på att implementera procedurella metoder för att skapa ett visuellt resultat. Det fanns stora valmöjligheter och jag bestämde mig för att simulera blixtar i realtid.
Målet var att få fram ett resultat där man får se en relativt verklighetstrogen och slumpmässig blixt som genereras och animeras i realtid.

För att generera en slumpmässig blixt använde jag mig av L-system, ett form av språk som lämpar sig för trädstrukturer.
Implementeringen gjordes i C++, OpenGL och GLSL för animering i realtid.

Teori

Blixtar
Blixtar är ett komplext fenomen som man fortfarande inte vet exakt hur det uppkommer. Man tror dock att det är lätta positivt laddade ispartiklar i molnen som stiger medans tyngre negativt laddade isvattendroppar sjunker neråt. De övre delarna av molnet är då positivt laddade medans de nedre delarna är negativt laddade. När laddningen blir tillräckligt stark sker en urladdning, oftast mellan molnens olika skikt, men ibland slår de även ner på jordens yta.

En ström bildas neråt och när den når marken sker huvudurladdningen som är den som ger störst utslag, både visuellt och ljudmässigt. Ofta sker flera urladdningar inom bara några tiondels sekunder men vi ser de oftast som en sammanhängande urladdning.

De blixtar som slår i marken har i regel en vertikal riktning och i de flesta fall bildas "grenar" från huvdurladdningen som sprider sig relativt vertikalt men med en viss vinkel mot "huvudgrenen". Varje gren har en oregelbunden form som ser ut som större eller mindre vågor som hela tiden skiftar amplitud och våglängd.

L-system
En biolog vid namn Lindenmayer forskade i hur vissa typer av alger växer och kom på ett system att skriva ner sina observationer. Senare kom detta system till användning när man ville beskriva hur plantor, och andra trädstrukturer, breder ut sig.

L-system är ett slags språk med egen grammatik som kan översättas till en struktur med fraktala mönster. Grammatiken består av en uppsättning tecken som motsvarar regler. Man brukar uttrycka L-systemet som en sköldpadda som rör sig efter dessa regler. De mest grundläggande tecknen är:

F , som betyder 'gå ett steg i aktuell riktning',
- , som betyder 'sväng till vänster',
+ , som betyder 'sväng till höger'.

FF-F++F betyder alltså: Gå fram två steg, sväng vänster och gå ett steg framåt i den nya riktningen, sväng till höger två gånger och gå ett steg framåt.

Det finns många fler regler men språket kan begränsas, utökas och modifieras utefter egna behov. Man måste även definiera exakt hur långt ett steg är och hur många grader man ska svänga.

Implementering

L-systemen genererades med C++ kod, den visuella implementeringen gjordes med OpenGL och för blixtens shader användes GLSL.
Jag använde hjälpbiblioteket GLFW för hantering av fönster, avläsning av tangentbordkommandon samt inläsning av texturfiler. Jag tog även hjälp av Libglsl för att smidigt hantera laddning och kompilering av GLSL-shaders. Se figur 6 för en sammanfattning av implementeringen.

Generering av L-system
För att få fram en blixts oregelbundna, men till viss del förutsägbara, form krävdes dels en generering av oregelbundna linjer som går i en ungefärlig riktning och ett samband mellan dessa linjer, dvs att en huvudgren har några mindre grenar som utgår från den och att dessa grenar i sin tur också har grenar på sig.

Varje gren skapas med hjälp av en 'L-system generator' som i varje iteration tar ett slumpmässigt tal mellan 0 och 1 och beroende på det värdet lägger den till ett visst tecken/regel i L-systemet.
I mitt L-system har jag endast F, -, + och b som regler, med dessa definitioner:

F = gå 5 enheter i aktuell riktning.
- = sväng 30 grader åt vänster.
+ = sväng 30 grader åt höger.
b = lägg till en gren som utgår från aktuell position.

L-systemet översätts sedan enligt dessa regler, i en tolk, till koordinater för segment i två dimensioner. Dessa segmentkoordinater lagras i en vektor som tills vidare är definitionen för den grenen. De olika grenarna har alltså varsin vektor med koordinater. Dessa sätts ihop till en gemensam vektor som då innehåller koordinater för hela blixten.

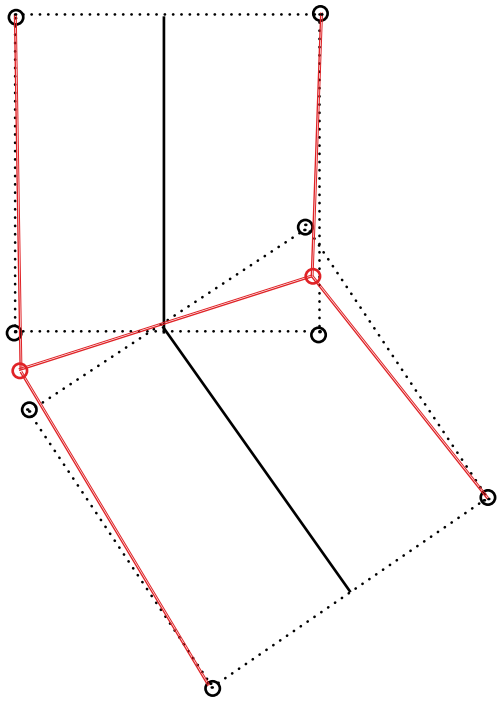
Generering av polygoner

För att få ett visuellt resultat av detta sätts de olika segmenten ihop som polygoner. I OpenGL görs detta lättast med s.k. quad-strips som sätter ihop en rad polygoner efter varandra.
Eftersom varje segment endast kan tolkas som en tvådimensionell linje måste de räknas om från två till fyra koordinater. Detta görs genom att helt enkelt definera en bredd för grenen (som är olika beroende på vilken typ av gren det är). Eftersom segmenten har olika vinklar får de olika koordinater där de möts. För att koppla ihop de väljs koordinater som ligger mellan kandidaterna (se figur 1). På detta vis läggs polygoner till allt eftersom programmet itererar genom vektorn med segment.

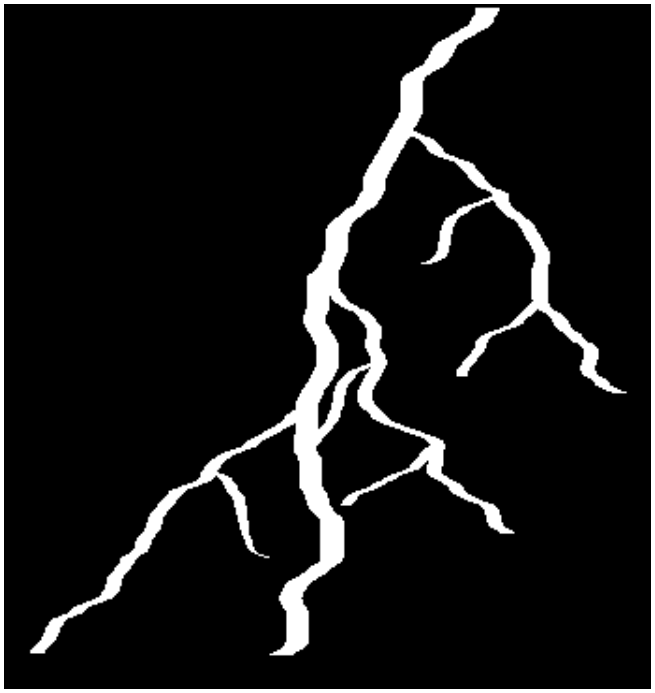
När respektive gren börjar närma sig sitt slut görs polygonerna smalare i varje iteration för att få avsmalnande effekten som ofta finns hos blixtar, bortsett från huvudgrenen. Resultatet kan ses i figur 2.

Shading av blixtobjektet

Det visuella resultatet av blixten i figur 2 är inte så imponerande. För att göra den mer verklighetstrogen och levande (i realtid) görs en shader i GLSL som sätter färg och opacitet på objektet beroende på texturkoordinater för



Figur 1. *Segmententen var för sig representeras av svarta linjer och punkter. De färdiga ihopkopplade polygonerna representeras med röd färg.*

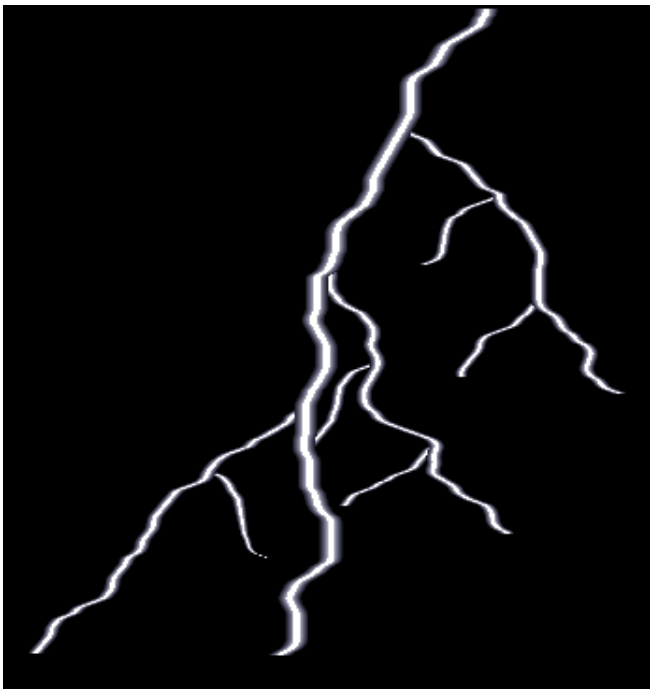


Figur 2. *Blixtobjektet uppbyggt av polygoner.*

polygonerna samt den tid som har gått sedan blixten dök upp.

En Vertex Shader skapas för att ta in hörnpunkter och modifiera dessa. I detta fallet används den dock inte till mer än att skicka texturkoordinater, samt räkna ut och skicka avståndet mellan aktuella hörnpunktskoordinater till startkoordinaterna för blixten, till en Fragment Shader.

Denna Fragment Shader använder endimensionella texturkoordinater (eftersom blixten kan beskrivas som en endimensionell linje med en fast bredd) för att räkna ut vilken färg blixten ska ha. Det ska vara vitt i mitten som avtar mot kanterna där den är helt genomskilnig. Resultatet kan ses i figur 3.
Alphakanalen som används för att representera det genomskilniga varierar även med tiden och avståndet till blixtens startpunkt. Från början ska blixten synas mest där den startar och med tiden syns mer tills hela blixten lyser upp och sedan försvinner.



Figur 3. *Blixtobjektet med shader.*

Skapande av värld
För att blixten ska få ett bättre sammanhang har en värld, i form av texturmappning av en kub, skapats där blixten uppstår.
Genom att använda sig av cube map i OpenGL kan man mappa sex sammanhängande texturer på insidan av en kub för att t.ex. efterlikna ett landskap. Texturerna som används i simuleringen är hämtade från [5]. Resultatet med blixten i det nya landskapet kan ses i figur 4.

För att ge ytterligare realism till animationen av blixten lyser landskapet upp då blixten visas. Detta görs mycket enkelt genom att istället använda ljusare versioner av texturerna då blixten syns.

Figur 4. *En komplett scen med blixten i en mörk "ovädersmiljö".*

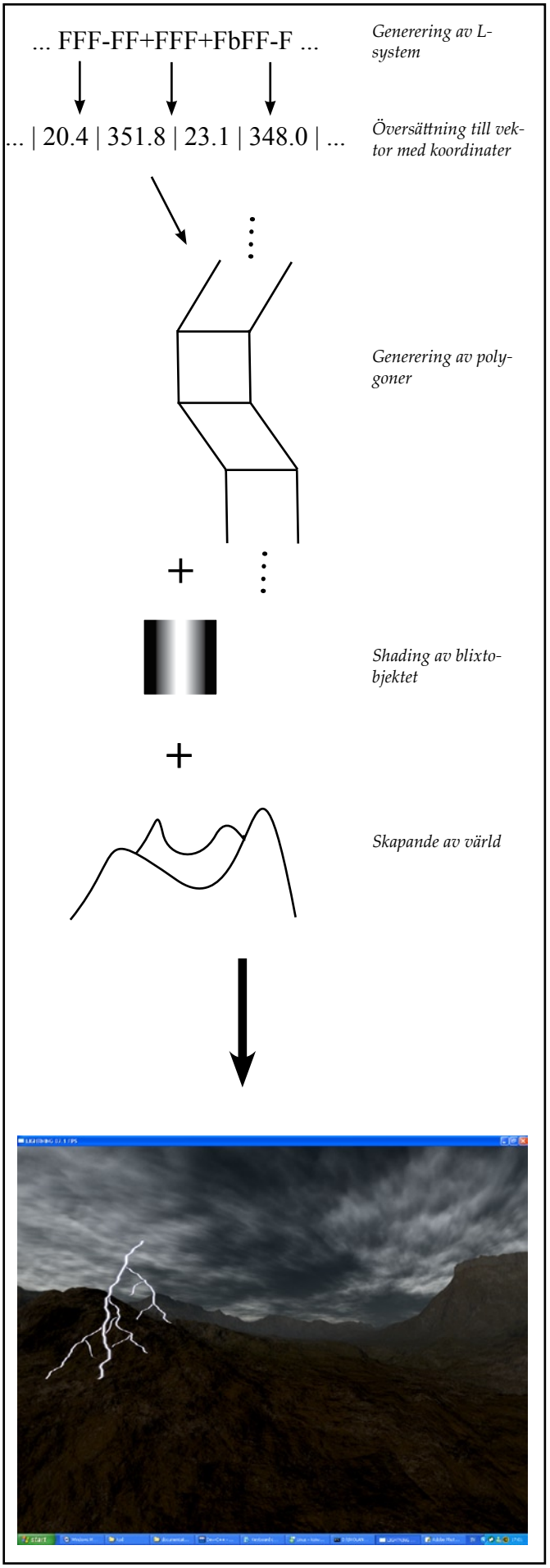
Resultat

Det slutgiltiga resultatet blev en blixt, uppbyggd av 2D-polygoner i en 3D-värld som animeras med en shader (se figur 6).
När programmet startar ser man endast det mörka landskapet. För att sedan generera en ny blixt och animera den, samt sätta på och stänga av funktioner, använder man specifika tangenbordskommandon (se figur 5).

ESC - Avsluta programmet
Right Shift - Generera ny blixt
Enter - Visa blixt
W - Visa landskap (på från start)
Q - Visa inte landskap
S - Sätt på shader'n (på från start)
A - Stäng av shader'n
X - Tillåt animation av blixt (på från start)
Z - Tillåt inte animation av blixt

Figur 5. *Tengentbordskommandon för programmet.*

En blixt skapas med en slumpmässig position och form. Den animeras sedan i ca 1 sekund, samtidigt som landskapet lyses upp, innan blixten försvinner igen. Det är betydligt längre än vad en blixt i verkligheten syns, men för att man lättare ska hinna se blixten är tiden så lång.



Figur 6. *En sammanfattning hur blixtsimuleringen går till i programmet och resultatet.*

Prestanda

På min dator, en Pentium D 2.8 GHz med GeForce 7300, renderas, i 1024x768 upplösning, blixten och landskapet med ca 80 bilder/sekund. Detta, tillsammans med att genereringen av en ny blixt går på nolltid, gör att simulering i allra högsta grad är i realtid.
På skolans dator, som har ett GeForce FX 5700LE grafikkort, fungerar simuleringen sämre eftersom polygonerna inte hinner renderas innan shader-animationen börjar. Vid prestandaproblem kan man testa att stänga av animationen genom att trycka 'Z' på tangentbordet. Om det går allmänt segt kan man även förminska fönstret och ta bort bakgrunden, genom att trycka 'Q' på tangentbordet, för att få en högre framerate.

Brister och möjliga förbättringar

Vissa brister finns i simuleringen. Ett exempel är att varje gren görs till ett självständigt objekt vilket gör att det bildas små mellanrum i blixten när shadern läggs på (se figur 3 & 4).
Ett annat exempel är att blixtens animation egentligen borde bero på vilken gren som animeras och inte bara avståndet till startpunkten.
En stor förbättring man skulle kunna göra är att göra blixtobjektet tredimensionellt med t.ex. cylindrar istället för polygoner.
Ännu en förbättring hade varit att även skapa en shader för landskapet för en mjukare övergång av ljusstyrka.

Slutsats

Dessa förbättringar är relativt lätta att göra men p.g.a. tidsbrist har jag låtit det vara. Målet är trots allt uppfyllt och jag är nöjd med resultatet. Mycket av tiden har jag lagt ner på att läsa om och programmera i C++ och OpenGL, vilket inte var det huvudsakliga syftet med projektet men å andra sidan har jag lärt mig en hel del som jag kan ha nytta av i framtiden. Förutom rena programmeringsproblem lade jag ner mycket tid på att få det slumpmässiga L-systemet att bete sig som jag ville. Det krävdes mycket tankearbete och framför allt testning av olika if-statements m.m. Om någon tänker implementera L-system i något sammanhang kan jag rekommendera att att hitta färdig kod som implementerar L-system och att först ha ett program som ritar ut L-systemet som linjer i 2D. Det underlättar mycket när man testar sig fram.

Referenser

[1] Wikipedia, http://www.wikipedia.org; http://en.wikipedia.org/wiki/Lightning http://en.wikipedia.org/wiki/L-system

[2] John C. Hart (m.fl.), "Texturing & Modeling - A Procedural Approach", tredje upplagan, s. 307 - 312, Morgan Kaufmann Publishers, 2003

[3] Richard S. Wright Jr. & Michael Sweet, OpenGL SuperBible, andra upplagan, Waite Group Press, 2000

[4] John Kessenich, The OpenGL® Shading Language, 3Dlabs, Inc. Ltd., 2006

[5] Codemonsters; http://www.codemonsters.de/html/textures_cubemaps.html