Procedurell 3D-eld på grafikkortet

TNM084 – Procedurella metoder för bilder

Anders Hedblom, andhe893@student.liu.se

2012-04-04

1. Bakgrund

1.1. Procedurella metoder

Procedurella metoder har ett stort användningsområde inom datorgrafik. Det handlar om att skapa texturer genom matematiska beräkningar istället för att använda färdiga texturer som läses från ett minne. Texturerna kan sedan användas för en mängd olika ändamål, t.ex. displacement mapping eller vanlig tvådimensionell texturering. Detta medför många fördelar:

Eftersom texturerna räknas ut och skapas vid exekvering av programmet behöver de inte sparas på något minne. De gör att man kan få väldigt kompakta program med komplexa innehåll. Med procedurella metoder kan man också skapa mönster som slumpmässigt skiljer sig ifrån varandra vid varje exekvering. På samma sätt kan man skapa rörelser hos objekt som aldrig är lika från den ena gången till den andra. Det är också väldigt enkelt att förändra och variera texturerna genom att justera de matematiska parametrarna som används.

En nackdel med att skapa texturer genom procedurella metoder är att det kan vara svårt att få just det mönster man söker efter. Då är det lättare för en artist att rita detta för hand.

1.2. Eld

Något som beter sig väldigt slumpmässigt i våra ögon är eld. Därför lämpar sig procedurella metoder väldigt bra för att skapa eld. Man slipper då använda förrenderade klipp på brinnande eld, som måste laddas in till minnet. Eld är egentligen gas som är upphettat till plasma och beter sig därför som en fluid. Om man ska göra mycket realistisk eld bör man därför använda sig av en fluidsimulering. Detta är dock en metod som är väldigt beräkningstung och svårt att göra i realtid. Det går dock att fuska och efterlikna eld med noise.

2. Metod

2.1. Perlin Noise

Figure 1. En oktav av Perlin Noise

Perlin Noise, skapat av Ken Perlin, är en effekt som används för att skapa slumpmässiga naturliga mönster, som i *figur 1*. Eld, rök och moln är exempel på vad som kan anses ha slumpmässiga utseenden. Perlin Noise existerar i flera versioner, t.ex. kan 3D Noise användas för att skapa en tvådimensionell textur som varierar med tiden. På samma sätt kan *n* dimensioner av

noise användas för att skapa mångdimensionella texturer. För mer information om hur noise faktiskt fungerar rent matematiskt hänvisar jag till Stefan Gustavsons rapport; Simplex noise demystified: http://webstaff.itn.liu.se/~stegu/TNM084-2011/simplexnoise-demystified.pdf

För det här projektet användes Perlin Noise som en funktion som tar in *x-, y-* och *z-position*, i rummet. Den tar även in tiden som en parameter. All kod som skapar noise är skriven i GLSL och körs på grafikkortet. 4D Noise-funktionen som används är skriven av Stefan Gustavson, forskare och lärare på Linköpings universitet, och finns tillgänglig på https://github.com/ashima/webgl-noise/blob/master/src/classicnoise4D.glsl.

2.2. Noise med shaders

En shader är ett litet program som exekveras på grafikkortet. Att använda grafikkortet för att göra parallella beräkningar är ofta ett utomordentligt sätt att snabba upp program. Detta lämpar sig t.ex. bra i fall där man kan jobba enskilt med en pixel åt gången, utan att behöva bry sig om intilliggande områden. Shadern som används för att skapa elden i det här projektet kallas för Fragment shader. Den påverkar enskilda pixlar hos en textur, medan en Vetrex shader jobbar med att förflytta punkterna som bygger upp polygoner. Den används dock endast som en mellanlandning här. I Fragment shadern finns en funktion för Perlin Noise i fyra dimensioner samt olika anrop av denna funktion. Det enda funktionen ger tillbaka är ett flyttal mellan minus ett och ett.

Det här innebär att en godtycklig position i rummet vid godtycklig tid kan matas in i noise-funktionen för att få ut ett enstaka pixelvärde. Men andra ord kan vi skapa en 3D-textur som varierar med tiden. Än så länge har den dock ingen liknelse vid eld.

2.3. Strukturen på Eld

Genom att kombinera flera olika skalor av noise kan man skapa så kallat fraktalt noise. Det innebär att man kan få delar där större mönster kan urskiljas, men som på djupare nivå innehåller noise i sig självt.

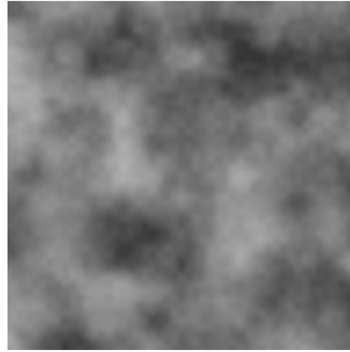


Figure 2. Flera oktaver av Perlin Noise.

Så hur kan vi då beskriva eld med olika noiseanrop? Först behöver vi veta vad som ska uppnås:

- Eld kan oftast beskrivas genom att den har en fyllig grund som blir allt med findetaljerad och mer högfrekvent varierande ju längre ut mot eldens kanter man kommer.
- Den rör sig också snabbare ut mot kanterna.
- Eftersom den är varmare än sin omgivning bildas luftströmmar som är riktade uppåt.
- Rent färgmässigt kan eld variera kraftigt beroende på vad som brinner. Allmänt brukar kärnan på lågan ha ett mer intensivt vitaktigt ljus. Intensiteten avtar och blir mer rödaktig i kanterna.

För att sätta en färg och intensitet i en pixel kan vi nu köra den kod som finns i slutet på rapporten, vilket i princip är all kod som behövs för att ge elden dess karaktär.

Vektorn *stu* är position i rymden. Variablerna *fadex, fadey* och *fadez* är till för att elden inte ska se ut som en låda utan dämpas mot kanterna. Variabeln *n* är den som bestämmer texturvärdet för varje position i volymen. För varje nytt anrop av noisefunktionen *cnoise* förändras dess värde. Eftersom *cnoise* ger ett värde mellan -1 och 1 används absolutbeloppet av det returnerade värdet. På så vis skapas de ”vassa kanterna” som eld gärna har, istället för att ge en mjuk variation. Konstanten som står först på varje rad där *n* tilldelas ett värde bestämmer hur mycket av den noiseoktaven som ska påverka slutvärdet. Konstanterna framför *stu.x, stu.y* och *stu.z* bestämmer storleken på noiset. En större konstant innebär högre frekvens på den

spatiala variationen. Samma sak gäller konstanten framför *time*. Ett större värde ger snabbare variation med tiden.

De sista raderna skapar den färggradient som behövs för att likna eld. Vad de gör rent matematiskt är att fördröja värdet i de olika färgkanalerna. Om värdet som står först i funktionen *clamp* är mindre än 0 returneras 0. Om det är större än 1 returneras 1. Vid små *n* får bara den röda färgkanalen ett värde över 0. Sen kommer den gröna kanalen in för att skapa en mer gulaktig färg. Sist aktiveras även den blå kanalen, vid större *n*, för att skapa en vit färg. Slutligen sätts pixelns färg med *gl_FragColor*.

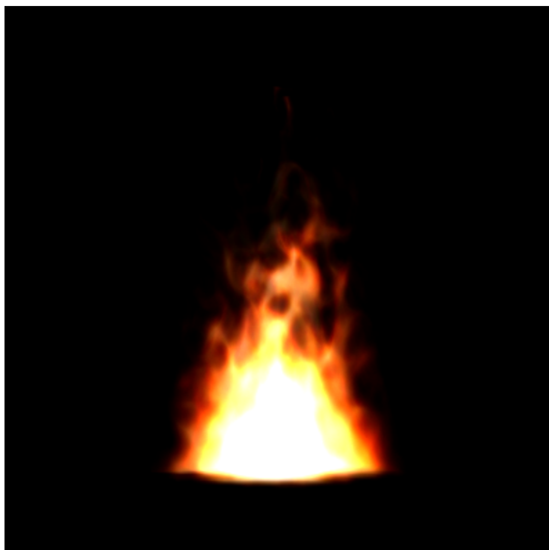
2.4. Att visualisera elden

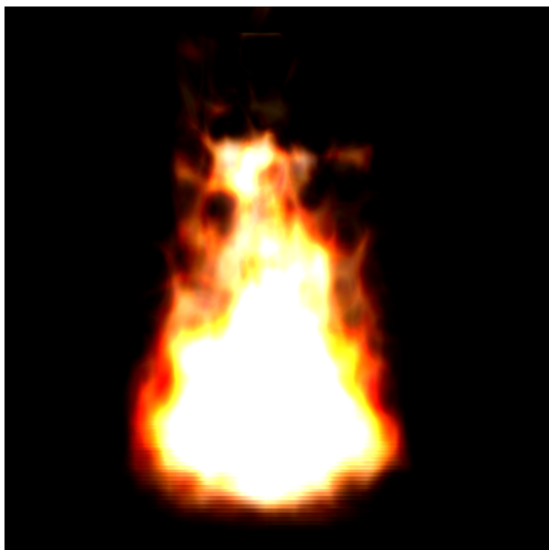
Eftersom eld är genomskinligt skapa en del komplikationer när den ska visualiseras. Varje pixel på skärmen måste ju visa en blandning av alla texturvärden som skapas i djupled. Här är Raycasting en bra metod. Det innebär att man skickar en ”stråle” från varje pixel i bildplanet (eller från en kamera genom bildplanet för perspektivprojektion) rakt in i rymden. Om en stråle passerar elden hittar man positionen där eldens voxelrymd passeras för första gången och sedan stegar man sig igenom elden i strålens riktning. För varje steg adderar man det texturvärde som hittas till en variabel. När man passerat hela volymen får pixeln i bildplanet det värden som variabeln fick.

För enkelhetens skull valde jag dock att lösa detta problem på ett mycket enklare sätt. Genom att sätta ut ”slices” i en kubform, sätta pixelvärdena på varje slice och sedan blenda ihop dessa med OpenGL’s inbyggda funktion *glblend*, skapas ett motsvarande resultat. Nackdelen är dock att man bara kan titta på elden väldigt begränsat, nämligen ortogonalt mot alla slices, men man slipper hanterandet av vektorer hit och dit.

3. Resultat

Det slutliga resultatet blev ett mellanting mellan eld i två och tre dimensioner. Det viktiga är att elden har texturvärden i tre dimensioner, vilket gör att den varierar även i djupled. Själva visualiseringen är dock mer av en tvådimensionell projektion för att göra det hela en aning smidigare. De flesta konstanter har jag testat mig fram till och gett ett värde efter vad som sett bra ut. Här är några olika typer av eld där jag påverkat variabeln *n*, slutfärgen och även hur många slices som renderats:





4. Diskussion och förbättringar

Att köra slutresultatet kräver ett riktigt bra grafikkort om man vill ha en rimligt hög biluppdatering. Jag har kört med ett Nvidia GTX570 och kan få cirka 20 fps när elden täcker ett 400x400 pixlar stor fönster. Då är antalet djupsteg 80 st. Bilduppdatering uppsnabbas förstås om man zoomar ut lite. Jag har testat att använda både Classic Noise och den nyare versionen Simplex Noise. Det senare alternativet ger nästan en fördubblad fps, men jag tyckte personligen att Classic Noise gav ett snyggare resultat.

Det finns en mängd förbättringar som skulle kunna göras med denna eld. Först och främst, om man implementerar en Raycaster kan man få en mycket tydligare känsla av 3D och kunna flytta runt kameran hur som helst utan att kunna se några slices. Vidare skulle man även kunna implementera Flow Noise, som använder sig av derivator för att skapa en mer flödesliknande förändring i texturen.

Man skulle också kunna lägga in en enkel interaktion med elden genom att justera de övre texturkoordinaterna när man flyttar omkring elden.

5. Referenser

http://en.wikipedia.org/wiki/Perlin_noise

https://github.com/ashima/webgl-noise/blob/master/src/classicnoise4D.glsl

http://webstaff.itn.liu.se/~stegu/TNM084-2011/simplexnoise-demystified.pdf

Kodurklipp

```
1.  void main (void)
2.  {
3.      float fadex = (0.6-0.1*stu.y-abs(0.5-stu.x))*3;
4.      float fadey = 1.15 - 0.5*stu.y;
5.      float fadez = 1.4 - abs(stu.z);
6.
7.      float n = 0.0;
8.      stu.y -= time*1.8;
9.      n = 0.05*(2.2-fadey)*abs(cnoise(vec4(16.0*stu.x, 12.5*stu.y, 16.0*stu.z, 2.8*time)));
10.     n += 0.4*(1.8-fadey)*abs(cnoise(vec4(8.0*stu.x, 6.0*stu.y, 8.0*stu.z, 2.0*time)));
11.     n += 0.6*(1.4-fadey)*abs(cnoise(vec4(4.0*stu.x, 3.0*stu.y, 4.0*stu.z, 1.0*time)));
12.     n += 1.8*(1.2-fadey)*abs(cnoise(vec4(2.0*stu.x, 1.5*stu.y, 2.0*stu.z, 0.2*time)));
13.     n = (1.0 - 3.0*n)*fadex*fadez;
14.
15.     vec3 color;
16.     color.r = clamp(1.0 * (3.0 * n - 0.9), 0.0, 1.0);
17.     color.g = clamp(1.0 * (3.0 * n - 1.2), 0.0, 1.0);
18.     color.b = clamp(1.0 * (3.0 * n - 1.5), 0.0, 1.0);
19.
20.     gl_FragColor = vec4(color,0.0);
21. }
```