

My Little Planet

Klas Eskilson
TNM084, Linköpings Universitet
klaes950@student.liu.se
<http://klaseskilson.se/TNM084-my-little-planet>

Sammanfattning—Denna rapport diskuterar teorin, implementationen och resultatet utav procedurella brusfunktioner i ett WebGL-program. I programmet används flera shaders skrivna i GLSL för att skapa variation och animation över ytor med goda resultat ur ett prestandaperspektiv.

I. INTRODUKTION

Både webben och datorgrafik är två områden i ständig utveckling. Starkare och mer avancerade tekniker blir tillgängliga i webbläsare, medan utvecklingen inom datorgrafen möjliggör allt mer avancerade visualiseringar och program. Kombinationen av dessa två tekniker möts i WebGL, där en förenklad version av grafikbiblioteket OpenGL är tillgänglig. Detta gör det möjligt att skapa avancerad datorgrafik i webbläsare på flera olika sorters enheter, såväl datorer som mobila enheter [1].

I programmet som tas upp i denna rapport har flera olika varianter av så kallat procedurellt brus använts. Att använda procedurella metoder för bilder innebär att använda matematiska funktioner för att exempelvis skapa en textur eller varians över en yta. Att använda procedurellt brus som i denna rapport är således att skapa variation som lätt kan uppfattas som slumpartat, när den egentligen inte är det.

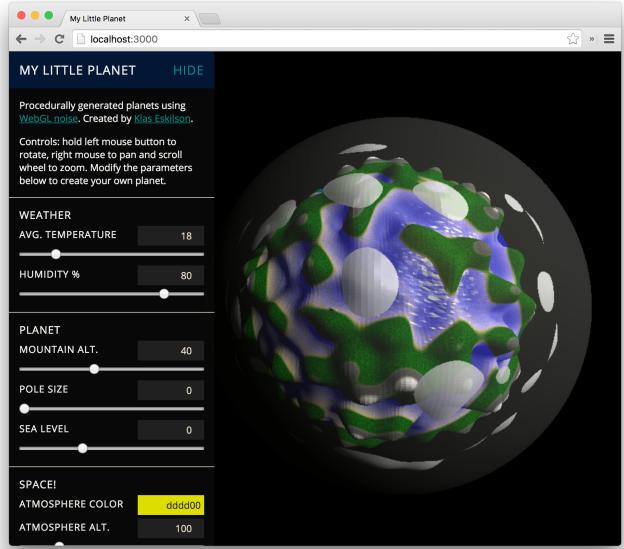
II. TEORI

Brusfunktionerna som används är dels *simplex*-brus och dels vad som kallas klassiskt Perlin-brus. Simplex-metoderna är skrivna av I. McEwan på Ashima Arts och de klassiska Perlin-brusfunktionerna är skrivna av S. Gustavsson på Linköpings Universitet.

Perlin-brus fungerar så att punkter över en linje, plan, volym eller motsvarande form beroende på antalet dimensioner som används ges en pseudo-slumpartad lutning. Punkterna mellan dessa fasta punkter interpoleras sedan mjukt. [2]

Simplex-bruset är betydligt mer avancerat, men har fördelarna att högre dimensioner inte nödvändigtvis påverkar prestandan lika negativt som dimensionsökning i Perlin-brus kan göra. För en förklaring av dessa metoder hänvisas till S. Gustavssons artikel *Simplex noise demystified* [2]. Kortfattat fungerar simplex-brus så att den enklast möjliga formen för varje dimension skapas så att den om den upprepades skulle kunna fylla hela rymden. Detta ger en kropp, eller motsvarande beroende på dimension, som har så få hörn som möjligt, $N + 1$ hörn om rymden har N dimensioner. Varje hörn påverkas sedan av, och endast av, sina närmaste grannar, vilket underlättar vid interpolationen. Detta leder till att Perlin-brus inte lider av att användas i högre dimensioner. [2]

III. IMPLEMENTATION



Figur 1: Det färdiga programmet.

Programmet är skrivet i Javascriptramverket Three.JS, vars mål är att förenkla skapandet av WebGL-program i 3D. I scenen renderas fyra sfärer för planeten; en för ytan, en för vattnet, en för molnen samt en för atmosfären. Dessa har egna *vertex-* samt *fragmentshaders*. Även en enkel sfär som representerar ljuskällan renderas.

A. Ytvariation samt -animation

För jordytan har tredimensionellt brus applicerat för att skapa toppar och dalar. Vertex-punkterna har förskjutits längs med normalen till punkten. Detta skapar en kontinuerlig förändring över ytan. För att skapa olika miljöer på planeten används denna förskjutning till att bestämma punktens höjd, varpå olika färger appliceras på olika höjd. I ordning från högst till lägst: snö, berg, skog, sand, berg. Fyrdimensionellt brus användes på vatten- samt molnytan för att låta dessa variera även över tid.

För att skapa en planetyta som ser så trovärdig ut som möjligt appliceras brus med nio oktaver utöver grundbruset. Detta gjordes i en loop på vertexshadern där brusfunktionens effekt minskade i takt med att dess variation ökade. Detta leder till att det blir variationer med olika storlekar över ytan.

Kod 1: Uträkning av höjdförändringen för planetytan (GLSL)

```

1 float elevation = 0.0;
2 for (float i = 1.0; i < 10.0; i += 1.0) {
3     elevation += (1.0 / i)
4         * snoise(i * position);
5 }
```

B. Ljussättning

Samtliga ytor gavs en Phong-reflektans med ambient, diffus och spekulär del. För att ljussättningen skulle fungera korrekt trots att vertex-punkterna förflyttats måste även normalerna uppdateras. Detta innebar ett problem då varken vertex- eller fragmentshadern inte vet om vilka punkter som ligger omkring den aktuella punkten. Varje förskjutning längs med normalen behandlas därför helt individuellt, ovetande om hur den förskjutning ställs i relation till de omkringliggande punkterna och hur detta påverkar punktens lutning. För att lösa detta användes GLSLs funktioner $dFdx$ samt $dFdy$ i fragmentshadern. Dessa funktioner kan beräkna en punkts lokala derivata i x- respektive y-led. Normalen ges sedan av kryssprodukten mellan dessa två vektorer, se ekvation 1.

$$\vec{n} = \overrightarrow{dFdx} \times \overrightarrow{dFdy} \quad (1)$$

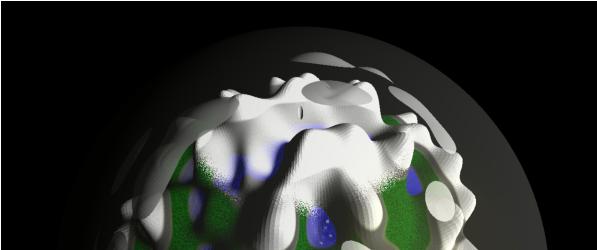
$dFdx$ och $dFdy$ fungerar som så att fragmentshadern hanterar upp bildpunkterna i grupper om fyra, så kallade *quads*, i en grupp av trådar på grafikkortet. Dessa punkter utgör tillsammans en kvadrat, och det är inom denna kvadrat som derivatorna i x- och y-led räknas ut genom att varje tråd på grafikkortet hämtar de omkringliggande trådarnas motsvarande värde. Se kod 2 för implementation. [3]

Kod 2: Uträkning av ny normal (GLSL)

```

1 vec3 dx = dFdx(position);
2 vec3 dy = dFdy(position);
3 vec3 newNormal = normalize(cross(dx, dy));
```

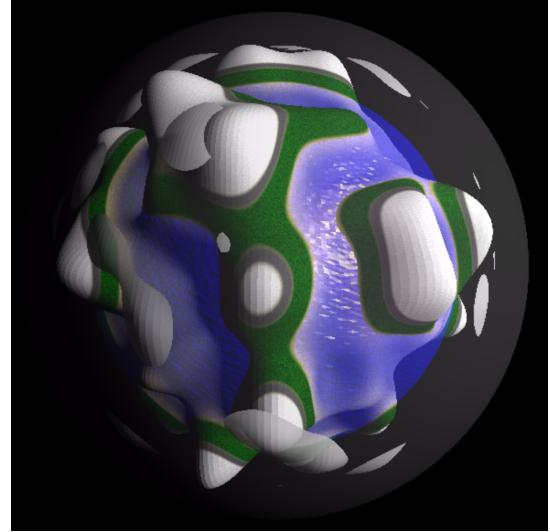
C. Gränssnitt



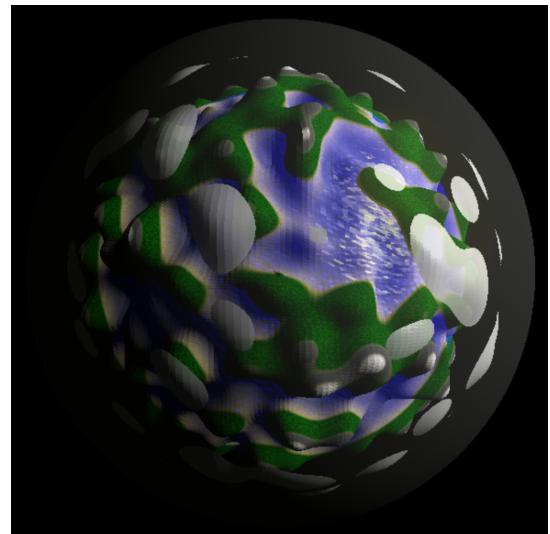
Figur 2: Detaljbild av planet med ökad polstorlek samt sänkt vattennivå.

För att göra programmet mer intressant för användaren skapades ett enkelt gränssnitt. Här kan användaren kontrollera en mängd parametrar som styr planetens utseende. Ett urval är polernas storlek, bergens höjd, molnenas täthet och atmosfärens spekulära reflektionsfärg. Det går även att styra mer abstrakta parametrar såsom medeltemperatur och luftfuktighet, vilka i sin tur styr andra parametrar. I figur 2 och 3 ses planeter

där några parametrar ändrats. I figur 4 ses en planet med standardinställningarna.



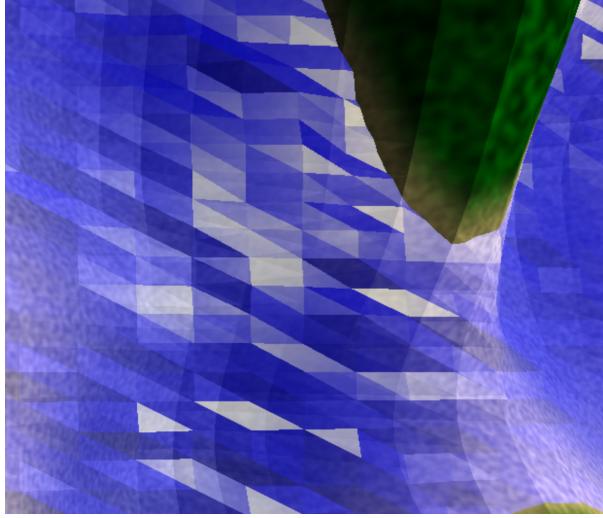
Figur 3: Planet med sänkt ytvariation samt ökad bergshöjd.



Figur 4: Planet med standardinställningar.

IV. DISKUSSION

Det är framförallt en del av programmet som bör diskuteras, nämligen reflektionsmodellen som används i kombination med brusfunktionerna planetens ytor. För att behålla en så hög prestanda som möjlig görs samtliga vertex-operationer på grafikkortet, i vertex-shadern. Detta leder till att punkternas normaler måste uppdateras i och med att punkterna förskjuts. Det är här shaderfunktionerna $dFdx$ och $dFdy$ används. Dilemmat är dock att positionerna för bildpunkterna mellan de olika vertex-punkterna interpoleras linjärt, vilket i sin tur leder till att kurvan som noise-funktionerna följer inte tas till hänsyn. Effekten av detta är att det blir tydligt synliga polygoner över ytan, se figur 5. Detta får ytan att framstå som lågupplöst.



Figur 5: Polygoner på vatten- och landyta.

En metod för att undvika detta problem är att istället för att beräkna ytorna på CPUen direkt i programmets renderingsloop, alternativt i uppstarten av programmet om ytan inte skall gå att ändra, istället för på grafikkortet. Three.js inbyggda funktion för att räkna om normalerna skulle då kunna användas. Dilemmat med detta är att det blir en potentiell prestandaförlust, framförallt då vissa ytor använder fyrdimensionellt brus för att även animeras. En annan metod är att använda samma brusfunktion som används i vertexshadern även i fragmentshadern, och för varje bildpunkt beräkna en ny normal. Detta skulle innebära att de omkringliggande punkternas nya position även de skulle behöva beräknas. Även detta är en potentiellt dyr metod ur ett prestandaperspektiv, då en fyrdimensionell brusfunktion utför ett stort antal beräkningar och det skulle krävas två extra anrop till denna funktion - en för varje förskjuten punkt.

REFERENSER

- [1] *WebGL - 3D Canvas graphics. Can I Use.* Hämtad 2016-01-27. <http://caniuse.com/#feat=webgl>
- [2] Gustavson, S. *Simplex noise demystified.* 2005, Linköping University.
- [3] Dodd, C. *Explanation of dFdx.* Stack Overflow. 2013-05-03, hämtad 2016-01-27. <http://stackoverflow.com/a/16368768>