Öppet vatten

Linus Hjeltman[1]

Sammanfattning
Enkel vågsimulator som använder sig av olika typer av brus.
Source code: `https://github.com/linushjeltman/tnm084`

Authors
[1] *Media Technology Student at Linköping University, linhj310@student.liu.se*

Keywords: WebGL, Noise, Procedural generation

Innehåll

1 **Introduktion** 1
2 **Bakgrund** 1
2.1 Perlin brus 1
2.2 Andra metoder för att skapa brus 2
Trukerad sinusvåg • Hashfunktioner i serie
3 **Implementation** 2
4 **Intressanta problem** 3
5 **Slutsatser** 4

1. Introduktion

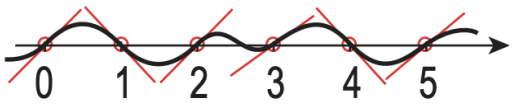
För att skapa en fiktiv värld utan att förutbestämma hur olika materian rör på sig eller ser ut så kan man använda sig av procedurella metoder. Detta ger möjligheten att generera oändligt många slumpmässiga materian som kan användas inom spelutvecking exempelvis. I detta arbete har vatten skapats med procedurella metoder i webGL med hjälp av THREE.js, i programmet finns även ett användargränssnitt där användaren kan ändra på ett antal parametrar. Då arbetet är gjort för webGL så kan det köras direkt i webläsaren.

2. Bakgrund

För att generera vågor kan man använda sig av trigonometriska funktioner såsom sinus eller cosinus. Dessa är harmoniska vågor som har en fast period vilket gör att vågformen är konstant och förutsägbar. Detta skulle inte vara en särskilt verklighetstrogen efterliknelse till vatten. För att då kunna göra vågorna oförutsägbara krävs brus, bra brus. Vad är då bra brus? Det är implementationsberoende, i detta fall önskas ett brus som är oförutsägbart men som ändå förhåller sig till intilliggande punkter som genererats. En brusmetod som uppfyller detta krav är perlin brus.

2.1 Perlin brus

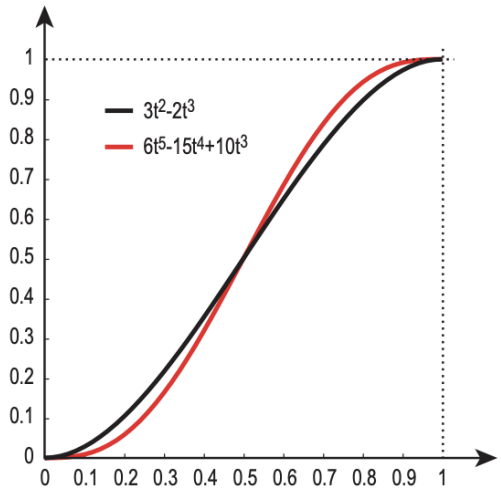
Bruset uppfanns av Ken Perlin i början av 1980-talet, det är baserat på att pseudo-slumpmässigt generera ett antal lutningar med fasta intervall som sedan interpoleras med en speciell sorts funktion så att lutningarna möts. Detta beskrivs väl av Stefan Gustavsson i sin artikel “Simplex noise demysified”. [Gus05]



Figur 1. Röda linjer: Pseudo-slumpmässiga lutningar i fasta intervall. Svart linje: Interpolerad kurva.

Vad innebär egentligen pseudo-slumpmässigt valda lutningar? Först och främst innebär det att om man matar generatorn med samma värde två gånger så ska resultatet vara samma. Detta är för att bruset inte ska bli helt oförutsägbart och försöka att efterlikna “riktig” variation. För att lyckas med detta så väljer man ett antal vektorer ur enhetssfären som pekar åt olika håll, dessa vektorer (lutningar) skickas genom en hashfunktion som sedan används som en tabell över pseudo-slumpmässigt valda lutningar, se Figur 1.

För att knyta samman de slumpmässigt valda lutningarna krävs en blandningsfunktion. Denna funktion måste erhålla kriteriet att andraderivatan måste vara kontinuerlig för att inte ge artefakter. I Perlins första iteration använde han sig av Hermites blandningsfuntion $f(t) = 3t^2 - 2t^3$, problemet med den är att andraderivatan inte blir kontinuerlig i ändpunkterna. Därför tog man fram en annan funktion som uppfyller just detta vilket blev $f(t) = 6t^5 - 15t^4 + 10t^3$, se Figur 2. Fördelarna beskrivs i Perlins “Improving noise” [Per].



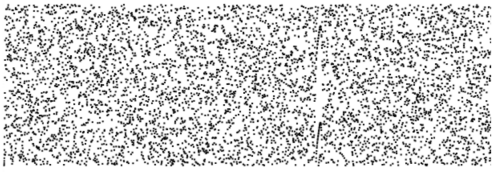
Figur 2. Röd linje: Hermites blandningsfunktion. Svart linje: Bättre variant med kontinuerliga andraderivator

2.2 Andra metoder för att skapa brus

2.2.1 Trukerad sinusvåg

Detta är ett litet trick som kan användas för att få oförutsägbart brus utan avancerade metoder. Genom att ta en harmonisk sinusvåg, ta bort heltalsdelen, multiplicera den med ett högt tal och ta absolutbeloppet av det så får man ett brus, se Figur 3 [Rag21].

```
f := frac(sin(x) * 100000);
```



Figur 3. Trick för att ta fram brus med inbyggda funktioner

2.2.2 Hashfunktioner i serie

Detta är en metod Stefan Gustavsson själv föreslagit som tidigare nämnts i denna rapport. Där tar man

$$hash(x) = (34x^2 + 10x) mod 289$$

och kör sedan $hash(hash(x)+y)$ för att få ett bra slumpmässigt genererat nummer. Fördelen med denna metod över trunkerad sinusvåg är att denna kommer generera samma resultat på alla maskiner [Rag21].

3. Implementation

Vattnets logik är implementerat i Three.js och visualiseras i webläsaren med webGL. Three.js är som ändelsen föreslår ett javascript bibliotek där man kan skapa, visa och animera 3D objekt direkt i webläsaren.

Programmet är uppbyggt i ett antal delar som listas nedan:

1. Helper functions
2. Init values
3. Canvas & camera
4. Geometry & Material
5. Lights
6. GUI
7. Rendering

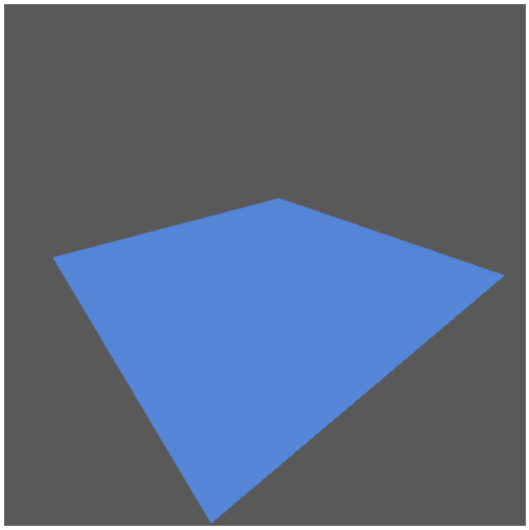
Under “Helper functions” finns funktioner till perlin bruset, blandningsfunktionen inkluderas här.

I “Init values” sätts alla initiala värden på hur primitiven ska se ut, hur frustumets sätts, ändringsbara attribut som modifieras med användargränssnittet samt definieras brusfunktionerna som implementerats.

Sedan sätts kanvasen och kameran, geometrin som snart ska bli vatten skapas och den får ett material som är beräknat med phong-shading modell, enkel ljussättning tillsätts, användargränssnittet skapas och variablernas gränser sätts och sist men inte minst kommer rendering loopen där allt kommer till liv.

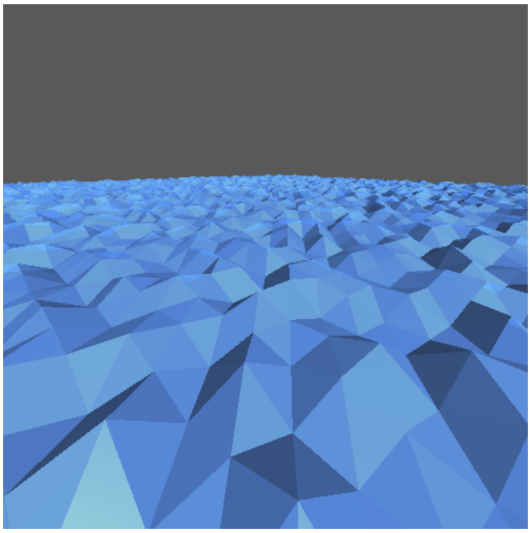
De enklare brus funktionerna är direkt implementerade genom matten som beskriver dem. Däremot så är perlin bruset kopierat och översatt från Stefan Gustavssons github [Gus16].

Det är i renderingsloopen som bruset beräknas och appliceras på primitiven som i detta fall är ett enkelt plan. Innan någon operation är applicerad ser scenen ut som i Figur 4.

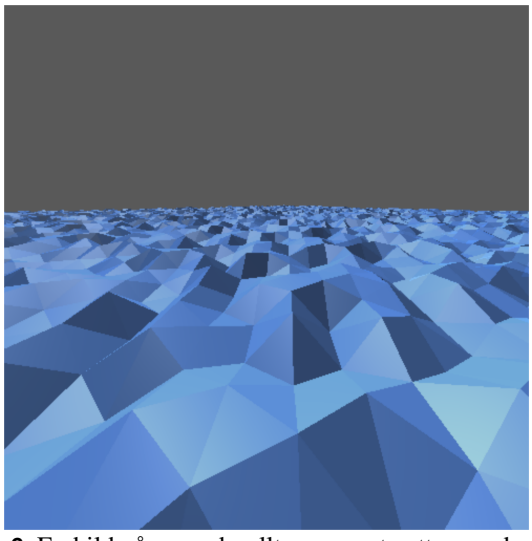


Figur 4. Enkel kanvas med ett plan

För att nu skapa vågor på denna platta tråkiga yta så appliceras ett antal sinusvågor som blir slumpmässigt matade med brus från de brusgenererande funktionerna som beskrivits ovan. Då fås följande resultat i Figur 5 och 6.



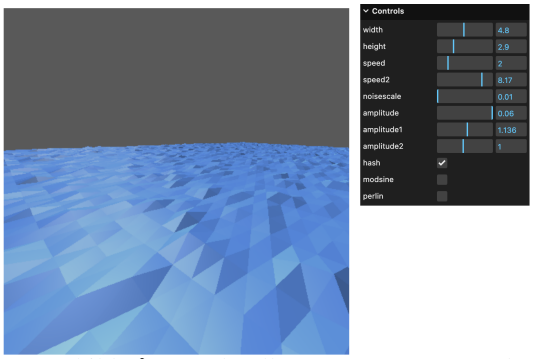
Figur 5. En bild på procedurellt genererat vatten



Figur 6. En bild på procedurellt genererat vatten med reflektioner

Med hjälp av användargränssnittet som kan ses i Figur 7 kan användaren ändra på en rad parametrar som påverkar vågornas beteende. Ännu ett exempel i Figur 8.

Figur 7. Användargränssnittet



Figur 8. En bild på procedurellt genererat vatten med reflektioner

Här är renderingsloopen, se figur 9.

Variabeln “frame” används som tidsvariabel vilket får vågorna att röra på sig, sen loopas planets punkter igenom och i “array[i+2]” ändras z-komponenten vilket är höjden för planets punkt. Den modifieras då alltså av summan av två sinusvågor vars respektive amplitud och frekvens kan ändras. Utöver det finns en global amplitud som också kan ändras men också brusfunktionerna som implementerats. Det vill säga att bruset används i vågorna istället för att direkt bestämma höjden av planet i varje punkt.

4. Intressanta problem

Ett intressant problem som jag stötte på var när jag försökte använda den inre for-loopen till att modifiera vågorna med,

```
frame += 0.01; // This is used to make things move
let i = 0;
for (let ix = 0; ix < Math.sqrt(count); ix++) {
  for (let iy = 0; iy < Math.sqrt(count); iy++) {
    mods = (modsineenabled === true ? modSine(iy * ix) : 1); // Ternary operators that enables the noise functio
    hashed = (hashenabled === true ? hash(hash(iy) + ix) * noisescale / 100 : 1);
    perlin = (perlinenabled === true ? perlinNoise(hash(hash(iy) + ix) * noisescale / 100) * 100: 1);

    array[i + 2] = amplitude * (
      (amplitude1 * Math.sin(frame * speed / 10 * (mods + hashed + perlin )) +
      amplitude2 * Math.sin((frame * speed2 / 10 * (mods + hashed + perlin))))); // Only modify the z-component

    i += 3;
    // x = i
    // y = i + 1
    // z = i + 2
  }
}
```

Figur 9. Koden som körs för att skapa vågorna

det gjorde så att förändringarna enbart skedde i den riktningen vilket skapade en i princip ren sinusvåg trotts att bruset var pålagt.

5. Slutsatser

Det som producerats i projektet är en bra grund till något som skulle kunna bli mycket bättre. Jag är relativt nöjd med vågformerna som kan produceras med nuvarande kod, däremot skulle jag vilja utnyttja mer sinusvågor för att få lite mer liv i vågorna eller producera två- eller tre-dimensionellt perlin brus som direkt kunnat styra vågorna istället för att applicera bruset inbakat i en sinusvåg. Med nuvarande kodbas räcker det modifierade sinusbruset för att skapa något trovärdiga vågor. Sedan skulle man kanske introducera någon rörelse för x- och y-led för att få vågorna att se ännu bättre ut.

Gäss hade kunnat tillföra för att göra vågorna något mer verklighetstrogna, det skulle kanske kunna göras genom att ha ett tröskelvärde för vågtoppen som gjorde att toppen skulle byta färg.

Jag själv har fått en insikt i hur perlin brus fungerar och att simplex brus (som är en förbättring av perlin brus) är att föredra då den är mer resurssnål.

Referenser

[Gus05] Stefan Gustavsson. *Simplex noise demystified*. 2005. URL: https://weber.itn.liu.se/~stegu/TNM084-2015/simplexnoise.pdf.

[Gus16] Stefan Gustavsson. *perlin-noise*. 2016. URL: https://github.com/stegu/perlin-noise.

[Rag21] Ingemar Ragnemalm. *Lecture 2*. 2021. URL: https://computer-graphics.se/TNM084/Files/pdf21/2a%20Randomness.pdf.

[Per] Ken Perlin. *Improving noise*. URL: https://mrl.cs.nyu.edu/~perlin/paper445.pdf.