# Projektrapport TNM084

Jonathan Grangien
Civilingenjör i Medieteknik
Linköping University, 2019
jonathan.grangien@gmail.com

***Sammanfattning*—Rapporten beskriver ett projekt som tillämpar procedurella metoder för bilder i en interaktiv 3D-applikation där deras generativa egenskaper uttnyttjas. Metoderna används för att generera terränggeometri och texturer i realtid. Resultatet går att interagera med i viss grad; terrängen och texturen därpå går att ändra i realtid, och användaren kan gå in i ett *spelarläge* för gå runt på terrängen. Applikationen undersöker och belyser exempel på hur procedurella metoder kan användas som alternativ till förberäknade metoder, och hur de med fördel kan konfigureras lättare i realtid.**

## I. Inledning

Procedurella metoder för bilder har fördelar i vissa användningsområden inom datorgrafik. De kan användas för att beräkna texturer och terräng i realtid, vilket sparar plats. De kan användas för att generera mycket innehåll utan — eller med minimal — manuell interaktion för att spara tid. Den kan användas för att ge användaren ett gränsnitt för att påverka genereringsalgoritmerna, för att interaktivt skapa former för bilder, geometri, eller konst.

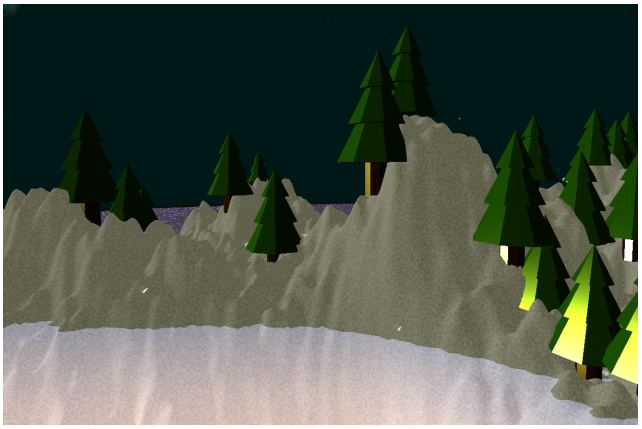
Applikationen består av en konfigurerbar 3D-miljö utvecklad med JavaScript, stödbiblioteket *three.js*, och GLSL-shaders. Projektet tillämpar procedurella metoder som ersättning för förberäknade metoder, främst genom användning av *brusfunktioner*. Brusfunktioner används för att skapa en bit terräng i fokus av applikationen, och texturer.

### A. Syfte

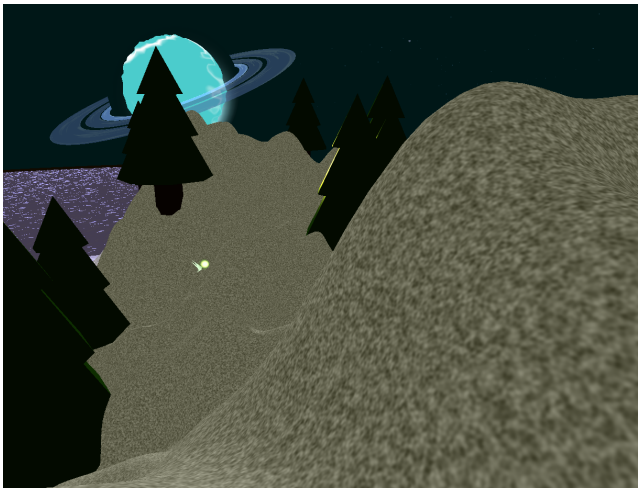
Syftet med applikationen är att utforska hur procedurella metoder kan användas interaktivt. Konfigurationsmöjligheten ska visa på hur de procedurella metoderna kan användas för att låta användaren interagera med dem och ändra miljön, samt visa hur de kan agera som rörliga och mysiga texturer. Det visuella som ska skapas är en miljö som på ett lekfullt sätt efterliknar en magisk bit natur med gräs, träd och omgivning. Användaren bör kunna ändra i miljöns utseende.

Ett mål med projektet är att användaren ska kunna, efter konfiguration av miljön, gå runt i miljön i ett förstapersonsperspektiv. Detta för att ge lite mervärde till den konfigurerade miljön; att kunna se den ur ett mer interaktivt perspektiv. Därför behövs två lägen, ett där man konfigurerar, och ett där man går runt.

Ett annat mål med applikationen är att den ska vara utbyggbar — att det ska finnas rum för många fler objekt och funktioner för framtida intresse. Detta görs genom att försöka hålla bra kodstruktur, men också genom att forma miljön i applikationen så att det finns plats för saker att lägga till, och tillämpa procedurella metoder på.



Figur 1. En bit terräng



Figur 2. Terrängens genererade bildtextur på nära håll

## II. Arbetet

Grovjobbet av projektet realiserades som terrängen i miljön, som genereras med brusfunktioner i shaders och visualiseras i JavaScript-applikationen. Applikationen är utvecklad i TypeScript, som är en variant av modern JavaScript. Valet av TypeScript motiverades av målet att försöka hålla god och påbyggbar kodstruktur. För att enklare skapa upp grundformer att applicera procedurella metoder på är grafikscenen och dess 3D-objekt definierade med grafikbiblioteket *three.js*. Shaderfilerna som används är skrivna i GLSL och applicerade på three.js-objekt. Objekten använter geometri- och material-

klasser från three.js, objekt som använder egenskrivna shaders använder three.js-klassen ShaderMaterial, som applicerar egenskrivna shaders. GLSL-filerna laddas in som strängar med hjälp av ett stödbibliotek.

Ett litet GUI-fönster implementerades med vanliga webbtekniker för att ge ett gränssnitt med konfigurationselement till användaren. GUI:t innehåller sliders och checkbox:ar för att konfigurera saker i scenen.

### A. Terrängen

Terrängens geometri genereras med hjälp av en brusfunktionen *noise3Dgrad* som bestämmer höjdvärden av en från början platt yta. Valet av noire3Dgrad motiverades av att funktionen inte bara genererar 3D-brus utan också ger tillbaka gradientvärden, som behövs här för att använda terrängen grafiskt. Medan höjdvärdet bestäms av bruset för en punkt sparas också gradientvärdet från bruset som normal för punkten. Detta görs i en shader vars utdata är höjdvärdet (flyttal) och normalen $(x, y, z)$ för varje punkt i motsvarande koordinat. Utdatan sparas ner i en textur som inte renderas ut direkt, en teknik som kallas *off-screen rendering*. För enkelhetens skull hänvisas texturen härmed till som en *FBO*, även om texturen i JavaScript-implementationen inte blir ett *framebuffer object* på samma sätt som i en OpenGL-implementation, utan en JavaScript-array av typen Float32Array.

FBO:n läses in i andra shaders, så att andra objekt i scenen kan få ut höjdvärdena och normalerna på terrängen. FBO:n läses in som textur i en shader som appliceras på ett plan (*PlaneBufferGeometry* från three.js). Vertex-shadern tar höjdvärdena från FBO:n och sätter dem som höjdvärden på geometrin. Fragment-shadern använder normalen i varje punkt för att beräkna ljus/skugga från scenens solobjekt. Fragment-shadern räknar också ut ett gräsliknande mönster med hjälp av noise3Dgrad och blandar med solens bidrag. Exempel på planets terränggeometri kan ses i figur 1 och texturen på terrängen i figur 2.

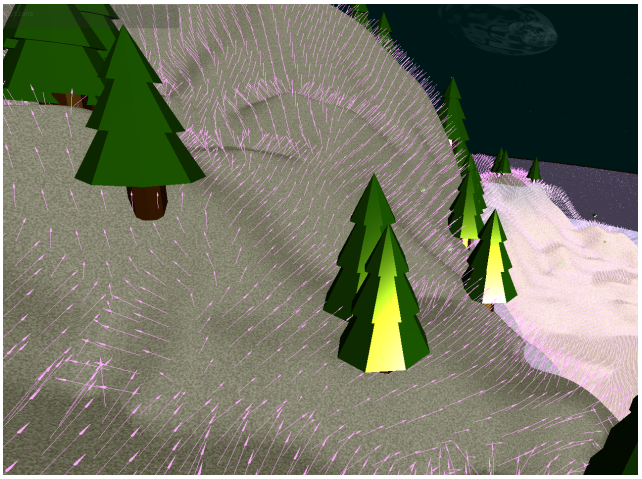
Träd-objekt och små sfärer som grovt liknar fantasivarelser placeras på terrängen. De använver sig av en funktion exponerad av terräng-klassen som ger höjdvärde $z$ ur FBO:n för $x, y$-koordinater. Varelserna rör sig runt och får ett nytt höjdvärde från FBO:n varje frame.

### B. Uniforms och andra konfigurerbara värden

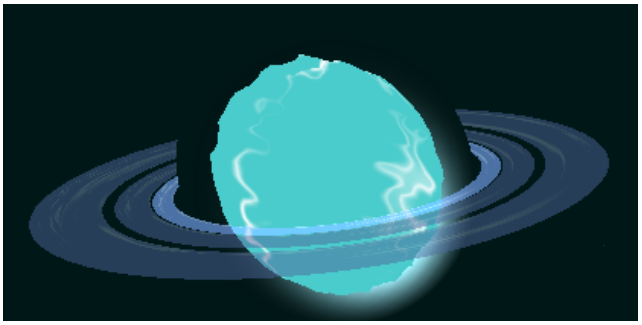
En singletonklass som innehåller applikationens alla uniforms implementerades och används överallt i applikationen. Viktigast är ett antal uniforms som används i shader-filerna som genererar terrängen och bildtexturen. Terrängens höjd och skala kan ändras, samt bildtexturens grovhet och färgstyrka. Användaren kan även slå på och av rendering av ett par objekt och funktioner, som till exempel visualisering av terrängens normaler.

### C. Visualisering av normaler

Eftersom normalerna är viktiga att få korrekta då terrängen genereras fram och har ljussättning på sig, är ett visualiseringsläge för normalerna implementerat. Om det är påslaget via GUI:t går en algoritm genom FBO:n med ett visst x- och y-interval, och för den givna punkten tar ut höjdvärde och normal, som den använder för att rita ut en pil. Pilen ritas ut med hjälp av three.js' *ArrowHelper*-klass.



Figur 3. Med visualisering av normalerna påslagen

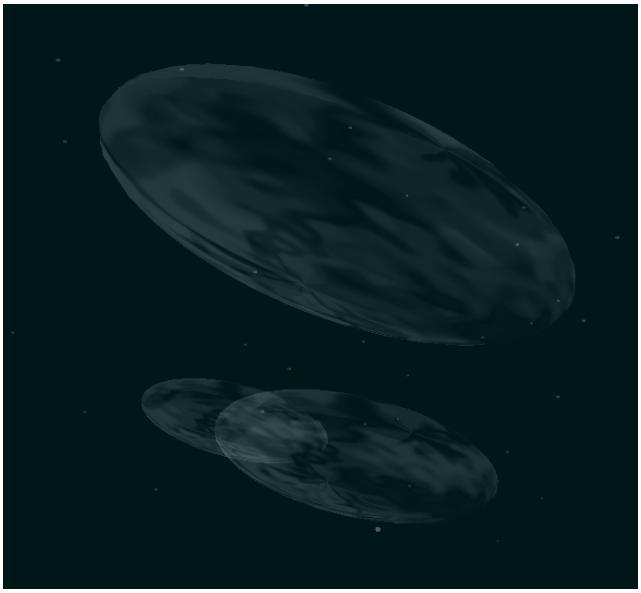


Figur 4. Sol, eller i alla fall en planet

Att se normalerna är bra för att visa på att de är korrekta, samt ge kontext till ljuset/skuggan till solen. Främst var det användbart för att under projektets utveckling inspektera dem och rätta till ett fel i shadern som förskjöt dem alla i en riktning åt sidan. De till synes korrekta normalerna kan ses i figur 3.

### D. Sol

En sol-eller-i-alla-fall-en-planet finns i scenen, se figur 4. Solen funkar som ljuskälla till scenen, men det syns mest på terrängen. Huvudsaken är att kunna använda det som rörlig ljuskälla för att visa på att ljussättningen och skuggorna på terrängen styrs av en sådan, och inte är hårdkodade. Om användaren i GUI:t klickar i att solen ska röra på sig cirkulerar den planet och ljussättningen hålls uppdaterad. Utöver fragment-shadern använder sig vertex-shadern av brus för att förflytta vertex-punkterna lite grann, för att få det att se ut lite som en ojämn, diffus yta. Vertex-shadern förflyttar punkterna med hjälp av noise3Dgrad, och fragment-shadern kombinerar noise3Dgrad och *cellular*-brus.



Figur 5. Fula moln

### *E. Moln*

Några moln som använder sig av ungefär samma fragmentshader som solen finns i scenen, se figur 5. De är en grov representation, det syns att de är ovaler med en ytlig genomskinlig textur.

### *F. Spelarläge*

Spelarläget kunde implementeras bra genom att byta mellan kamerastyrningslägen. Spelarläget aktiveras i GUI:t och samtidigt döljer det. Kameran flyttas ner till terrängen och instruktioner för hur man tittar och rör sig visas snabbt. En klass i koden byter kamerakontroller från *trackball controls* som roterar kameran runt origo, till *pointer lock controls* som roterar kameran i enlighet med datormusen och Eulerberäkningar som appliceras på quaternions. Det använder webbläsarnas inbyggda pointer lock-API, som låter applikationer ta kontroll över musen och låsa den till applikationen, som i det här fallet låter spelaren flytta kameran 360 grader utan att det tar stopp för att musen når kanten av skärmen.

När spelarläget är på får kameran (som agerar som hela spelarobjektet) varje frame terrängens höjdvärde pluss lite till i korresponderande koordinat. Kamerans position görs om från världskoordinater till koordinater i terrängobjektets lokala koordinatsystem för att stämma överens, sedan kan samma exponerade funktion som träden och fantasivarelserna använder användas.

### *G. Kodstruktur*

3D-objekten i scenen delades upp i klassfiler, ett slags objektorienterat komponentsystem. Det gjorde att de alla kunde hålla redas på i ett hashtable av komponenter till scenen för att inte blandas ihop med resten av scenens children, och tas ut en och en för att göra saker som att kalla på terrängobjektets funktion som ger tillbaka ett höjdvärde på terrängen. De har alla också en *update*-funktion som enkelt kallas på i varje frame och gör olika saker beroende på komponent.

Användandet av TypeScript underlättade strukturen genom att låta mer ordentliga klasser, interfaces och statiska typer implementeras. Biblioteket three.js har anmärkningsvärt många definitioner och typer i sin struktur och dokumentation, vilket gör TypeScript passande för det, något som många verkar tycka då stöd för TypeScript-typer har införts i bibliotekets källkod under det här projektets gång. Det säger mycket eftersom det gör bibliotekets källkod nästan dubbelt så många filer större. Komponentstrukturen underlättar läsbarheten av koden, något som har varit av personlig nytta då projektet har jobbats på med pauser av några månader emellan ett par gånger, och koden har behövts återvisiteras med färskt perspektiv några gånger.

Att shaderfilerna skrevs som GLSL-filer underlättar filstruktur och läsbarhet. De placeras tillsammans med relevant komponent.

Det webbaserade GUI:t skrevs med Preact, en mindre variant av det populära frontendbiblioteket React, tillsammans med en liknande variant av ramverket Redux. Dessa tekniker är vanliga i modern frontendutveckling av händelseberikade webbapplikationer, och även om det är lite i överdrift av ett *use case* här så underlättade det utformning av GUI:t och kommunikationen mellan det och grafikapplikationen.

## III. Diskussion

Projektet har resulterat i ett användningsområde för en del procedurellt genererat material och belyst fördelar med det i och med användarkonfiguration. En lite avancerad teknik — off-screen rendering — har tillämpats för att bepröva en bredare sorts användning: brusgenererade mönster som texturer som används som input fast även här konfigurerbart. Spelarläget kunde implementeras och bidra med ett lite lekfullt perspektiv. Projektet anses därför uppfylla målen som sattes upp. Resultatet anses inte ge mycket från forskningens synvinkel utan är mer av en lite rolig 3D-applikation som kan byggas vidare på som sidoprojekt.

Det finns med det sagt mycket som kan byggas vidare på, till exempel mer realistiskt och procedurellt genererat vatten och moln, fler saker på terrängen att interagera med och konfigurera, mer realistisk koppling mellan terräng och vatten och havsbotten.

Projektet har gett lärdom om procedurella metoder, andra metoder inom datorgrafik som off-screen rendering, och shaderprogrammering.