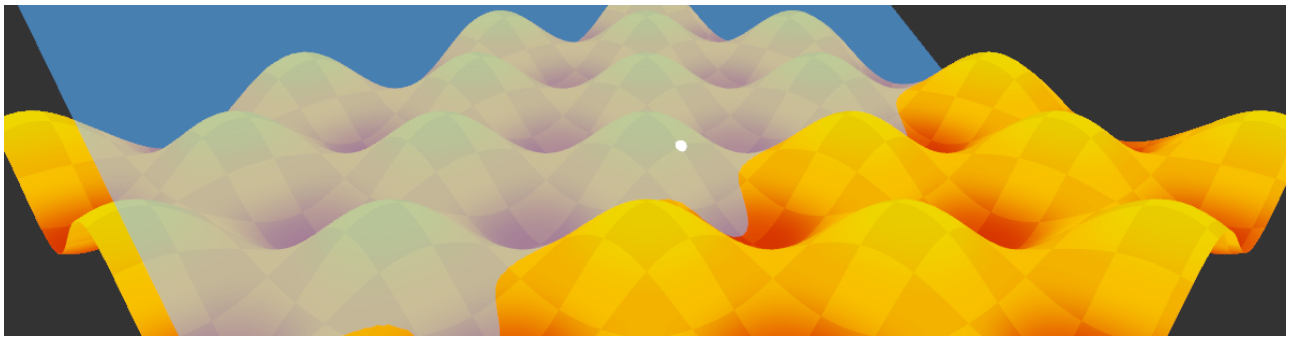


Matematisk visualisering

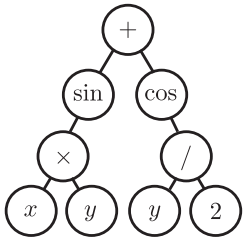
Tomas Forsyth Rosin och Emil Axelsson

I våras fick vi frågan från Olof Svensson och George Baravdish om vi ville göra interaktiva visualiseringar av innehållet i ITN:s kurser i flervariabel- och vektoranalys, TNA006 och TNA007. Eftersom analys i flera variabler lämpar sig väldigt bra att visualisera med grafik i tre dimensioner såg vi vår chans att arbeta med visualiseringsprojektet i kursen i 3D-datorgrafik.

Ett viktigt designmål har varit att skapa en så lättillgänglig 3D-applikation som möjligt. Detta i kombination med att vi båda har stort eget intresse för webben, gjorde att vi valde att genomföra projektet i WebGL. Efter att ha läst igenom några introducerande artiklar om WebGL på learningwebgl.com och efter lite eget experimenterande, insåg vi att någon form av framework med stor sannolikhet skulle göra livet enklare för oss. Vi valde att basera vårt projekt på Three.js - ett framework som tillhandahåller verktyg som en scengraf, en smidig abstraktion för att skicka data till grafikkortet samt grundläggande vektoroperationer.

Matematik

För att kunna göra lite matematisk visualisering är det bra om programmet bakom har en någorlunda vettig intern representation av matematiska uttryck.



$$\sin xy + \cos\frac{y}{2}$$

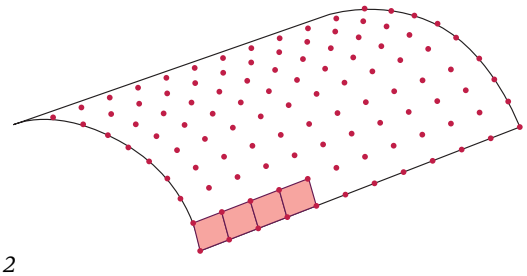
Figur 1
Uttryckträd – så här sparas information om matematiska uttryck och funktioner.

För att rita funktionsytor behöver man kunna lagra funktioner och för att rita tangentplan behöver man kunna derivera. Vi vill även kunna bilda funktioner utifrån användarens inmatning. En inte helt obetydande del av arbetet har därför, även om det inte har någon klockren koppling till 3D-datorgrafik, handlat om att jonglera med matematiska utryck. Denna del av programmet har ett gränssnitt som tillåter oss att sedan göra:

```
var expr = CALC.parse('sin(x)*cos(y)');
var dzdx = expr.differentiate('x');
```

Geometri

Vårt program behöver kunna rita upp matematiska objekt såsom kurvor, ytor och vektorer. För att rita en funktionsyta behöver vi bygga ihop en samling polygoner utifrån våra matematiska uttryckträd. Detta görs genom att beräkna funktionsvärden med jämna steg i x- och y-led. Därefter sammanfogas dessa punkter till fyrhörningar, som kan skickas till grafikkortet för rendering. Parametriska ytor kan göras på liknande sätt, men här behöver x-, y- och z-värden beräknas utifrån att stega igenom parameteriseringens definitionsmängd.

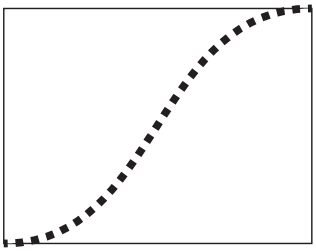


Figur 2
Tesselering av funktionsyta

För att i förlängningen kunna visualisera även vektorfält (det får bli en senare utmaning) är det bra om vi kan rita vektorpilar på ett smart sätt. Om en vektorpil byggs upp av trianglar, men ska ändra längd dynamiskt för varje frame som renderas, måste primitiverna flyttas stup i sekunden. För att avlasta CPU:n från att behöva köra onödigt mycket javascript, har vi experimenterat lite med vertex-shaders för att ändra på vektorers längd.

Animation

En viktig byggsten i våra visualiseringar är rörelse. Vi tror att mjuka rörelser då objektet man studerar roteras är ett måste för att den bortskämde betraktaren ska orka fortsätta använda vår visualisering. Jämna rörelser minimerar också risken att användaren tappar fokus från det vi strävar efter att visualisera. Vi har därför jobbat med ett verktyg vi kallar *scheduler*, som har stöd för att köa händelser på varandra såväl som att generera mjuka interpolationer mellan två lägen. En intressant frågeställning har varit huruvida schedulern ska räkna tid i sekunder eller i frames. Vi har i nuläget valt att räkna frames, vilket har fördelen att om renderingen stannar upp i en bråkdels sekund kommer ”filmvisningen” att fortsätta där den stannade. Å andra sidan riskerar förloppet att ta farligt olika lång tid på olika datorer. En möjlig medelväg vi har diskuterat är att under körningen ta fram ett medelvärde på datorns fps, och anpassa alla rörelser därefter.

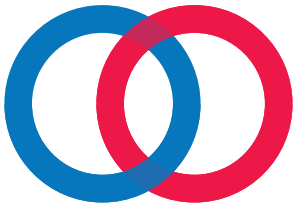


Figur 3
Schedulerns möjlighet att interpolera mellan värden med hjälp av olika ease-funktioner

Rendering

Three.js har en stor uppsättning färdiga material, som gör det enkelt att rendera ytor med exempelvis Phongshading. För att rendera funktionsytor på ett lämpligt sätt har vi dock valt att skriva en egen shader, som vi kapslat in i ett material vid namn CheckerMaterial. Den tillåter att färgen skiftar beroende på ytans z-värde, samt lägger till ett schackrutigt mönster som gör det lättare att få en uppfattning av hur lång en längdenhet är.

Om objekt dyker upp i en scen helt plötsligt utan att tonas fram, tror vi att det kan ha en viss disorienterande effekt på användaren. Därför vill vi gärna kunna ändra opaciteten hos alla våra objekt helt fritt. Här stötte vi på ett ganska stort problem, som vi bara lyckats lösa delvis. Z-buffern i OpenGL fungerar inte som vi naivt hade hoppats på. Om två fragments med alpha mindre än 1 ska renderas på samma xy-koordinat kommer den bakersta att inte renderas alls om den främre råkar renderas först. Detta beror på att z-bufferns värde uppdateras vid den första renderingen, varpå det bakre fragmentet kommer att kastas bort. Genom att alltid skicka in primitiver till vertex-shadern i en sådan ordning så att primitiver långt bort från kameran kommer först, går det att påverka ordningen på fragment-renderingen. Men hur gör man det om man vill tillåta att kameran flyttar sig hela tiden?



Figur 4
Svårt fall för en stackars fragment shader som inte vet så mycket

Problemet är för oss fortfaranade olöst om två halvtransparenta objekt ligger både framför och bakom varandra, som i figur 2. Däremot har vi löst problemet för en ensam transparent yta som skymmer delar av sig själv. Genom att vid tesseleringen av ytan generera 6 olika ordnade meshar (en från varje sida på en kub) och byta ut dessa när kameran flyttar sig kan man uppnå önskad effekt.

Interaktion

Ett viktigt designmål är att skapa ett ramverk som gör det enkelt att programmera nya specifika visualiseringar av olika matematiska begrepp. Att bygga ett ramverk för interaktion som uppmuntrar till mycket återanvändning av kod, men som ändå tillåter skräddarsytt beteende då det behövs, är för oss en stor utmaning. Ett grepp vi har tagit till för att göra systemet flexibelt är designmönstret Strategy, som tillämpas genom att vi när som helst i en visualisering kan ”ge” ett nytt verktyg åt användaren. Vänster musknapp kan i ett fall välja en punkt på en yta och i ett annat fall användas för att rotera scenen. Även om ytterst få Strategies vid det här laget är implementerade, hoppas vi att detta grepp kommer göra det enkelt för oss att fortsätta att utveckla programmet.

Andra interkationsfrågor som ligger närmare till hands i en kurs i 3d-datorgrafik är hur zoom och rotation ska fungera. Vi har valt att designa rotationsverktyget så att ”camera roll” inte är tillgängligt, så att användaren slipper hamna upp och ned när hon dessutom kämpar med att förstå vad ett tangentplan är för något. För att zooma har vi provat att använda Three.js’ kamerors *field of view*-parameter, till skillnad från att translatera kameran. En fördel är att man aldrig riskerar att ”zooma igenom” ytor, medan en nackdel är att perspektivet ser mystiskt ut då man zoomar ut väldigt mycket.