Flödesanimering i realtid med 3D-rotationsbrus

— en kort teknisk saga angående dess fasor och dess under —

Martin Estgren <mares480@student.liu.se>
Rasmus Hedin <rashe877@student.liu.se>
Alfred Rundquist <alfru536@student.liu.se>
Erik S. V. Jansson <erija578@student.liu.se>

1 Introduktion

Ett viktigt område inom *datorgrafik och visualisering* är förmågan att *simulera/animera* verklighetstrogna *flödessystem* i *realtid*. Dessa används i bl.a. spel och filmer för att enkelt skapa *visuella effekter* som t.ex. *rök- och vattenrörelser*; effekter som vanligtvis inte animeras "för hand", utan är mer lämpliga att genereras av en dator. I detta projekt implementeras ett *grundläggande flödessystem* för att *animera ett partikelsystem* i 3D med hjälp av ett så kallat *rotationsbrus* ("curl-noise"), först presenterad av *Bridson et al.* [1] 2007.

Rotationsbrus används för att framställa verklighetstrogna *partikelflöden* genom att procedurellt animera *turbulenta flöden*. Detta görs genom att beräkna rotationen av ett vektorfält som genererats av *procedurellt brus*. På så sätt skapas ett *divergensfritt vektorfält*, som påverkar partiklarnas rörelsebanor utan att de ackumuleras på punkter i fältet. Metoden är ett billigare alternativ till *Navier-Stokes ekvationer* för flöden, som är fysiskt korrekta, men eftersom det i detta fall är främst de visuella egenskaperna som är intressanta, är rotationsbrus tillräckligt bra. Dessutom kan rotationsbrus beräknas i realtid, vilket gör den lämplig för integrering i t.ex. spel och informationsvisualisering.

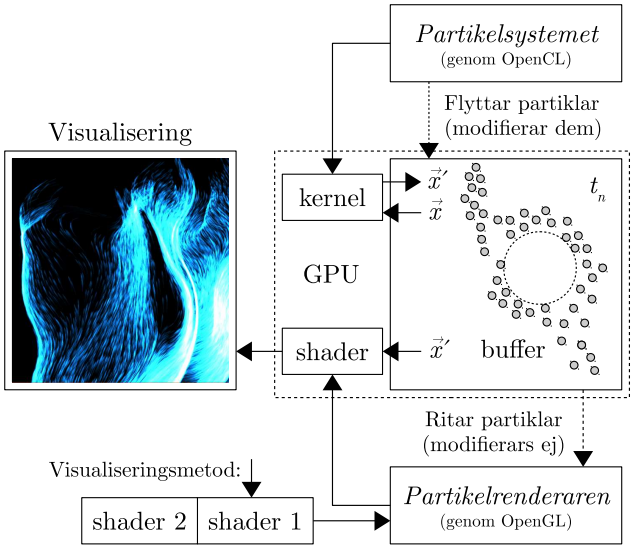
Målet med detta projekt var att skapa ett enkelt *3D-partikelsystem* där partiklarnas rörelse bestäms av ett vektorfält genererat genom *rotationsbrusmetoden* tillsammans med några enkla visuella effekter.

Arbetsprocessen gick till stor del ut på parprogrammering, och versionshantering via *Git*. För att realisera partikelsystemet användes *OpenCL* för partikelberäkningar, *OpenGL* för rendering, *GLFW* för fönsterhantering, *AntTweakbar* för "run-time" konfigurering, och *GLEW* för att hantera *OpenGL*-tillägg. Själva koden skrevs i C++ med *Premake* som byggsystem, kernels skrevs i *OpenCL* C variant, och shaders i *GLSL*.

2 Teori och metodik

2.1 Systemöversikt

Vid tidpunkten t_n befinner sig varje partikel vid sin egen position $\vec{x}$, lagrad i en *buffer*. Målet är att räkna ut partikelns nästa position $\vec{x}'$ vid tidpunkten t_{n+1} (ett steg fram i simuleringen). Detta görs genom $\vec{x}' = \vec{x} + \hat{\mathbf{V}}(\vec{x})$, där $\hat{\mathbf{V}}(\vec{x})$ är *fältriktningen*. Allt detta hanteras av *partikelsystemet*, där dessutom $\hat{\mathbf{V}}(\vec{x})$ skapas, och beskrivs under Rubrik 2.2. Detta görs parallellt på GPU:n för varje partikel under samma t_n. Vilket gör att positionerna $\vec{x}'$ kan återanvändas vid *visualisering* i *partikelrenderaren*, eftersom inget av systemen behöver lämna grafikprocessorns minnesrymd. Detta leder till god prestanda, och arbetstypen passar dessutom bra till sådana typer av processorer eftersom synkroniseringen är minimal. Se Figur 1.



Figur 1: konceptuell översikt över de två subsystemen.

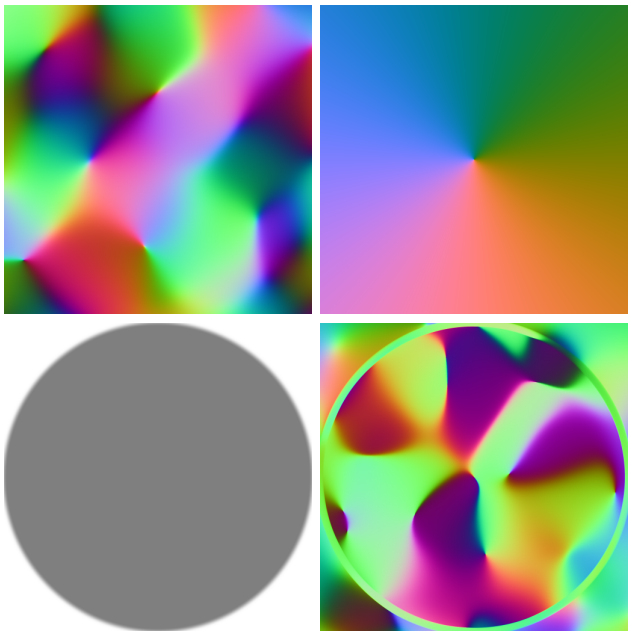
Detta görs genom att ladda upp en *kernel* (beräkningskärna) som läser in $\vec{x}$ och därefter skriver över denna med $\vec{x}'$. Genom att läsa $\vec{x}'$ från den delade buffern kan en *shader* användas för att rita partiklarna, genom att tolka hela $\vec{x}'$ som en *vertex buffer*. Man kan enkelt ändra *visualiseringsmetod* (hur man ritar $\vec{x}'$) genom att byta shader, vilket beskrivs under Rubrik 2.3.

2.2 Partikelsystemet

Partikelsystemet bygger på att varje partikels nästkommande position beräknas parallellt av en *OpenCL* kernel för varje frame.

I korthet kan man säga att tre olika vektorfält produceras och kombineras. De första två fälten representerar turbulensen och den allmänna riktningen som partiklarna ska följa. Dessa kombineras genom addition. Sedan modifieras resultatfältet så att partiklarna tar sig runt solida kroppar och slutligen appliceras en rotationsoperation, vilket resulterar i ett divergensfritt vektorfält $\hat{\mathbf{V}}$, själva riktningsfältet till partiklarna.

I originalrapporten presenteras beräkningar för både 2-D och 3-D men i detta projekt är endast det sistnämnda aktuellt. Dessutom är vissa av de beräkningar som presenteras här justerade för just detta projekt.



Figur 2: xy-plan genomskärningar av a) *turbulensfältet* b) *bakgrundsfältet* c) *rampfunktionen* d) *riktningsfältet.* (i den ordningen som visas: vänster→höger, upp→ner)

Turbulensfältet

Turbulensfältet beräknas genom att sampla en procedurell brusfunktion. Vi valde att använda *Simplex noise*[1] vilket förklaras i detalj av *Stefan Gustavsson* [2]. Brusfunktionen samplas runt varje partikels position enligt följande funktion.

$$\vec{N}(\vec{x}) = \begin{pmatrix} n((\vec{x}+\vec{\epsilon}_x)/L) \\ n((\vec{x}+\vec{\epsilon}_y)/L) \\ n((\vec{x}+\vec{\epsilon}_z)/L) \end{pmatrix} * \gamma M_n L \qquad (1)$$

Där $\vec{\epsilon}$ representerar en förskjutning så att alla vektorkomponenter inte blir korrelerade med varandra. $n(\vec{x})$ är brusfunktionen som samplas för att producera turbulens i fältet. γ representerar förhållande mellan brus och bakgrundsfält där 0 är inget brus och därmed ingen turbulens medan 1 är fullt brus utan något bakgrundsfält. M_n är styrkan på bruset (fältriktningen är normerad så vi skalar upp det till vad vi vill ha). L är längdskalan på bruset vilket i vårat fall är relativt stor (runt 20) då vi vill ha långa övergångar i bruset.

Backgrundsfält

Backgrundsfältet representerar den generella riktningen vi vill att partiklarna ska följa. Detta beskrivs inte något vidare i referensartikeln. Vi valde istället att anpassa en redan existerande implementation[2] av K. Bladin, vilket resulterar i ett bakgrundsfält som går i en uniform riktning efter att $\nabla\times$ operatorn har applicerats. $\vec{F}(\vec{x}) = (1.0-\gamma) * \vec{D}(\vec{x}) * M_f$. $\vec{D}(\vec{x})$ representerar fältriktningen i den angivna punkten och M_f är magnituden vi vill ha.

Fältriktningen är beräknad enligt följande formel $\vec{D}(\vec{x}) = \vec{x} \times \vec{p}$. Där $\vec{p}$ är fältriktningen vi vill att partiklarna ska följa.

Solida kroppar

Turbulensfältet adderas med backgrundsfältet ($\vec{\psi} = \vec{N}(\vec{x}) + \vec{F}(\vec{x})$) för att sedan justeras så att partiklarna beaktar solida sfärer utplacerade i vektorfältet.

För att få alla partiklar att respektera de utplacerade sfärerna används rampfunktionen

$$ramp(r) = \begin{cases} 1 & r > 1 \\ 6r^5 - 15r^4 + 10r^3 & 0 \leq r \leq 1 \\ 0 & r < 0 \end{cases} \qquad (2)$$

där r är distansen $\vec{x}$ till närmaste sfär delat på distansen av sfären inflytande.

Resultatet från rampfunktionen $\alpha = |ramp(d(\vec{x})/d_0)|$ där d_0 är en arbiträr skalfaktor. Slutligen beräknar vi fältet runt sfärerna genom

$$\vec{\psi}_c(\vec{x}) = \alpha * \vec{\psi}(\vec{x}) + (1.0-\alpha) * \hat{n} * \vec{\psi}(\vec{x}) \cdot \hat{n} \qquad (3)$$

där $\hat{n}$ är normalen från $\vec{x}$ till den närmsta sfären. Detta producerar en övergång från fältriktningen till en riktning tangentiell mot sfären.

Curl

Vi får fram $\nabla\vec{\psi}$ genom en *finit differensmetod* där vi samplar fältet väldigt nära punkten vi vill ha gradienten i och tar genomsnittet. Sist beräknas den slutgiltiga fältriktningen genom att applicera curl operatorn på vår framtagna gradient genom

$$\hat{\mathbf{V}}(\vec{x}) = (\nabla \times \vec{\psi}(\vec{x}))/0.0002 \qquad (4)$$

och normeras så att vi själva kan bestämma vad hastigheten för partiklarna ska vara. En genomskärning i xy-planet av dessa fält kan ses i Figur 2.

2.3 Partikelrenderaren

Då varje position $\vec{x}$ finns tillgänglig i *vertex shadern* (enligt Rubrik 2.1) kan vi ändra hur de ritas ut genom att modifiera senare *shader "pipeline" steg.* Tre olika visualiseringsmetoder har implementerats, som beskrivs individuellt i kommande delar. Dessutom har vi valt att använda *additiv färgblandning* när vi skriver till skärmbufferten, främst av estetiska skäl (områden med fler partiklar ger ett intryck att det är "ljusare"), men dessutom av rent praktiska skäl (som kommer förklaras i kommande delar). Bilder över dessa visualiseringsmetoder finns i Rubrik 3.

Punktvisualiseringen

Den enklaste typen av visualisering är att bara skapa en "punkt" där partikeln bör ligga. Här behöver vi endast en *vertex shader* och en *fragment shader.* Då användaren ska kunna observera scenen från flera håll, måste positionerna $\vec{x}$ omvandlas till det korrekta

[1] https://github.com/stegu/perlin-noise/blob/master/src/simplexnoise1234.c

[2] https://github.com/kbladin/Curl_Noise/blob/master/shaders/point_cloud_programs/update_velocities_curl_noise.frag

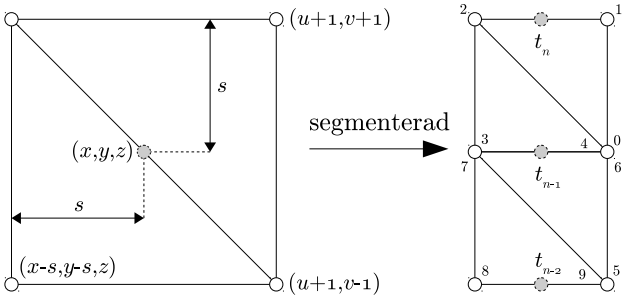
koordinatsystemet, $\mathbf{MVP}\vec{x}$ i *vertex shadern*. Därefter ger vi punkten en enkel färg genom att sätta resultatet från *fragment shadern* till $\begin{bmatrix}|x| & |y| & |z| & 1.0\end{bmatrix}$. Det vill säga, färgen av partikeln ges av dess position $\vec{x}$.

Billboardsvisualiseringen

Visuella effekter så som eld och rök kan enkelt representeras med hjälp av s.k. *"billboards"* (alt. *"impostors"*), som är ett texturerat plan som alltid är riktad mot åskådaren (vilket ger intrycket att den är 2-D). Detta görs enligt *Ingemar Ragnemalm* [4] enligt:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \ \vec{x} = \begin{bmatrix} x & y & z \end{bmatrix}.$$

Hur skapas detta plan, givet en punkt $\vec{x}$? Då vi har valt att göra allt detta i våra shaders, måste vi *skapa ny geometri* under körtid. Man gör detta med hjälp av en s.k. *geometry shader*, ett steg som ligger mellan en *vertex shader* och en *fragment shader*.



Figur 3: (a) en billboard (b) segmenterade billboards.

Som figur 3 (a) visar, skapar geometry shadern för varje punkt $\vec{x}$ (representerad som en grå cirkel i figuren), 4 nya hörn centrerade runt $\vec{x}$ (de vita cirklarna) genom `EmitVertex`. Dessa flyttats med s enheter åt respektive håll; en parameter som bestämmer hur stor planet ska vara. Man genererar primitiver (trianglar i detta fall) genom att anropa `EndPrimitive`. Sist sätter vi (u, v)-koordinaterna för de hörn vi skapat, och använder dessa i *fragment shadern* för att rita ut en 2-D textur som har "häftats" till/på vårat plan.

Vektorfältvisualiseringen

För att enklare kunna visualisera riktningsfältet har vi implementerat *"glyphs"*, som bl.a. beskrivs i *McQuinn et al.* [3]. Tekniken går ut på att "dra ut" en billboard så att den följer fältets riktning. Vi har gjort en förenklad variant, där de k senaste $\vec{x}$-värdena (vid tidsteg $t_{n-1}, ..., t_{n-k}$) lagras i en *cirkulär buffer* $\mathcal{C}$. Dessa används för att skapa en *segmenterad billboard*, där de olika segmenten skapas mellan två punkter $\vec{x}_{i-1}$, $\vec{x}_i$, där $\vec{x}_{i-1}$, $\vec{x}_i \in \mathcal{C}$. För varje segment skapas två hörn vid sidan av $\vec{x}_i$ och två hörn vid sidan av $\vec{x}_{i-1}$ enligt figur 3 (b), som sedan används för att generera primitiver för varje segment (som kopplas samman).

[3] `http://github.com/CaffeineViking/cnpf`

2.4 De övriga teknikaliteterna

Grafiska gränssnittet

AntTweakBar användes för att på ett smidigt sätt få ett grafisk gränssnitt där man kan ändra parametrar under körtid. När allt för en frame har renderats, så ritas gränssnittet ut ovanpå allt med `TwDraw`. Vi har lite olika parametrar som kan ändras för att få olika visuella effekter. Exempelvis kan man ändra riktningen på bakgrundsfältet, andel brus, och välja hur partiklarna ska visualiseras.

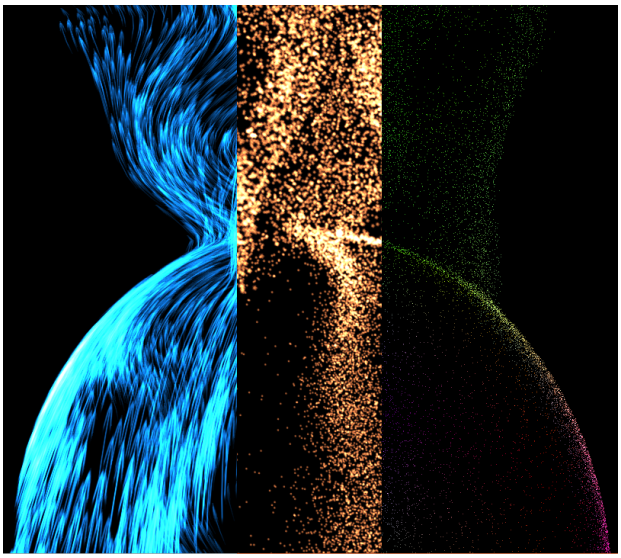
Kamerasvep

För att enkelt röra sig mellan olika perspektiv i scenen så har vi definierat "områden av intresse" där kameran bör vara (och i vilken riktning). För att få mjuka kameraövergångar interpolerades kamerans position med en *ease in/out*-funktion, med tiden som parameter.

3 Resultat & diskussion

En teknisk "demo" har byggts med metoderna som har beskrivits, och ett exempel av resultaten kan ses vid Figur 4. Våran implementation klarar av att trivialt animera och rendera 10 000 aktiva partiklar i realtid (60 FPS) med en NVIDIA GTX 980ti.

All kod[3] är open-source och delas under en permissiv licens, som kan byggas till Windows och Linux.



Figur 4: visualisering av *partikelsystemet* med tre olika metoder: *vektorfält-*, *billboard-*, och *punktvisualisering* som drivs av *partikelrenderaren*. Scenariot innehåller en solid sfärisk kropp och med en längdskala på $L \approx 20$.

3.1 Problem samt lösningar

"Svarta hål" av ej divergensfrihet

Tidigt i projektet hade vi problemet att partiklar konvergerade mot vissa punkter. Något som ej borde inträffa, då curl-noise skall vara ett divergensfritt vektorfält. Det visade sig (med examinatorns hjälp) att linjära interpoleringen som gjordes av riktningsfältet i GPU:n förstörde den egenskapen. Evaluering av riktningsfältet på GPU:n kontinuerligt löste de problemen.

Interpoleringsfel av CPU Simplex

Något liknande inträffade senare i projektet, där våra partiklar tenderade att röra sig mot en viss riktning. Till slut visade det sig att detta inträffade för att vi interpolerade simplexbruset och samplade alla komponenter på samma position. Genom att introducera $\vec{\epsilon}$ och flytta över samplingen till GPU:n så att vi inte behövde lagra det i en 3-D textur så löste sig problemet.

Konstigheter i "geometry shader"

För att få segmenterade billboards att fungera behövde partiklarnas gamla positioner samplas och skickas till en "geometry shader". En buffer med plats för alla samplade gamla positioner användes av OpenCL. Att direkt föra över delar av denna buffer till en buffer i OpenGL fungerade inte och gav bara positioner i origo. Ett nytt försök gjordes med en separat buffer (den $\mathcal{C}$ i rapporten) för varje tidpunkt. Alltså, för varje tidpunkt finns en buffer med alla partiklars gamla positioner för den tidpunkten. Detta var en lösning till problemet, och vi hade nu tillgång till alla tidigare positioner som behövdes för att skapa segmenterade billboards.

Ett annat problem som dök upp var att det saknades delar av våra billboards när de skulle ritas ut. Orsaken till problemet var att för varje segment skapades bara 4 hörn som sedan skulle bindas ihop till en quad vilket inte gav rätt resultat. Lösningen var att skapa varje quad med två polygoner manuellt (Se figur 3).

3.2 Framtida förbättringar

Segmentering med "B-splines"

Eftersom vi har en minnesbegränsning på $\mathcal{C}$ kan vi endast segmentera billboards till en viss nivå tills det börjar bli "dyrt". Här kan vi använda en *B-spline* för att interpolera mjukt t.ex. $\sum \vec{x}_i N_i^3(t)$, mellan den ändliga mängden kontrollpunkter och få "fler" $\vec{x}_i$, som kan användas för att skapa mer kurvade segment.

Filformat för "egna" scenarion

Att kunna modifiera upplägget med en egen fil (t.ex. i JSON eller XML) skulle ha gjort våra demon mer flexibla för att skapa scenarion snabbare. Exempelvis att kunna bestämma shaders, flera solida kroppar och möjligtvis omgivning (ladda in 3-D modeller).

Konkreta scenarion av shaders

Några mer "praktiskt relevanta" shaders och scenarion skulle ha varit intressanta att ha i demonstrationen. Exempelvis, att ha en gryta över en brasa som dessutom skapar lite rök. Det skulle innehålla all teknik som finns i detta projekt, dock krävs mer kod för att ladda in modeller samt sortera våra partiklar.

3.3 Projektreflektioner

Över lag är vi mycket nöjda med resultatet. Framför allt är vi nöjda med hur visualiseringen med hjälp utav segmenterade billboards och beräkningen av partikelflödet. Billboard-segmenteringen var inte något vi hade tänkt implementera förens väldigt sent i projektet men resultatet höjde slutvisualiseringen signifikant.

Partikelflödet var först tänkt att beräknas helt på CPU:n men efter de problem som vi redan nämnt bestämde vi oss för att flytta över det till GPU:n. Detta var inte en lätt sak att göra, vi hade nämligen redan implementerat en modell för vektorfällt på CPU:n som tagit åtskilliga dagar i arbete. Resultatet blev dock mycket bättre än vad vi fick från CPU implementationen både när det kommer till responstid och mängd partiklar.

Arbetsmetoden fungerade bra men parprogrammeringen led av att alla projektmedlemmar har haft olika scheman. Vilket gjort det svårt att hitta tider under vardagar där alla som jobbade tillsammans har haft tiden till det. Därför blev det ofta att alla satte sig tillsammans på helgerna och planerade vad varje person skulle få gjort under den kommande veckan. Det krävdes mycket arbete att få mjukvaran att köra på allas datorer (det är fortfarande lite problem för vissa). Det har varit en kombination av vida skilda prestanda och en blandad GNU/Linux och Windows miljö. Ett av de mer involverade delarna av projektet var att få alla bibliotek och byggsystem att fungera.

Dessutom så har det förvärvats en del nya kunskaper som vi ej hade innan projektet. Något väldigt nytt för oss alla var att arbeta med "geometry shaders", vilket vi tror kommer vara användbart i framtida projekt. I tidigare kurser så har vissa i gruppen arbetat med CUDA, men inte OpenCL (i stora mängder i alla fall). Efter detta projekt så har vi blivit mycket mer bekanta med OpenCL och dess kernels, samt den komplexa ritualen för att sätta upp OpenGL interoperabilitet.

Referenser

[1] R. Bridson, J. Houriham, and M. Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (ToG)*, 26(3):46, 2007.

[2] S. Gustavson. Simplex noise demystified. 2005.

[3] E. McQuinn, A. Chourasia, J. P. Schulze, and J.-B. Minster. Glyphsea: visualizing vector fields. In *IS&T/SPIE Dig. Imaging*. Society for Optics and Photonics, 2013.

[4] I. Ragnemalm. Polygons feel no pain. 2013.