Rymdgymnasiet

Gravitation

Och dess lag simulerad

Karl Andersson

GYMNASIEARBETE

KIRUNA 2016

Handledare: Camilla Nilsson

# Sammanfattning

# Innehållsförteckning

# Inledning

1.1 Introduktion

I ett samhälle som blir allt mer beroende av teknologin både i privatlivet som arbetslivet

Jag gör simuleringen för gymnasiearbetet med inriktningen teknik och fysik.

1.2 Syfte

Syftet med programmet är att simulera gravitationen med hjälp utav Newtons gravitationslag och sedan visualisera det på datorskärmen.

1.3 Frågeställningar

1. Går det att upptäcka några liknelser mellan simuleringen och universum?
2. Vad skulle kunna göra simuleringen bättre?

# Bakgrund

# Material

3.1 Programspråk

Ett programspråk är ett språk där man ger instruktioner till datorn med hjälp av text något som vi människor har lätt att förstå. Texten kompilerar (översätts) sedan till maskinkod (binärt), något som t.ex. en dator förstår men vi människor har svårare för, det går men är inte optimalt för effektivitet.

Som programspråk valde jag C++ p.g.a. de kunskaper jag redan har över det och att det är nära besläktad med programspråket C som brukas räknas som modern till alla de nu moderna språken. C är dessutom väldigt kraftig över hur nära den bestämmer maskinens processer och är ett utav det vanligaste språken som man använder när man programmerar en maskin, men även också datorer, fast där brukar C++ och JAVA ligga över de andra språken, vilket också är en anledning jag valt C++ då den förekommer vanligen.

3.2 IDE (Intergrated Development Environment)

IDE är där du skriver programmet som brukar bestå av en textredigerare, kompilator och debugger.

Jag valde Microsoft Visual Studio som mitt IDE då de först av allt har en gratis version. IDE:n är lätt att använda och kräver inte så mycket avancerade kunskaper för att komma igång. Nackdelen med denna IDE:n är att den inte är den bästa för multiplattform och det faktumet att man inte kan ändra kompilator.

3.3 Grafik Bibliotek

Ett grafik bibliotek är ett program bibliotek med optimerade funktioner som hjälper en att kommunicera med sitt grafikkort och på så sätt rita på sin skärm.

Jag har valt OpenGL, främst för de insamlade kunskaper jag redan har utav det men annars du det är brett stött av flera grafikkort och stödjer multiplatform. OpenGL står för Open Graphics Language, är gratis och uppdateras regelbundet via grafikkorts drivrutiner. OpenGL är ett av de största grafik biblioteken bredvid DirectX som är Microsofts eget grafik bibliotek och stödjer inte multiplatform. OpenGL har flera versioner med den senaste versionen just nu 4.5. Jag har valt versionen 3.3 då den är tillräckligt modern och stöds just nu av de flesta grafikkorten och är den version jag är mest van vid.

För att kunna programmera enklare med OpenGL har jag tagit hjälp av ett bibliotek som kallas GLEW (OpenGL Extension Wrangler Library) som hjälper en med bl.a. optimering för ens system och operativsystem.

3.4 Övriga Bibliotek

Övriga bibliotek som jag använde var GLFW och GLM. GLFW står för OpenGL Framework och är det bibliotek som läser in data från tangentbord och mus och är essentiellt om man ska kunna interagera med programmet under den tid det körs. GLFW är även ansvarig för att skapa ett fönster som är viktigt för att kunna använda OpenGL då när man ska rita så måste det ritas på ett fönster, vilket är hur operativsystem funkar (Windows).

GLM används vid lite mer avancerad matematik och används främst inom grafik programmering för dess vektorer och matriser. GLM står för OpenGL Mathematics.

Förutom de externa biblioteken som måste hämtas från nätet använder jag mig av flertalet interna bibliotek såsom <iostream> som använder kommandotolken för att ta in och ge ut data i form av text och siffror, används främst till debugging. <stdlib> som jag använde främst för funktionen rand som genererar ett slumpat tal. <ctime> för funktionen time som är ger ett värde räknat i sekunder sedan första januari 1970 UTC vilket är användbart då funktionen rand använder ett så kallat seed för att få fram sitt ”slumpade tal” och om inte seedet ändras så blir det slumpade värdet densamma varje gång då programmet körs, men om man använder time funktionen som byter värde varje sekund så kommer det slumpade värdet variera varje gång. Även <algorithm> används för algoritmer och <vector> för att lagra liknande information i flera mängder så som massa till olika objekt.

# Metod

4.1 Shader

En shader är ett program som gör skuggning och skrivs i programspråket GLSL som är redan för installerat med OpenGL. GLSL, OpenGL Shading Language, är ett språk likt C men används för att ge instruktioner till grafikkortet om hur den ska gå tillväga för att skriva på skärmen. Det finns tre olika shaders i OpenGL; Vertex, Fragment och Geometry shader, varav de två första är obligatoriska för att kunna skriva något överhuvudtaget.

OpenGL använder än fast väg för att kunna skriva ut något på skärmen som går ungefär så här: Vertex Shader -> Shape Assembly -> Geometry Shader -> Rasterization -> Fragment Shader -> Tests and Blending.

Vertex shadern kan kort beskrivas med att den tar emot en mängd angivna tre dimensionella positioner och med angivna instruktioner har möjlighet att göra något speciellt med dem men i vårt fall har vi bara skickat vidare dem till Shape Assembly som monterar ihop dem beroende på hur vi valt att de ska monteras t.ex. som trianglar. Geometry shader är frivillig men har möjligheter att kunna skapa nya former som t.ex. två trianglar från en. Efter det så rastreras formen, den med andra ord pixeleras och skickas sedan till fragment shader där vi kan föra lite allt möjligt men kan man sammanfatta att vi ger vår form en eller flera färger. Sedan testas det om vår form ska blandas med eventuella andra former om t.ex. om en är transparent och är framför den andra.

4.2 Globalt

Det globala fältet är det fält som inte involverad i någon funktion och är tillgängligt för alla inom filen.

Förutom de bibliotek som var tvungna att inkluderas, inkluderade jag en annan C++ fil vars syfte är att öppna de shader filerna som behövs och läsa in dem till vårt grundprogram. Dessutom finns det några globala variabler så som höjden och längden på fönstret och en pekare som pekar mot det fönstret som skapas senare. Det finns även en funktion som ger ett värde för den relativa mus hjulpositionen som rings varje gång den får input från användaren.

I det globala fältet finns även strukturen Point som innehåller viktig data som varje partikel kommer att erhålla. Datan innehåller tre variabler och tre vektorer: mass, radius, density, position, velocity och color.

4.3 Initialisering

Inne i main funktionen börjar programmet med att fråga användaren efter dimensionerna på fönstret och därefter vilket scenario som ska väljas. Sedan initialiseras GLFW och skapar ett fönster med den givna dimensionen, därefter initialiseras GLEW. En visuell oktogon konstrueras utav åtta trianglar med en array som ställer upp värden efter varandra, det finns alltså åtta tringlar och varje triangel har tre punkter och varje punkt har tre dimensioner vilket ger oss 72 värden radade upp efter varandra.

En Vertex Array Object genereras sedan för att samla punkternas egenskaper och även Vertex Buffer Object om de skulle bli fler. Vertex Buffer Object princip är att samla all data som vi angett i vår form i en buffer och skicka allt direkt till grafikkortet istället för en och en, vilket får programmet att gå snabbare. Sedan laddar vi shaderna vertex.glsl och fragment.glsl och deklarerar alla de variabler vi kommer att behöva innan programslingan börjar, där bl.a. Gravitationskonstanten.

Innan programslingan börjar så definierar vi de olika scenarion som ska med; solsystemet, homogenitet, slumpat och massiv.

I solsystemet använder finns solen, de officiella planeterna och vår egen måne. För att få solen på plats och planeter i omloppsbana har jag använt mig av deras perihelium som radie ifrån solen och maximum hastighet vinkelrät mot radien. Dessutom har jag använt data om massa och himlakroppens radie. I homogenitet så har alla kroppar samma massa och radie och samma avstånd på x och y ledet. Den slumpade scenariot har både massan och positionen slumpad dessutom är radien smått slumpad men anpassad efter massan. Det massiva scenariot är likt den slumpade endast det att även hastigheten är slumpad och i centrat så finns en massiv partikel med liten radie.

4.4 Programslinga

Programslingan är den del som upprepas konstant och uppdateras. Det är här partiklarna jämförs med varandra och ritas ut på skärmen. Det är även här som tangent och mus input beräknas.

Slingan börjar med att uppdatera tiden för att senare kunna räkna ut skillnaden i tiden under slingans gång. Sedan kontrolleras om vänstra musknappen trycks ner och ritar ut en linje efter den relativa positionen från där musen först klickades. När väl knappen släpps skapas en partikel som har en riktning efter den utritade linjen och en hastighet baserad på dess längd. Massan får ett konstant värde skriven i programkoden.

Partikelns färg har fyra lägen att baseras på, antingen dess sanna färg som anges när partikeln skapas, eller färgen baserad på massan, färgen baserad på hastighet, eller färgen baserad på accelerationen. Vilken färg partikeln ska ha skiftar mellan dessa fyra lägen varje gång tangenten u trycks ner.

Skärmen flyttas även runt om W, A, S, D trycks ner för att kunna navigera runt i den två dimensionella rymden, skärmen skalas även till svar att tangenterna R och F trycks ner. Skrollar man med mushjulet så ökar man tidens hastighet eller sänker den och trycker man på mellanslagstangenten pausas tiden och alla partiklar står stilla. En kvadrat i nedersta högra hörnet finns för att indikera vilken hastighet tiden har, den är bl.a. röd om tiden står still.

Huvuddelen i programmet är när alla de befintliga partiklarna jämförs med varandra och beräknar kraften. Jämförelsen börjar med en for loop som börjar med första talet i vektorn och upprepar till varje partikel har jämförts. I loopen finns åtta variabler definierade: acceleration i x och y ledet, distans i båda leden och även distansen övergripet och kraften i x och y och även kraften övergripet. Vid varje iteration nollställs dessa variabler då de är i jämförelse syfte och personliga till varje partikel. Den givna partikeln ska jämföras med de övriga partiklarna i en ytterligare for loop och börjar med att räkna differensen i distansen i de två givna partiklarna i x och y led. Sedan räknas den övergripande distansen med hjälp av Pythagoras sats. Den övergripande kraften beräknas med hjälp av Newtons gravitationslag där m1 och m2 motsvarar massan för de två jämförande partiklarna och r motsvarar distansen mellan dem. G motsvarar gravitationskonstanten och definierades vid initialiseringen med värdet . Kraften bryts sedan ner i x och y led genom att multiplicera kraften med sinus respektive cosinus som går att definieras genom att dividera distansen på x led med den övergripande distansen och densamma för y ledet. För varje jämförelse adderas den nedbrutna kraften för att räkna den totala kraften för partikeln i jämförelse.

Ifall en partikel har kommit i kontakt med en annan partikel i jämförelse fasen eller ifall partikeln förutspås komma i kontakt med en partikel under delta tiden så sammanfogas partiklarna. Partikeln som har den största massan kommer att erhålla den andra partikelns värden så som massa, radie, position och hastighet samtidigt som partikeln annihileras. Massan summeras och den nya radien räknas ut genom att dividera den nya massan med partikelns densitet för att få arean och sedan delas med pi och drar roten ur för att får radien, . Den nya positionen räknas ut genom att subtrahera halva differensen i x och y led i proportion till de båda partiklarna radier till den tidigare positionen. Därefter beräknas den nya hastigheten genom att addera rörelsemängden på den andra partikeln dividerat med massan på den dominerande partikeln, , vilket görs både i x och y led.

Efter jämförelse processen ska den adderade kraften i x och y led omvandlas till acceleration på partikeln vilket görs genom att dividera med partikelns massa, , därefter omvandlas accelerationen till hastighet som fås fram genom att dividera accelerationen med deltatiden som räknas ut varje iteration, . Slutligen omvandlas hastigheten till förändring i distans genom att ytterligare dividera med deltatiden och adderas sedan till partikelns helhetliga position, . Därefter översätts positionen in i oktogonens alla punkter med hänsyn till skalan, partikelns radie och kamerans position.

Efter all omvandling och översättning är partikeln redo att skrivas ut på skärmen som görs genom att skicka in oktogonens uppställning av tal till OpenGL som skickar den sedan till vertex shadern och vidare till fragment shadern där vi har skickat in en färg beroende på vilket läge vi låter färgen basera sig på. Efter det är hela partikelns process genomförd och upprepas nu för de andra partiklarna.

Efter att den sista partikeln har gått igenom sin process så läggs en simpel effekt till som gör att partikeln får ett kort spår efter sin bana. Eftersom skärmen kräver ett kommando för att rensas så ritas allting på samma skärm men om man ignorerar kommandot och istället fyller hela skärmen med bakgrundsfärgen med ett givet alpha (genomskinlighet) värde så kommer partikelns förra position tona ut för varje iteration och skapar ett spår av sin bana. Sen vid slutet av programslingan så uppdateras deltatiden efter hur lång tid det tog att utföra hela iterationen, och efter det upprepas programslingan om igen tills programmet stängs ner genom att någon trycker ner mellanslagknappen.

# Resultat

5.1 Solsystemet

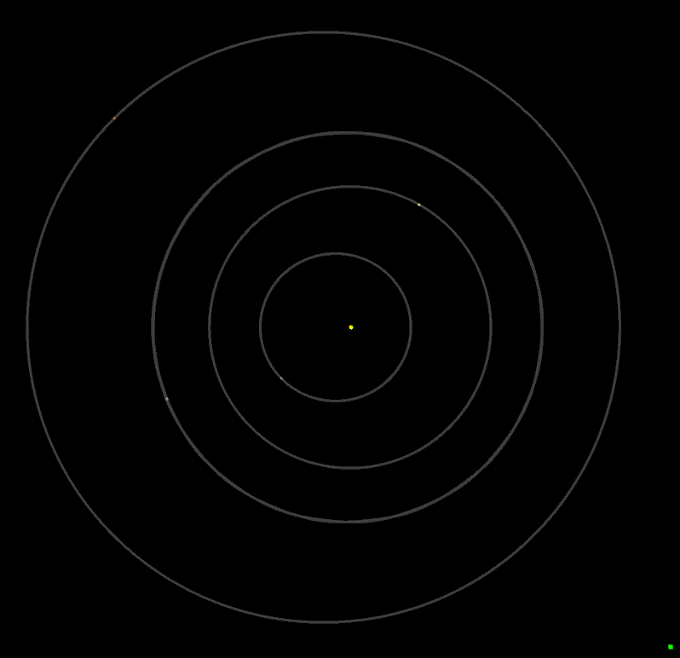
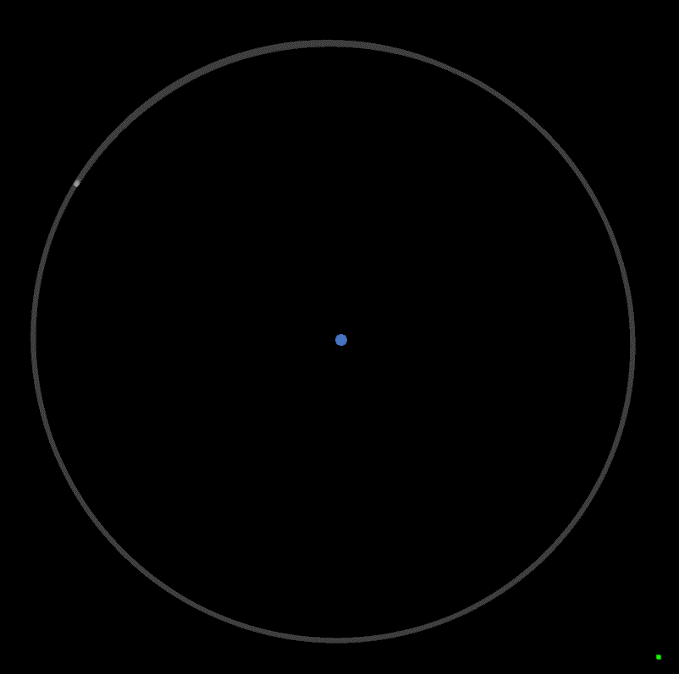
I scenariot där solsystemet simuleras i utgång av planeternas; radie, massa, perihelium och max hastighet uppstår en tydlig omloppsbana där ju närmare planeten befinner desto kortare varar omloppstiden.

Den allra innersta planeten, Merkurius, har en någorlunda oval omloppsbana med aphelium ca 1,5 gånger så stor som perihelium. Omloppstiden tog 1 minut 16 sekunder i simulerad tid vilket med deltatiden på 100 000 sekunder per sekund får den verkliga omloppstiden 87,962 dygn. Enligt NSSDCA har Merkurius en omloppsperiod på 87,969 dygn och förhållandet 1,518 mellan aphelium och perihelium.

Planeten efter Merkurius, Venus, har en mer cirkulär i jämförelse med Merkurius och tar 3 minuter och 14 sekunder för att göra ett varv i simulerad tid med samma deltatid. Översatt till real tid tar det ca 224,537 dygn. Venus tar egentligen 224,701 dygn och aphelium är nära 1,01 större än perihelium.

För Jorden tar det i simulerad tid 5 minuter och 16 sekunder att göra en revolution runt solen vilket motsvarar 224,741 i realtid. Jorden har också en någorlunda cirkulär omloppsbana. Jordens naturliga satellit Månen har en omloppsbana runt Jorden som tar 24 sekunder simulerat, 27,778 i realtid, och med ögonmått är banan väldig cirkulär. I verkligheten tar det Jorden 365,256 att fullborda ett varv runt solen och ett förhållande mellan aphelium och perihelium på 1,03. Månen tar egentligen 27,322 dygn för att göra ett varv runt jorden och har apogee, perigee förhållande på 1,12.

Mars har en simulerad omloppstid på 9 minuter 53 sekunder vilket motsvarar 686,342 dygn i verklig tid. Mars förhållande mellan aphelium och perihelium i simuleringen motsvarar ungefär 1,25 och har alltså en mer oval omloppsbana. Mars har i verkligheten en omloppstid på 686,980 dygn och förhållandet mellan aphelium och perihelium på 1,21.

*Figur 1. De inre planeternas omloppsbanor runt solen i simuleringen*

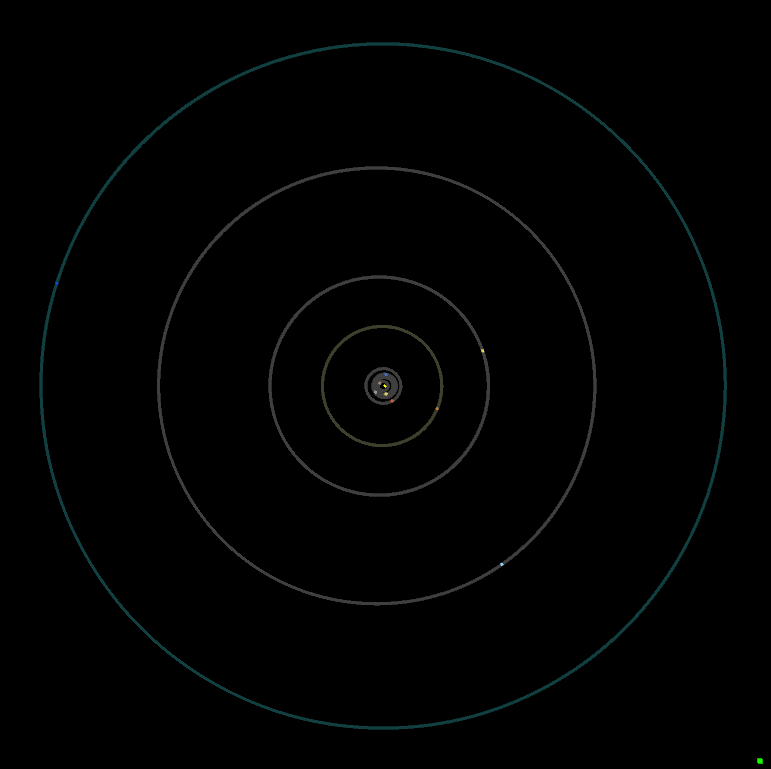
*Figur 2. Månen i omloppsbana runt Jorden i simuleringen*

De yttre planeterna har betydligt längre omloppstid vilket även gäller den simulerade tiden så för att förkorta den simulerade tiden så ökas deltatidens värde till 10 000 000 sekunder per sekund. Under förloppet av simuleringen med den givna deltatiden lämnade Månen omloppsbana runt Jorden.

I simuleringen har Jupiter en omloppstid på 37 sekunder vilket motsvarar 4 282,407 dygn eller 11,733 år i realtid. Jupiter har även i simuleringen en cirkulär omloppsbana. Det tar 4 332,589 dygn enligt NSSDCA eller 11,862 år dessutom har aphelium och perihelium förhållandet ett värde på 1,1.

Saturnus simulerat har en omloppstid på 1 minut och 33 sekunder som motsvarar 10 763,89 dygn eller 29,490 år och en någorlunda cirkulär omloppsbana. I verkligheten har Saturnus en omloppstid på 10 759,22 dygn (29,457 år) och förhållande mellan aphelium och perihelium på 1,12.

Både Uranus och Neptunus har något cirkulära banor, förhållandet mellan aphelium och perihelium är 1,1 resp. 1,02 enligt NSSDCA. Omloppstiden för Uranus är 30 685,4 dygn (84,011 år) och 60,189 dygn (164,79 år) för Neptunus. I simuleringen tar det 4 minuter 22 sekunder för Uranus att slutföra ett varv runt solen medan det tar 8 minuter och 35 sekunder för Neptunus. Båda planeterna har en någorlunda cirkulär omloppsbana. Simuleringstiden översatt till realtid motsvarar 30 324 dygn (83,08 år) resp. 59 606 dygn (163,305 år).

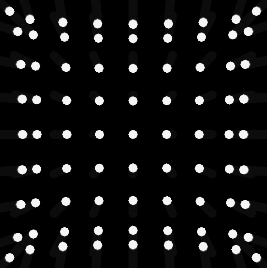
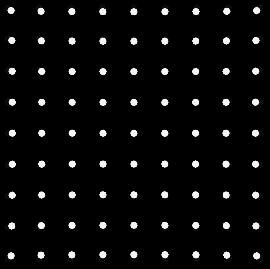
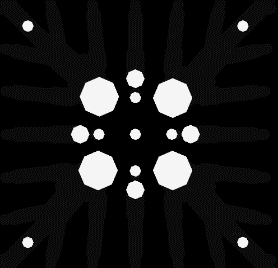
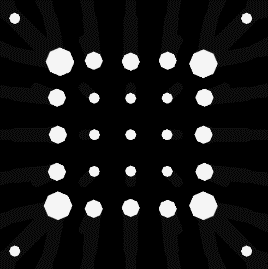
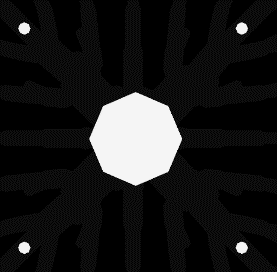


*Figur 3. De inre och yttre planeternas omloppsbanor runt solen i simuleringen varav Månen lämnat Jorden*

5.2 Homogenitet

I det homogenetiska scenariot finns 81 partiklar där de är anordnade så att distansen mellan partiklarna i x- och y-led är densamma och skapar ett rutnät så att det finns 9 kolumner och 9 rader. Partiklarna har en massa på 1,0 kg och avstånd mellan varandra på 0,25 m varav partiklarnas grundradie är 0,025 m.

Partiklarna kollapsar inåt mot centern i en astroid form, där de centrala partiklarna vid kanten accelereras snabbare än de partiklar närmare hörnen. Ju längre ut mot kanten ju snabbare accelereras partikeln och center partikeln står still. Alla partiklar förutom de yttre hörnpartiklarna accelereras inåt och kombineras där formen förblir kvadratlik t.o.m. de inre hörnpartiklarna knappar in och får en cirkulär form tills de har kombinerats till en partikel.



*Figur 4. Utvecklingsprocessen för det homogenetiska scenariot där partiklarnas radie är 0,0025 m och massan 1,0 kg*

5.3 Slumpat

5.2 Massivt

# Diskussion och Slutsatser

# Källförteckning

http://www.learnopengl.com/

http://nssdc.gsfc.nasa.gov/planetary/factsheet/