Projektet

Tillämpad realtids-fysik FY1426

Karlos Napa Häger - Kahg17

Introduktion

Det här projektet i fysik är en simulering av solsystem och planeter med kraftfält och använder två steg metoder. Simulering är uppbyggd i Unity Game engine i 3D och steg-metoderna semi-implicit Euler och Runge-Kutta 4 används i projektet.
Målet är att få objekt att påverka varandra med kraftfält och dra varandra mot sig själva med deras massa och få dem att röra sig i en omloppsbana.

Modell

Formlerna som används för simuleringen är Kraftfälts-formeln, Newtons andra lag och accelerations-formeln, och nedan så beskrivs de hur de används i simulationen.
För att räkna ut kraften så påverkar planeterna så jämförs alla planeter med varandra.
Kraften som påverkar planeten ger oss acceleration och sedan hastighet efter timestep.

Kraftfält

$$F = G \cdot \frac{m_1 \cdot m_2}{r^2}$$

F är force, G är gravitationskonstanten, m_1 och m_2 är två olika planeter som jämförs och r är distansen mellan kropparna.

Newtons andra lag

$$F = m \cdot a$$

$$a = \frac{F}{m}$$

Accelerationen betecknas med a och den får vi med kraften som påverkar planeten dividerat med planetens massa.

Accelerations formeln

$$a = \frac{v_1 - v_0}{t}$$

$$a \cdot t = v_1 - v_0$$

$$v_1 = v_0 + a \cdot t$$

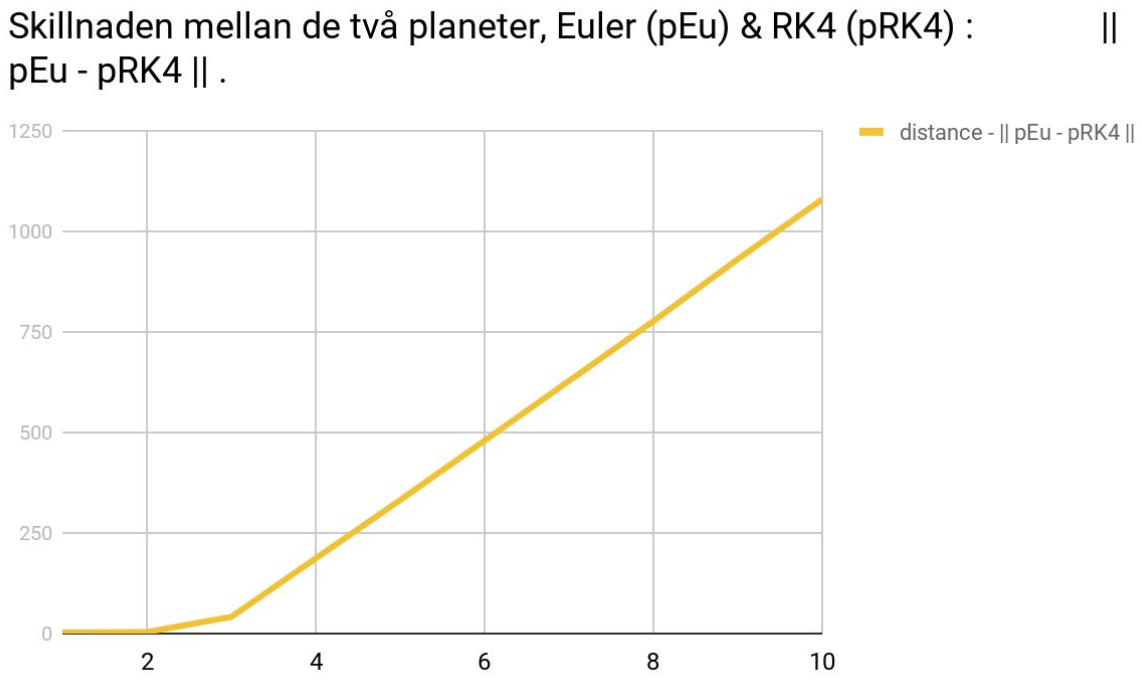
För att få ut hur mycket en planet rör sig används accelerations formeln med den data vi har.
Efter att ha skrivit om formeln så kan den representeras som $v \mathrel{+}= a * t$ i skriptet.

Steg-metoder

Steg-metoderna Semi-implicit Euler och Runge-Kutta 4 används också för att räkna ut fysiken i simuleringen och beskrivs och jämförs nedan, implementationen finns under nästa avsnitt

Enligt Gaffer on Games så är Semi-implicit Euler är en simplare stegmetod än Runge-Kutta 4 och räknar stegvis ut positionen med första ordningen av integration. Medans Runge-Kutta 4 räknas med fjärde ordningens integration, vilket gör att felen som uppstår hamnar i den fjärde derivatan, vilket gör den mycket mer noggrann.

Semi-implicit Euler är bara noggrann när accelerationen är konstant, men eftersom accelerationen inte är konstant och varierar under programkörningen så kommer en viss error uppstå. Som diagrammet visar så avviker Semi-Implicit Euler från Runge-Kutta 4.



Då jag inte kan köra båda steg-metoderna samtidigt så kan diagrammet jämföras mellan semi-implicit Euler och Runge-Kutta 4.
Diagrammet visar 50 sekunder in på körningen, och visar data från Semi-Implicit Euler och Runge-Kutta 4 under 10 sekunder och följer två specifika planet kroppar, med en tyngre planet som drar till sig alla planeter och slungar sedan iväg den mindre planeten.
Diagram är distansen mellan två planeterna och visar skillnaden mellan Euler (pEu) och Runge Kutta 4 (pRK 4) efter viss tid, som räknas i excel med distans formeln.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Jag valde 50 sekunder in på körningen eftersom det är då planeten hamnar så nära den tyngsta planeten att den tvingas in i en kraftig omloppsbana och slungas ut, så accelerationen som tillkommer hade jag redan förutspått att den skulle vara som mest intressant för steg-metoderna att räkna ut.
Och som diagrammet visar så avviker semi-implicit Euler från Runge-Kutta 4 efter lite mer än 2 sekunder, den verkar inte nå samma hastighet eller acceleration under den tid då kraften påverkar planeten som mest och får inte samma vinkel när den slungas iväg.
För övrigt så håller Semi-Implicit Euler och Runge-Kutta 4 jämnt intill varandra innan de avviker, men den stora skillnaden är där den tyngre planeten påverkar den planet som vi följer och krafterna som påverkar den får den att slungas iväg med höga accelerations skillnader och Semi-Implicit Euler inte hänger med att räkna korrekt.

Implementation

Fysiken är skriven som script i C# i Unity Game Engine, alla planet object är bara gameobjects som varje har en transform, mesh filter, mesh-renderer samt ett script, fysiken är implementerad så att det lätt går att växla mellan steg-metoderna.
Scriptet är kodat så att alla objekt kollar och jämför med varandra och påverkar varandra med sina egenskaper såsom massa och position.

Efter att kollat igenom guiden på Gaffer On Games och på hur de implementerade semi-implicit Euler och Runge-Kutta 4 så blev min implementation enligt bilderna nedan i skript-delen.

Skript

I Gaffer On Games använder de sig av 2 structs för att hålla koll på position och velocity i en och derivative variabler i en struct, men jag använder bara en för derivative eftersom position tillhör game-objektet och jag kan ha velocity utanför struct med massan och accelerationen.

```
4   struct Derivative
5   {
6       public Vector3 dpos;//delta_position;
7       public Vector3 dv;//delta_velocity;
8   }
    14 references
9   public class Attractor : MonoBehaviour
10  {
11      const float G = 0.6674f; //Gravitational constant 6.674×10^(-11)
12      public static List<Attractor> Attractors;
13      public float mass;
14      public Vector3 velocity;
15      public Vector3 acceleration;
```

Gravitationskonstanten är egentligen $6.674 \cdot 10^{-11}$ men jag har för simpel hetens skull använt mig av att sätta gravitationskonstanten multiplicerad med 10^{10}, alltså 0.667.

I Funktionen “formula()” simuleras semi-implicit och i funktionerna “accelerationRK”, “evaluate()” och “integrate()” så simuleras fysiken enligt Runge-Kutta 4.

```
64    void formula(Attractor objToAttract)
65    {
66        Vector3 dir = this.transform.position - objToAttract.transform.position;
67        float dist = dir.magnitude;
68        float absorb = Planet_Tracker.MyInstance.absorb_radius;
69        if (dist >= absorb)
70        {
71            //f=G*(m1*m2)/(r^2)
72            float forceMagnitude = G * (this.mass * objToAttract.mass) / (dist * dist);
73            Vector3 force = dir.normalized * forceMagnitude;
74
75            //f=ma  ---> a=f/m
76            acceleration = force / mass;
77
78            //a= v1-v0/t  --->  a*t= v1-v0  --->  v1=v0 + at ----> v += at
79            this.velocity += acceleration * Time.deltaTime;
80            this.transform.position -= velocity * Time.deltaTime;
81        }
82        else
83            checkMasses(objToAttract);
84    }
```

```
85    Vector3 accelerationRK(Vector3 dir, float dist, Attractor objToAttract) // gravity here
86    {
87        //f=G*(m1*m2)/(r^2)
88        float forceMagnitude = G * (this.mass * objToAttract.mass) / (dist * dist);
89        Vector3 force = dir.normalized * forceMagnitude;
90
91        //f=ma  ---> a=f/m
92        acceleration = force / mass;
93        return acceleration;
94    }
```

```
108   void integrate(float dt, Attractor objToAttract)
109   {
110       Derivative a, b, c, d;
111       Vector3 dir = this.transform.position - objToAttract.transform.position;
112       float dist = dir.magnitude;
113       float absorb = Planet_Tracker.MyInstance.absorb_radius;
114       if (dist >= absorb)
115       {
116           a = evaluate(dir, dist, 0.0f, new Derivative(), objToAttract);
117           b = evaluate(dir, dist, dt * 0.5f, a, objToAttract);
118           c = evaluate(dir, dist, dt * 0.5f, b, objToAttract);
119           d = evaluate(dir, dist, dt, c, objToAttract);
120
121           Vector3 dxdt = 1.0f / 6.0f * (a.dpos + 2.0f * (b.dpos + c.dpos) + d.dpos);
122           Vector3 dvdt = 1.0f / 6.0f * (a.dv + 2.0f * (b.dv + c.dv) + d.dv);
123
124           //a= v1-v0/t  --->  a*t= v1-v0  --->  v1=v0 + at ----> v += at
125           this.velocity += dvdt * dt;
126           this.transform.position -= dxdt * dt; //minus here
127       }
128       else
129           checkMasses(objToAttract);
130   }
131   }
```

```
95      Derivative evaluate(Vector3 dir, float dist, float dt, Derivative d, Attractor objToAttract)
96      {
97          //a= v1-v0/t  --->  a*t= v1-v0  --->  v1=v0 + at ----> v += at
98          Vector3 tempVelocity = this.velocity + d.dv * dt;
99          Vector3 tempPosition = this.transform.position - d.dpos * dt;
100
101         Derivative output;
102         output.dpos = tempVelocity;
103         output.dv = accelerationRK(dir, dist, objToAttract);
104
105         return output;
106     }
```

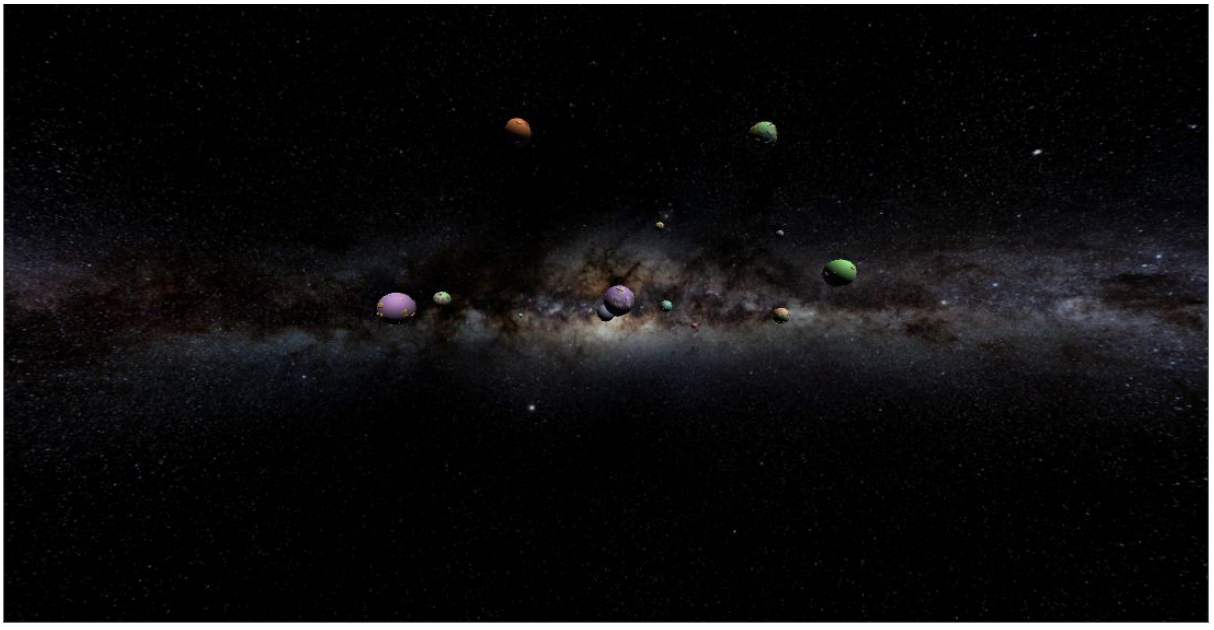
Validering

För att se om simulering är korrekt så måste alla planeter observeras, och olika massa sätts på alla planeter så att de verkar unika. Kommer de ens påverka varandra? Hur påverkar de varandra? Rör de sig i en bana som liknar ett solsystem?

Resultat

När simulering startas kan alla planeter observeras där deras kraftfält påverkar varandra. Då Kraftfält inte syns grafiskt så syns det däremot att tyngre planeter har starkare kraft påverkan på varandra.

När planeter kommer för nära varandra så absorberar planeterna varandra så att den tyngre blir större och får vikten av den andra men hastigheten är fortfarande kvar så att just absorberingen är inte helt verklighetstrogen utan mer en extra effekt.



Slutsats

Efter att ha kollat på en youtube video, (How to Simulate Gravity in Unity) och läst igenom Gaffer On Games, (Integration Basics) rapporten från Tomas så tycker jag att jag fick en bra instruktioner med grund för att bygga simulationen med solsystemets fysik och kraftfält.

En Bonus funktion finns också med i simuleringen som nämndes tidigare där alla planeter som kommer för nära varandra blir absorberade av den tyngre planeten, och om de har samma vikt så kollas istället namn värdet som integer för att se text som nummervärde.
Det har varit en lärorik och kul uppgift att bygga upp simuleringen.