EKSAMEN FYSIKK SPILL

Torkil Wathne Svinø

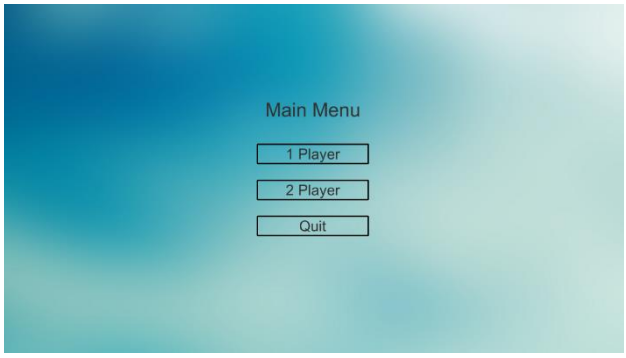
DFSM3101 Simulering og modellering

Innhold

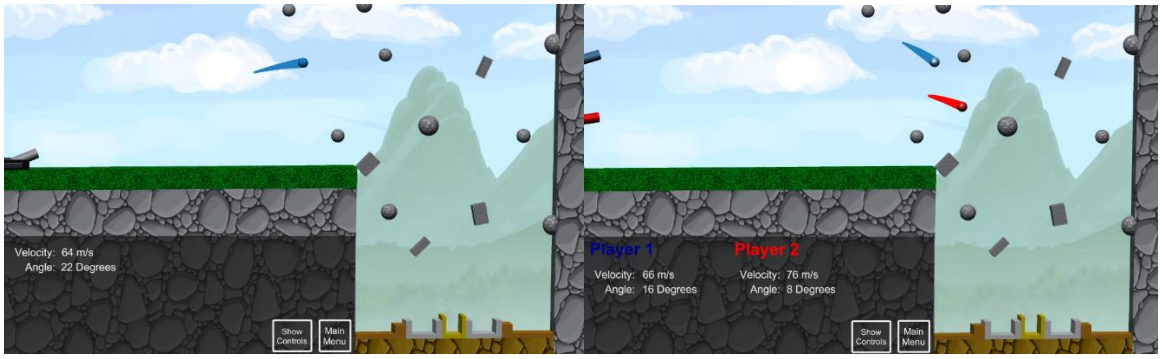
1. Introduksjon

Inspirasjonen for dette spillet kommer fra et brett spill jeg spilte da jeg var yngre, hvor målet var å sende av gårde en kule på et skråplan for så å få den til å lande i en kurv med mest mulig verdi.

Spillet jeg har laget har et 2.5D perspektiv med gravitasjon, luftmotstand og kollisjon. Hvor målet er å avfyre en kanon med retning og fart til å lande i kurven med høyest verdi.



I spillet er det tre forskjellige scener. Hovedmenyen (se ovenfor), 1 Player og 2 Player (se under).



2. Assets

2.1 Materials og Textures

Enkelte textures er laget i unity men noen er også hentet fra asset store (se kilder)

2.2 Sounds

To forskjellige lyder er brukt i dette spillet. En «klokk» lyd og en «bopp» lyd. Begge er selvlagde og er laget i programmet Audacity. Disse lydene blir spilt av når kulen kolliderer med noe.

2.3 Sprites

Fjell bakgrunnen og hovedmeny bakgrunnen er hentet fra nettet og kan bli funnet under kilder. De resterende spritesene som er brukt har jeg laget i photoshop og vises etter man har trykket på «Show Controls» knappen.

2.4 Scripts

I spillet er det tre mapper med scripts:

- 1 Player. Som inneholder scriptsene:
  - Control
  - MotionPhysics*

- 2 Player. Som inneholder scriptsene:
  - ControlMP
  - MotionPhysicsMP*

- General. Som innholder scriptsene:
  - ODESolver*
  - ODE*
  - Projectile (arver fra ODE) *
  - DragProjectile (arver fra Projectile) *
  - Medal
  - SceneSelect

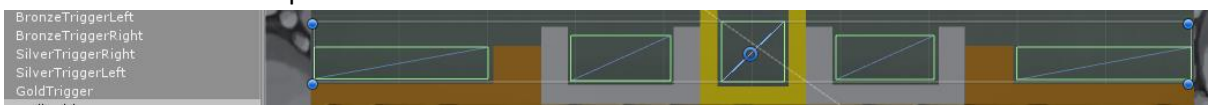
I noen av scriptsene har jeg implementert kode og formler som har blitt vist i boka, dette gjelder:

- Kapittel 4: Basic Kinematics
- Kapittel 5: Projectiles
- Kapittel 6: Collisions

Disse scriptsene er markert med en stjerne «*». All kode er skrevet i C#. Koden blir forklart senere i dokumentet.

3. Generelt i spillet

Når spilleren stopper opp i en kurv eller på gresset har jeg laget «usynlige» bokser som er markert som Is Trigger. Disse boksene, i kombinasjon med Medal scriptet avgjør og oppdaterer medalje teksten ettersom hvor spilleren lander.



Kulen ignorerer kollisjon med disse boksene ved hjelp av denne kodelinjen

```
&& hit.collider.tag != "MedalTag"
```

Som befinner seg i SphereCast sjekken i FixedUpdate() i MotionPhysics(MP) scriptet. Medal scriptet er festet til alle disse boksene. Så når kulen triggerer en av boksene kan riktig medalje tekst bli satt utifra dette. Her er en kodebit fra Medal scriptet:

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject == player1)
    {
        medalNameP1 = this.gameObject.name;
    }
    else if (other.gameObject == player2)
    {
        medalNameP2 = this.gameObject.name;
    }
    else
    {
        medalName = this.gameObject.name;
    }
}
public string giveMedal()
{
    return medalName;
}
```

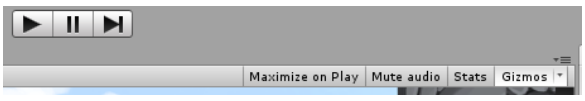
Så når kulen står stille blir stringen fra Medal scriptet hentet inn i MotionPhysics(MP) scriptet og medaljeteksten blir oppdatert ettersom hvor kulen til slutt lander.

```
else if (medal.giveMedal() == "BronzeTriggerLeft" ||
         medal.giveMedal() == "BronzeTriggerRight")
{
    control.medalLabel.text = "Bronze medal...";
    medalGiven = true;
}
```

For å visualisere hastighet, retning og hvor kulen kolliderer la jeg til to kodelinjer for å gjøre dette. Dette har ikke noe spesiell funksjon, men det kan være litt gøy å se dette. Det ser slik ut:

https://gyazo.com/aacf102ccccdb6dde971c4904c8dfe73

Hvis ønsket kan dette skrus på ved å aktivere «Gizmos».



Dette blir laget ved hjelp av disse to kodelinjene:

```
Debug.DrawLine(transform.position, hit.point, Color.red, 1, false);

Debug.DrawLine(transform.position, transform.position + velocity, Color.green, 0.05f,
false);
```

4. Fysikken

Nå til den viktigste delen av spillet, fysikken.

Det hele begynner etter man har trykket Space. Noe som registreres i Control(MP) scriptet:

```
if (Input.GetKey(KeyCode.Space))
{
    if (motionPhysics.fired != true)
    {
        fireVector = startPos.transform.position - barrel.transform.position;
        velocity = fireVector.normalized * startVelocity;
        motionPhysics.setVelocity(velocity);
        motionPhysics.Fire();
    }
}
```

Når dette har skjedd blir Fire() funksjonen kjørt i MotionPhysics(MP).

```
public void Fire()
{
    fired = true;
    velocity = control.getVelocity();
    DragProjectile = new DragProjectile(transform.position, velocity,
                         Time.fixedDeltaTime, mass, area, rho, dragCo);
}
```

4.1 DragProjectile

Her blir hastighetsvektoren hentet fra Control scriptet og brukt til å lage et DragProjectile objekt som bruker parameterne som sett ovenfor til å kalkulere bevegelsen til kula. For å utregne denne bevegelsen blir fire forskjellige scripts brukt; **ODESolver**, **ODE**, **Projectile** og **DragProjectile**. Det er i hovedsak tre funksjoner som regner ut det hele:

- **RungeKutta4**. Som befinner seg i **ODESolver**.
- **updateLocationAndVelocity**. Som befinner seg i **Projectile**.
- **getRightHandSide**. Som befinner seg i **DragProjectile**.

La oss start i ODESolver klassen. Hovedoppgaven til denne klassen er å løse fjerde orden RungeKutta likning, og ser slik ut.

```
public class ODESolver
{
    public static void RungeKutta4(ODE ode, float ds)
    {
        int j;
        int numEqns = ode.getNumEqns();
        float s;
        float[] q;
        float[] dq1 = new float[numEqns];
        float[] dq2 = new float[numEqns];
        float[] dq3 = new float[numEqns];
        float[] dq4 = new float[numEqns];

        s = ode.getS();
        q = ode.getAllQ();

        dq1 = ode.getRightHandSide(s, q, q, ds, 0.0f);
        dq2 = ode.getRightHandSide(s + 0.5f * ds, q, dq1, ds, 0.5f);
        dq3 = ode.getRightHandSide(s + 0.5f * ds, q, dq2, ds, 0.5f);
        dq4 = ode.getRightHandSide(s + ds, q, dq3, ds, 1.0f);

        ode.setS(s + ds);

        for (j = 0; j < numEqns; ++j)
        {
            q[j] = q[j] + (dq1[j] + 2.0f * dq2[j] + 2.0f * dq3[j] + dq4[j]) / 6.0f;
            ode.setQ(q[j], j);
        }

        return;
    }
}
```

Her brukes følgende likninger, i tillegg til getRightHandSide funksjonen.

$$\Delta z_1 = v(z_n, t_n)\Delta t$$

$$\Delta z_2 = v(z_n + \frac{1}{2}\Delta z_1, t_n + \frac{1}{2}\Delta t)\Delta t$$

$$\Delta z_3 = v(z_n + \frac{1}{2}\Delta z_2, t_n + \frac{1}{2}\Delta t)\Delta t$$

$$\Delta z_4 = v(z_n + \Delta z_3, t_n + \Delta t)\Delta t$$

Hvor $\Delta z_{1\ til\ 4}$ er dq1 til dq4. Og likningen under brukes i **for loopen,** som sett på bildet, til å oppdatere variabel verdiene til de uavhengige og avhengige variablene.

$$z_{n+1} = z_n + \frac{\Delta z_1}{6} + \frac{\Delta z_2}{3} + \frac{\Delta z_3}{3} + \frac{\Delta z_4}{6}$$

Så kan vi gå over til getRightHandSide() funksjonen over hvor dataen fra RungeKutta4 brukes. Selve getRightHandSide funksjonen kan variere ettersom hva man vil oppnå. I mitt tilfelle vil jeg simulere bevegelsen til en projektil som blir påvirket av luftmotstand. Hvis jeg skulle bruke ODESolveren, som er en universal solver til å for eksempel simulere fjær kraft, ville denne funksjonen sett en del

annerledes ut. Dette er min getRightHandSide funksjon:

```
public override float[] getRightHandSide(float s, float[] q, float[] deltaQ, float ds, float qScale)
{
    float[] dQ = new float[6];
    float[] newQ = new float[6];

    for (int i = 0; i < 6; ++i)
    {
        newQ[i] = q[i] + qScale * deltaQ[i];
    }

    float vx = newQ[0];
    float vy = newQ[2];
    float vz = newQ[4];

    float v = Mathf.Sqrt(vx * vx + vy * vy + vz * vz) + Mathf.Pow(10, -8);

    float Fd = 0.5f * rho * area * dragCo * v * v;

    dQ[0] = -ds * Fd * vx / (mass * v);
    dQ[1] = ds * vx;
    dQ[2] = ds * (G - Fd * vy / (mass * v));
    dQ[3] = ds * vy;
    dQ[4] = -ds * Fd * vz / (mass * v);
    dQ[5] = ds * vz;

    return dQ;
}
```

Øverst kan man se parameterne funksjonen tar inn, disse parameterne er input fra RungeKutta4 funksjonen som nevnt tidligere. Her blir en rekke elementer regnet ut.

For disse utregningen er disse likningene brukt:

$v = \sqrt{v_x^2 + v_y^2 + v_z^2}$ Utregning av hastighet

$F_D = \frac{1}{2}\rho v^2 A C_D$ Utregning av Drag Force, med massetetthet, hastighet, areal, drag koeffisient

$a_x = -\frac{F_D v_x}{mv}$ Utregning av akselerasjon i X retning med Drag Force (dQ[0])

$a_y = -g - \frac{F_D v_y}{mv}$ Utregning av akselerasjon i Y retning med Drag Force og gravitasjon (dQ[2])

$a_z = -\frac{F_D v_z}{mv}$ Utregning av akselerasjon i Z retning med Drag Force (dQ[4])

$\frac{dx}{dt} = v_x$ Utregning av posisjon på X planet (dQ[1])

$\frac{dx}{dt} = v_y$ Utregning av posisjon på Y planet (dQ[3])

$\frac{dx}{dt} = v_z$ Utregning av posisjon på Z planet (dQ[5])

Når alle utregninger har blitt gjort så kan **updateLocationAndVelocity** funksjonen bli kalt.

```
DragProjectile.updateLocationAndVelocity(Time.fixedDeltaTime);
```

Dette gjøres i FixedUpdate() funksjonen, noe som gjør at posisjonen og hastigheten til kula blir oppdatert hver «fixed update». Dermed blir kula forflyttet gjevnt gjennom spillet, og beveger seg ikke noe raskere eller saktere på grunn av datamaskinens ytelse.

4.2 Kollisjon

All kode for kollisjon i spillet ligger i MotionPhysics og MotionPhysicsMP scriptene. Selv om disse to scriptene er ganske like er det litt forskjell når det gjelder kollisjon og oppdatering av medalje. Så jeg syntes at det ville være best om jeg separerte MultiPlayer og enkeltspiller scriptene.

La oss starte med enkeltspiller kollisjon (MotionPhysics). I enkelt spiller modusen er spiller kulen det eneste med fysikk som beveger seg. Alle elementer den krasjer med står stille. Dette gjøre at kollisjon fysikken blir litt «simplere» enn når spiller kulene kolliderer med hverandre.

Selve kollisjon sjekken skjer i FixedUpdate fordi det alltid må sjekkes om kulen kolliderer med noe. Siden det er en **kule** som skal kollidere med noe har jeg brukt SphereCast i Unity. Dette er deskripsjonen fra unity doc siden om SphereCast:

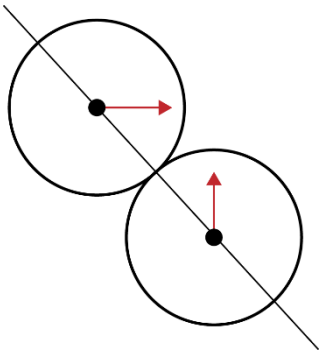
Casts a sphere along a ray and returns detailed information on what was hit.

https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html

Koden jeg har laget for sjekken ser slik ut:

```
if (Physics.SphereCast(transform.position,
                    playerRadius,
                    directionVector,
                    out hit,
                    directionVector.magnitude)
                    && hit.collider.tag != "MedalTag
{
```

Når dette skjer (når **if** sjekken er **true**) så starter kollisjon fysikken.



I det øyeblikket kulen kolliderer med noe så dannes det en «Line of Action». Denne brukes til å danne en normal, som vi derfra kan finne vinkelen mellom de to kolliderende objektene. I Unity er det ganske simpelt å finne denne vinkelen. Fordi man får tilgang til denne normalen med en gang kulen kolliderer med et objekt. Koden jeg har skrevet for dette ser slik ut:

```
angle = Mathf.Deg2Rad * Vector3.Angle(Vector3.right, hit.normal);
```

Etter kollisjonen «roteres» Line of Action for å finne utfalls vinkelen.

Det neste som skjer etter vinkelen har blitt funnet er at hastighets vektoren blir oppdatert til det den skal være etter kollisjonen har tatt plass, dette har jeg laget en funksjon for.

```
velocity = getPostVelocity(velocity.x, velocity.y, mass, massOther, e, angle);
```

Det funksjonen **getPostVelocity** gjør, er å finne inn og ut hastigheten ved å bruke parameterne som er gitt til den. Og deretter returnere den endelige hastigheten som brukes i FixedUpdate. Selve funksjonen ser slik ut:

```
public Vector3 getPostVelocity(float xVelocity, float yVelocity, float mass, float
massOther, float e, float angle)
{
    float Vp_1 = (xVelocity * Mathf.Cos(angle)) + (yVelocity * Mathf.Sin(angle));
    float Vn_1 = (-xVelocity * Mathf.Sin(angle)) + (yVelocity * Mathf.Cos(angle));

    float Vp_1After = ((mass - e * massOther) / (mass + massOther)) * Vp_1;

    Vector3 postVelocity =
    new Vector3(Vp_1After * Mathf.Cos(angle) - (Vn_1) * Mathf.Sin(angle),
                Vp_1After * Mathf.Sin(angle) + Vn_1 * Mathf.Cos(angle), 0);

    return postVelocity;
}
```

Formlene brukt i denne funksjonen er som følgene:

$v_p = v_x \cos(\theta) + v_y \sin(\theta)$ Formelen for Vp_1

$v_n = -v_x \sin(\theta) + v_y \cos(\theta)$ Formelen for Vn_1

$v'_{1x} = \frac{m_1 - e m_2}{m_1 + m_2} v_{1x}$ Formelen for Vp_1After, denne ser annerledes ut i MP scriptet.

$v'_x = v'_p \cos(\theta) - v_y \sin(\theta)$ Farten i X retning som returneres

$v'_y = v'_p \sin(\theta) + v_y \cos(\theta)$ Farten i Y retning som returneres

Etter disse kalkulasjonene blir den nye hastighets vektoren returnert og posisjonen til kula blir dermed oppdatert.

Nå over til kollisjon med to spillere (MotionPhysicsMP). Fysikken er i grunn ganske lik enkelt spiller, med noen untak. For det første har jeg lagt til enda en **if sjekk**, for å sjekke om kulene kolliderer med hverandre.

```
if ((hit.collider.tag == "Red" && tag == "Blue") ||
    (hit.collider.tag == "Blue" && tag == "Red"))
{
```

Hvis dette skjer så blir en modifisert **getPostVelocity** funksjon kalt, med navn **getPostVelocityP2P.** Denne returnerer et vektor array på størrelse 2, i stedet for å returnere en enkelt vektor.

```
Vector3[] newVelocity = getPostVelocityP2P(velocity.x, velocity.y, mass, e, angle);
velocity = newVelocity[0];
opponent.velocity = newVelocity[1];
```

MotioPhysicsMP scriptet er festet til begge spiller kulene så oppdaterer de hastighetene til hverandre med koden ovenfor. Siden det bare er noen tillegg i koden for **getPostVelocityP2P** kommer jeg bare til å nevne forskjellene. Disse forskjellene er at det har blitt lagt til enda et sett med variabler for fart til motstanderen i tillegg til at en av formlene er litt modifisert. Dette er lagt til:

```
float mass_Other = opponent.mass;
float velX_Other = opponent.velocity.x;
float velY_Other = opponent.velocity.y;

float Vp_2

float Vn_2

float Vp_2After

Vector3 postVelocity2
```

Vp_1After og Vp_2After bruker nå formlene:

$$v'_{1x} = \frac{m_1 - em_2}{m_1 + m_2} v_{1x} + \frac{(1+e)m_2}{m_1 + m_1} v_{2x}$$
$$v'_{2x} = \frac{(1+e)m_1}{m_1 + m_2} v_{1x} + \frac{m_2 - em_1}{m_1 + m_1} v_{2x}$$

Dette er for å endre farten og retningen til kula utifra hva farten og retningen på den andre kula er når de kolliderer.

Til slutt, hvis det ikke skjer en kollisjon så blir farten og posisjonen oppdatert som vanlig.

```
else
{
    velocity = DragProjectile.getVelocity();
    transform.position = DragProjectile.getPosition();
}
```

For å avslutte vil jeg bare nevne at kollisjonen ikke fungere som den skal 100% av gangene. Jeg har prøvd å fikse det så den gjør det, men har ikke fått tid til å få det til…

Kilder

Unity Assets:

Textures: https://www.assetstore.unity3d.com/en/#!/content/57060

Fjell BG: http://opengameart.org/sites/default/files/styles/watermarked/public/background0.png

Blurry BG: http://traveldilse.com/wp-content/uploads/2014/07/blurred-background-81.jpg

Fysikken:

Boka: Physics for Game Programmers av Grant Palmer

Dokumentasjonen:

Kollisjon bildet: http://orm-chimera-prod.s3.amazonaws.com/1234000001654/figs/htc2_0508.png