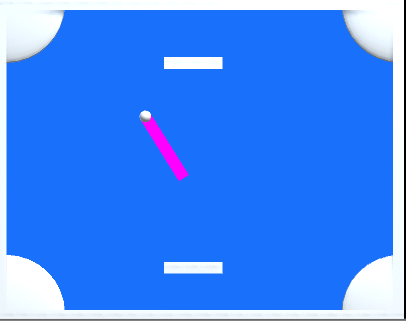
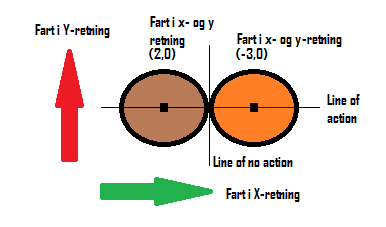
# Pongy

<<Pongy>> er et todimensjonalt sportspill som simulerer en slags bordtennis.

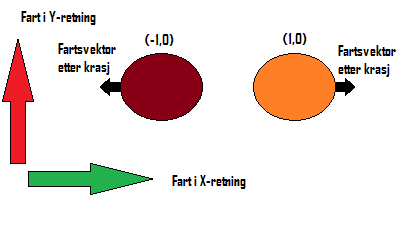


# Fysikken.

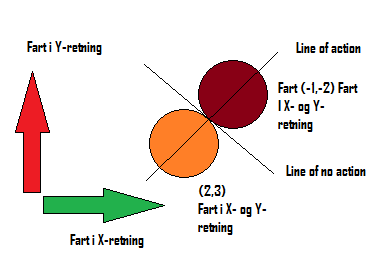
Fysikken som ligger til grunnlag for spillet er kollisjoner i to og «en» dimensjon. I en dimensjon jobber en bare med farten og posisjon i + og – retning av en akse (som for eksempel x aksen) og et eksempel på det kan være to klinkekuler som krasjer i hverandre sett fra siden. Etter et krasj vil farten og posisjon til klinkekulene endre seg i en retning. Klinkekulene kan for eksempel henge fast og stoppe opp ved et krasj, eller de kan begge bevege seg i motsatt retning og ha stikkmotsatt fartsretning i forhold til farten før krasjet. Det som det er viktig å notere seg her at det er ingen endring i farten i Y-retning. En trekker en linje (som kalles Line of Action) gjennom midtpunktet på begge klinkekulene i det kollisjonen pågår, og farten som er vinkelrett på denne linjen endrer seg ikke (Line of no action). Merk at fartsvektorene er parallell til «line of action» som igjen er parallell til X-retningen i koordinatsystemet.



Etter krasj..



I en dimensjon er dette veldig intuitivt, men det en kanskje ikke er klar over sånn helt uten videre er at det også stemmer i 2-dimensjoner. Farten som er vinkelrett på Line of Action endrer seg heller ikke i 2-dimensjoner.



Det eneste vi er klar over er at farten i retningen av «Line of no Action» ikke endrer seg mens farten i retningen av «Line of action» endrer seg. I **akkurat** dette tilfellet (se bildet over) er fartsvektorene ikke parallelle med «Line of action», men istedenfor parallelle med x og y-koordinatene. **1)** For å finne ut av farten i retningen som er parallell med «Line of Action», og «Line of no action» må vi bruke en roteringsmatrise. **2)** En roterer da farten i x-retningen slik at den blir parallell til «Line of action», og roterer farten i y-retningen slik at den blir parallell til «Line of no action». **3)** Videre kan vi så regne ut den nye farten parallell til «Line of action» og parallell til «Line of no action» etter kollisjonen. **4)** Tilslutt roterer vi fartskomponentene tilbake slik at de blir parallelle til x- og y-aksen igjen.

# Variabler og Input.

1. Massen til objektene, og
2. Farten til objektene , og
3. Koeffisient av restitusjon, .
4. Grader,

# Formler og implementasjon.

I 2 dimensjoner er det første en må vite hvor mange grader en må rotere fastvektorene. I koden under blir det laget en vektor fra det første objektets midtpunkt til det andre objektets midtpunkt. Denne koden brukes når to sirkler krasjer med hverandre. Med andre ord er dette «Line of action».

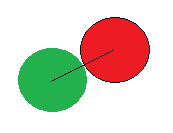
private Vector3 getLineofActionCircles(GameObject[] arry) {

Vector3 line;

line = arry[1].gameObject.transform.position - arry[0].gameObject.transform.position;

return line;

}



Denne kodebiten brukes når en sirkel kollider med en firkant. Her blir det laget en «Line of action» fra sirkelens midtpunkt til kollisjonspunktet.

private Vector3 getLineofActionSquares(GameObject[] arry)

{

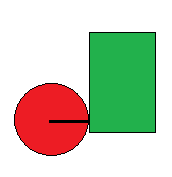
Vector3 line;

line = collis.point - arry[0].gameObject.transform.position;

Debug.Log(line.x + " " + line.y);

return line;

}



For å finne gradene fartsvektorene må rotere tar vi vektorlinjen vi lagde tidligere (Line of action), og sende den inn i funksjonen under. Funksjonen tar inn en Linje med x- og y-komponenter og finner gradene ved hjelp av den trigonometriske funksjonen . Svaret blir gitt i radianer og vi bruker derfor Mathf.Rad2Deg til å konvertere til grader. I den siste linjen 2) sørger vi for at gradene tilslutt alltid blir oppgitt i grader mellom 0 og 360 grader. Vis gradene er under eller lik 0 legger vi til 360 grader.

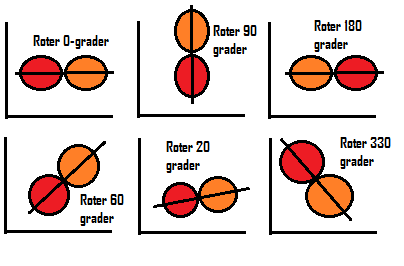
private float getDegrees(Vector3 line) {

1) float angle = (Mathf.Rad2Deg \* Mathf.Atan2(line.y, line.x));

2) angle = (angle > 0.0f ? angle : (360.0f + angle));

return angle;

}



Nå som vi har antall grader en skal rotere fartsvektoren må dette gjøres for begge objektene som kolliderer, og med formlene under. er farten langs «Line of actions», mens farten er farten langs «Line of no action».

Dette er formlene som brukes til å kalkulere den nye fartens langs «Line of action» for ***objekt 1***, og ***objekt 2***. er koeffisienten av restitusjon, og er et tall som vanligvis ligger mellom 0 og 1. Tallet er forholdet mellom den relative hastigheten etter, til den relative hastigheten før kollisjonen. Vis tallet er 1 vil det si at objektene ikke mister noe fart under kollisjonen, mens vis tallet er 0 vil de miste all fart og omtrent henge sammen. er massen til ***objekt 1*** mens er massen til ***objekt 2***.

Farten langs «Line of no action» er lik før og etter kollisjonen.

Ved hjelp av formlene under roterer vi farten tilbake til X og Y-retning.

public static class Collisionspeed{

public static Vector2[] getPostspeed(Vector2 objectonespeed, Vector2 objecttwospeed, float massone, float masstwo, float e, float angle) {

float Vp\_a = (objectonespeed.x \* Mathd.CosD(angle)) + (objectonespeed.y \* Mathd.SinD(angle));

float Vn\_a = (-objectonespeed.x \* Mathd.SinD(angle)) + (objectonespeed.y \* Mathd.CosD(angle));

float Vp\_b = (objecttwospeed.x \* Mathd.CosD(angle)) + (objecttwospeed.y \* Mathd.SinD(angle));

float Vn\_b = (-objecttwospeed.x \* Mathd.SinD(angle)) + (objecttwospeed.y \* Mathd.CosD(angle));

float Vp\_ac = (((massone - e \* masstwo) / (massone + masstwo)) \* Vp\_a) + ((((1 + e) \* masstwo) / (massone + masstwo)) \* Vp\_b);

float Vp\_bc = ((((1 + e) \* massone) / (massone + masstwo)) \* Vp\_a) + (((masstwo - (e \* massone)) / (massone + masstwo)) \* Vp\_b);

float Vn\_ac = Vn\_a;

float Vn\_bc = Vn\_b;

Vector2 objectonenewspeed = new Vector2(Vp\_ac \* Mathd.CosD(angle) - (Vn\_ac) \* Mathd.SinD(angle), Vp\_ac \* Mathd.SinD(angle) + Vn\_ac \* Mathd.CosD(angle));

Vector2 objecttwonewspeed = new Vector2(Vp\_bc \* Mathd.CosD(angle) - (Vn\_bc) \* Mathd.SinD(angle), Vp\_bc \* Mathd.SinD(angle) + Vn\_bc \* Mathd.CosD(angle));

return new Vector2[] {objectonenewspeed, objecttwonewspeed};

}

}

I programkoden over er det implementert formlene. Formlene er laget i en egen static klasse slik at en enkelt når som helst kan regne ut farten til to objekter som krasjer med hverandre.

Collisionspeed.getPostspeed(Vector1, Vector2, mass1, mass2, e, angle);

Kilder

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