LINKÖPINGS UNIVERSITET

MODELLERINGSPROJEKT

TNM085

Amazeballs

Sofie LINDBLOM
Anton ARBRING
David LINDH

Examinator
Anna LOMBARDI

13 mars 2014

Sammanfattning

I kursen TNM085, modelleringsprojekt skapas ett projekt vars syfte är att modellera ett fysikaliskt system av valfri typ. I den här rapporten redogörs om hur ett antal kolliderande studsbollar simuleras. Inspirationen är hämtad från en reklam Sony gjorde 2009 där man släpper tusentals studsbollar ner för en gata i San Fransisco [1].

Figur 1: Skärmbild från Sonys reklamfilm

Innehållsförteckning

Figurer

Kapitel 1

Inledning

Efter att ha resonerat kring att simulera luftballonger, rinnande vatten eller rök landade gruppen tillslut i beslutet att simulera studsbollar. Motiveringen till beslutet var att det på ett smidigt sätt skulle gå att involvera interaktivitet genom att användaren kan välja utvalda attribut som till exempel radien på studsbollarna. Huvudsakligt fokus har varit en fysikaliskt realistisk studs mellan boll och omgivning samt kollisionshanteringen mellan bollarna. Sekundärt fokus har varit möjlighet till interaktivitet för användaren och att simulera bollarna i en mer komplex omgivning.

1.1 Syfte

Syftet med det här projektet är att implementera fysiska samband i en teknisk tillämpning. Att kunna bemästra fysikaliska lagar och regler så pass väl att det är möjligt att avgöra vilka förenklingar som kan genomföras och fortfarande uppnå ett realistiskt resultat. Tanken är även att en introduktion till rapportskrivningsverktyget LaTeX ska erhållas.

1.2 Förarbete

Arbetet inleddes med undersökningar av hur liknande problem lösts av andra. I ett tidigt skede övervägdes att simulera studsar med hjälp av fjäderekvationer och konservering av volym. Det insågs dock att ett bättre alternativ är att använda sig av en så kallad 'Coefficient of restitution' , både med avseende på komplexitet och beräkningstyngd vid simulering av ett stort antal bollar [2].

Det spenderades en del tid på att undersöka olika simuleringsverktyg, jämföra openGL och WebGL samt vilka bibliotek och resurser som bäst lämpar sig att använda. Efter undersökning och diskussion togs beslutet att använda three.js vilket är ett javascript-bibliotek som hanterar WebGL. [3].

Kapitel 2

Implementation

2.1 Fysiskt System

Nedan redogörs för de två fysiska system som efterliknas i den tekniska implementationen.

2.1.1 Kollision med underlag

En studsande boll är ett dynamiskt hybridsystem. Det innehåller både kontinuerliga och statiska transformationer. De kontinuerliga tillstånden är givna i ekvation 2.1 och ekvation 2.2. Ekvation 2.1 beskriver accelerationen som beror på gravitationen. Ekvation 2.2 beskriver hastigheten i negativt y-led där dx är bollens position.

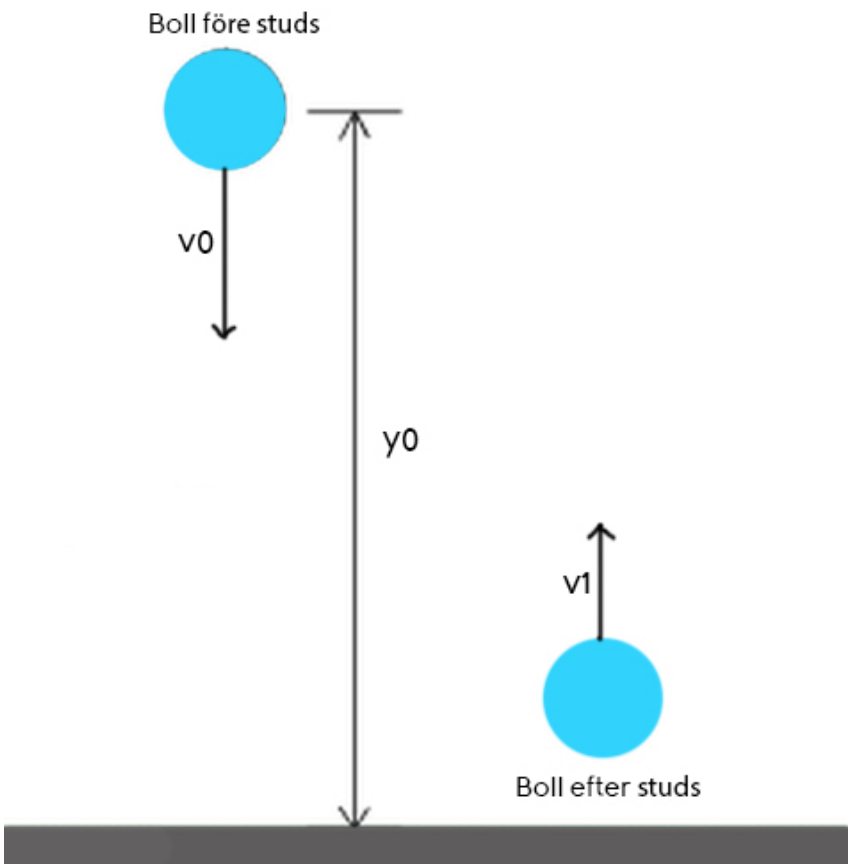
$$-g = dv/dt \tag{2.1}$$

$$v0 = dx/dt \tag{2.2}$$

Figur (2.1) visar en boll före och efter studs. Delvis elastisk stöt antas och ekvationen för kollision med underlaget anges i ekvation 2.3. Hastigheten före och efter kollisonen är relaterade genom en studskoefficient, k, nämnd i kapitlet ovan. Det uppstår därför en statisk transformation i studsögonblicket, därav är det ett hybridsystem.

$$v1 = -k * v0 \tag{2.3}$$

En studsande boll är exempel på Zeno-fenomenet. Vilket betyder att beteendet beror på ett oändligt antal studsar som sker under en fixt tidsintervall. När bollen förlorar energi uppstår ett stort antal kollisioner med marken i mindre intervall, det uppstår ett så kallat zeno-beteende. För att inte överbelasta datorn kommer en räknare införas som sätter hastighet och position till noll efter ett maxantal studsar. [4]



Figur 2.1: Illustration av kollision med underlag

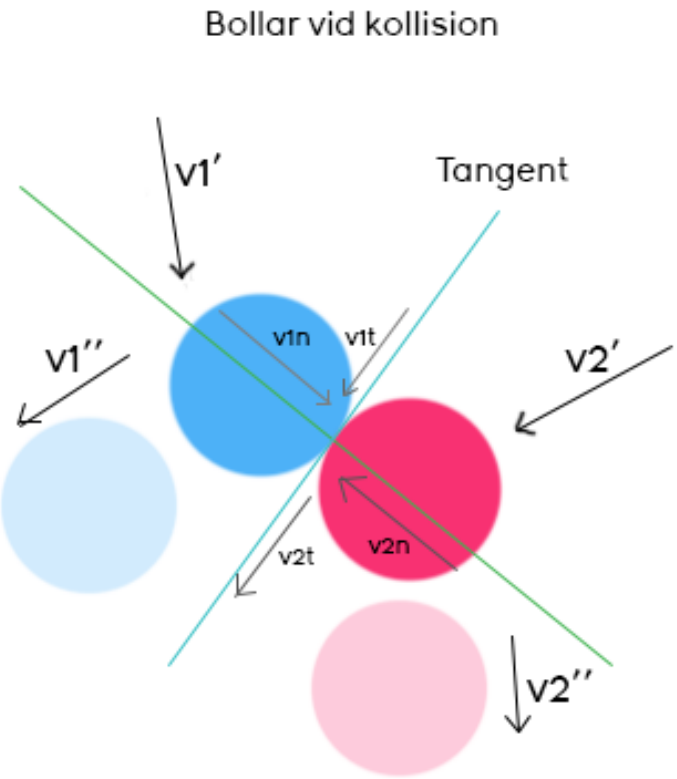
2.1.2 Kollision mellan bollar

En illustration över händelseförloppet vid kollision erhålles i figur 2.2.

Vid beräkning av de resulterande hastigheterna längs normalen efter kollision utnyttjas formeln för rörelsemängdens bevarande (ekvation 2.4) samt ett samband om hur stötkoefficienten är kopplad till bollarnas hastigheter (ekvation 2.5) [5]. Eftersom det i denna tillämpning antas att alla bollar har samma massa har bollarnas respektive massor eliminerats ur formlerna.

Tabell 2.1: Variabelbeskrivningar hastighetsvektorer

VARIABLER	BESKRIVNING
$V1t\prime$	Boll nummer ett, tangentvektor innan kollision
$V2t\prime$	Boll nummer två, tangentvektor innan kollision
$V1n\prime$	Boll nummer ett, normalvektor innan kollision
$V2n\prime$	Boll nummer två, normalvektor innan kollision
$V1t\prime\prime$	Boll nummer ett, tangentvektor efter kollision
$V2t\prime\prime$	Boll nummer två, tangentvektor efter kollision
$V1n\prime\prime$	Boll nummer ett, normalvektor efter kollision
$V2n\prime\prime$	Boll nummer två, normalvektor efter kollision
e	Konstant för energiförlusten
g	Konstant för gravitation
dt	Steglängd



Figur 2.2: Illustration av kollision mellan bollar

$$v1\prime + v2\prime = v1\prime\prime + v2\prime\prime \tag{2.4}$$

$$e = (v2n\prime\prime - v1n\prime\prime)/(v1n\prime - v2n\prime) \tag{2.5}$$

Utifrån dessa samband kan hastighetvektorerna efter kollision i normalriktningen beräknas genom ekvation 2.6 och 2.7.

$$v2n'' = (e(v1n' - v2n') + v1n' + v2n')/2 \tag{2.6}$$

$$v1n'' = v1n' + v2n' - v2n'' \tag{2.7}$$

Bollarna antas ha glatta ytor vilket betyder att den tangentiella kraften i kontaktytan är försumbar. Den tangetiella kraften bevars då vilket visas i ekvation 2.8 och 2.9.

$$v1t' = v1t'' \tag{2.8}$$

$$v2t' = v2t'' \tag{2.9}$$

Den resulterade hastigheten erhålles genom att addera hastighetskomposanterna i tangentiell- och normal-riktning för respektive boll (ekvation 2.10 och 2.11).

$$v1'' = v1t'' + v1n'' \tag{2.10}$$

$$v2'' = v2t'' + v2n'' \tag{2.11}$$

2.1.3 Eulers stegmetod

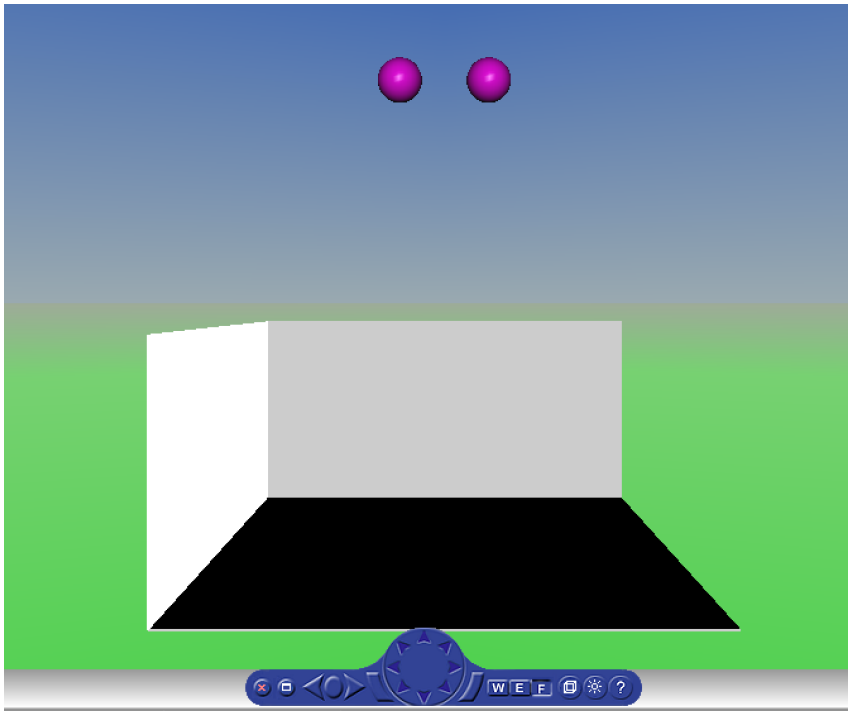
För att simulera rörelsen kommer Eulers stegmetod att användas. Detta är en numerisk metod där differentialekvationen för hastighet delas in i diskreta stegintervall och löses med hjälp av den aktuella derivatan och steglängden. Nedan hittas ekvation 2.12 som beskriver hur en ny hastighet i y-led påverkas av acceleration och steglängd.

$$v1'' = v1' + g * dt \tag{2.12}$$

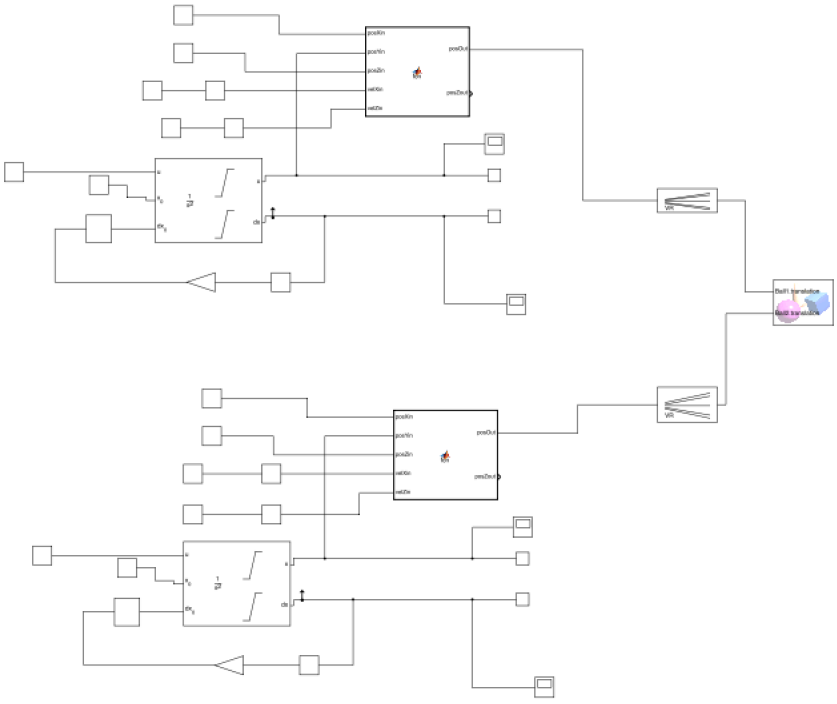
2.2 Simuleringar i MATLAB

Innan simulering av tillämpningen testades det att simulera två bollar med hjälp av en VRML (Virtual Reality Modelling Language) simulator i Matlab (Se Figur 2.3).

Funktionerna implementerades med hjälp av Simulink och gav en bra anknytning till tidigare kurser i modellering och simulering (Se Figur 2.4). Detta steg kan ses som överflödigt men gav trots allt gruppen en bra förståelse om de fysiska samband som råder samt vad som kan försummas utan märkbart försämrat resultat. Det utgjorde en bra utgångspunkt till att gå vidare och skriva egna funktioner och villkor för bollarna.



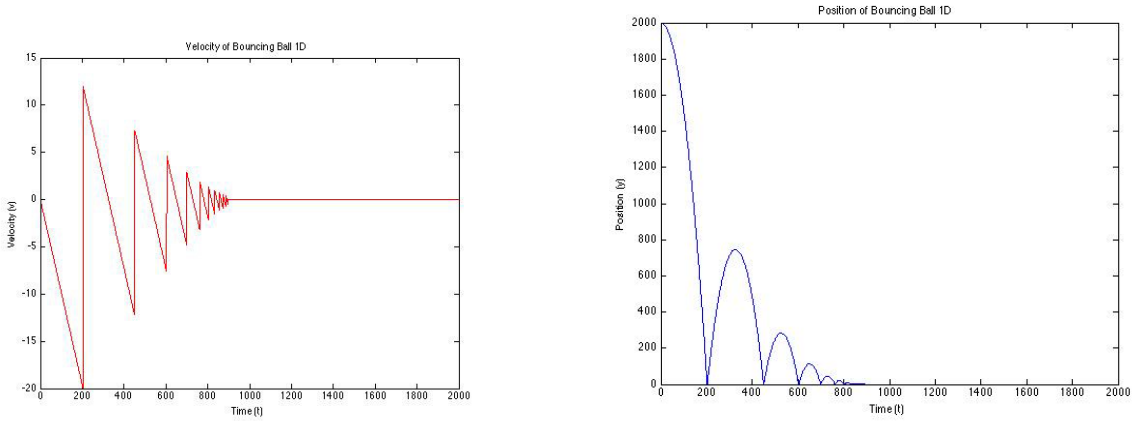
Figur 2.3: Två bollar simulerade med VRML-simulator i MATLAB



Figur 2.4: Blockschema över VRML-simuleringen i Simulink

2.2.1 Simulering i 1D

Som första steg modelleras en studsande boll i en dimension. Syftet med detta är att uppnå en god grundförståelse för att sedan utveckla modellen till flera dimensioner.



Figur 2.5: Hastighet och position för studsboll i 1D

För att verifiera framarbetade fysikaliska formler plottas positionen och hastigheten som funktion av tiden (Se figur (2.5)). Det fysikaliska sambanden som MATLAB implementationen är baserad på visas i ekvation (2.13) - (2.16) med tillhörande beskrivning av inblandade variabler i tabell 2.2.

Tabell 2.2: Variabelbeskrivningar MATLAB

	VARIABLER	BESKRIVNING VÄRDER
k	Energiförlust vid kollision med marken	0.6
g	Gravitationskraften	9.82
deltat	Tidsintervall mellan två uppdateringar	1
v[new]	Hastighet i kommande uppdatering	-
v[old]	Hastighet i nuvarande tillstånd	-
p[new]	Position i kommande uppdatering	-
p[old]	Position i nuvarande tillstånd	-

Om boll slår i marken:

$$v[new] = -k * v[old] \tag{2.13}$$

$$p[new] = v[new] * deltat \tag{2.14}$$

Om boll faller fritt i y-led:

$$v[new] = v[old] + g * deltat \tag{2.15}$$

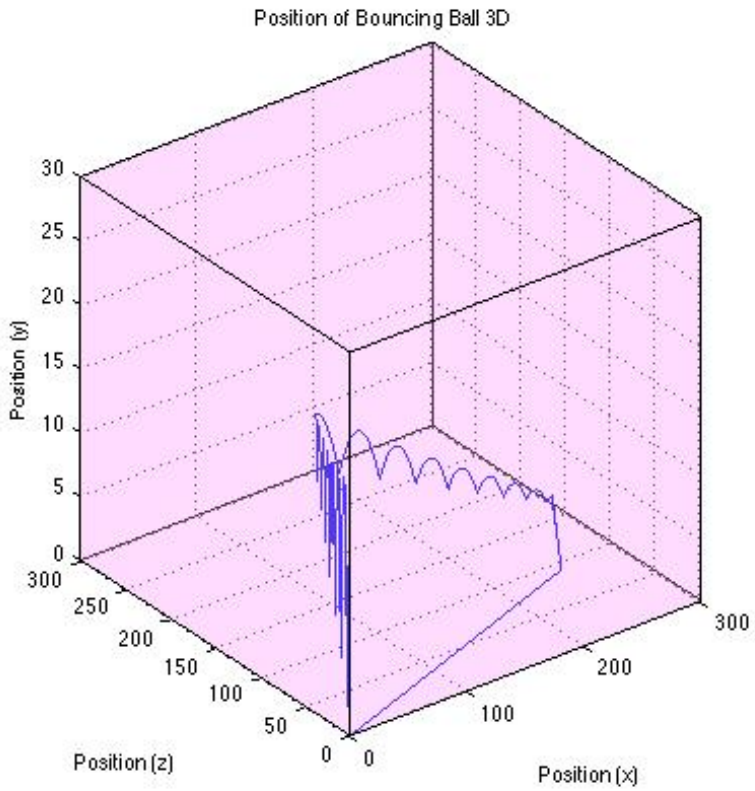
$$p[new] = p[old] * v[old] * deltat \tag{2.16}$$

2.2.2 Simulering i 3D

I ett steg mot slutliga implementationen utökas modellen från en dimension till tre. Mellansteget i två dimensioner har exkluderats från rapporten då det är samma tillväga gångssätt som i denna simulering.

Fysikaliska sambanden är desamma som i en dimension. Skillnaden är att samtliga variabler utökas från en till tre komponenter (x,y,z). Bollen får en initial hastighet (som är noll i y-led, eftersom det är i det ledet som gravitationen verkar) och en initial position.

Bollens position kontrolleras mot underlaget på samma sätt som tidigare men utöver det så kontrolleras även om kollision med väggarna i en kub har inträffat. Kuben har dimensionerna 300x300x300 och har sitt ena hörn placerat i origo (se figur 2.6).



Figur 2.6: Position av boll i 3D

Totalt görs fem stycken kontroller, en för varje sida av kuben bortsett från taket. Om koordinatens värde i det led som kontrolleras överstiger 300 eller understiger 0 görs en beräkning

av en ny hastighet längs den axeln. Om ingen kollision inträffar uppdateras position och hastighet precis på samma sätt som i simuleringen i en dimension. Nedan är ett exempel på hur en kollison med kubens sida i x-led hanteras, samtlig kod finns även i Bilaga B.

För kollision med vägg i x-led:

x-led:

$$v[new] \ = \ -k * v[old] \tag{2.17}$$

$$p[new] \ = \ p[old] \ + \ v[new] * deltat \tag{2.18}$$

y-led:

$$v[new] \ = \ v[old] + \ g * deltat \tag{2.19}$$

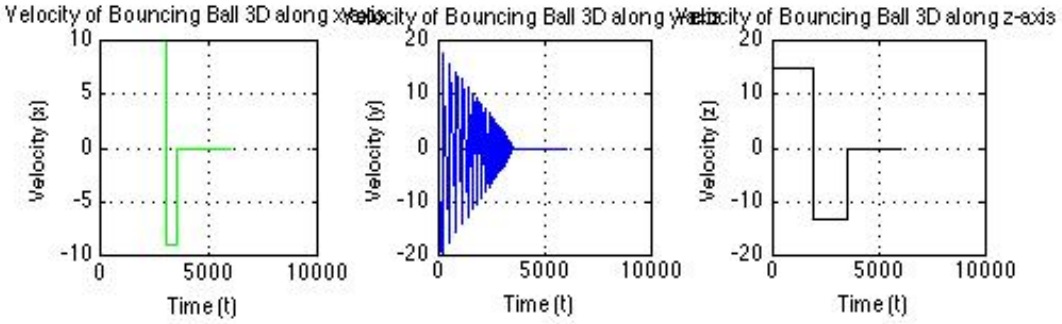
$$p[new] \ = \ p[old] \ + \ v[new] * deltat \tag{2.20}$$

z-led:

$$v[new] = v[old] \tag{2.21}$$

$$p[new] \ = \ p[old] \ + \ v[new] * deltat \tag{2.22}$$

Även i detta exempel används en räknare för att undvika ett oändligt antal studsar. Hastigheten i respektive led plottas gentemot en tidsaxel för tydligare tolkning av resultatet (Se figur 2.7).



Figur 2.7: Bollens hastighet i x-, y- respektive z-led

2.3 Simulering i WebGL

I den slutliga simulering används WebGL i kombination med three.js[3], målet är att simulera mer fysikaliskt realistiska kollisioner som även tar hänsyn till sneda sammanstötningar. Hastighetsvektorerna från de inblandade bollarna projiceras på två komposanter i kollisionens tangent- och normalriktningar. Tangenterna erhålles genom kryssprodukt med normalen och tangenten. Normalen erhålles genom ekvation 2.23 och tangenten genom att x- och y-koordinatens värden sätts till ett fixt värde, z-koordinatens värde räknas ut enligt ekvation 2.24, slutligen erhålles tangenten för de koliderande bollarna genom ekvation 2.25.

$$normal = p1.x - p2.x, p1.y - p2.y, p1.z - p2.z \tag{2.23}$$

Tabell 2.3: Variabelbeskrivningar JS

VARIABLER	BESKRIVNING
$Normal$	Normalen för de kolliderande bollarna
$p1$	Postionsvektor för boll nummer 1
$p2$	Postionsvektor för boll nummer 2
tz	tangentens z-koordinat
$Tangent$	Tangenten för de kolliderande bollarna

Tabell 2.4: Variabelbeskrivningar hastighetsvektorer

VARIABLER	BESKRIVNING
$V1t\prime$	Boll nummer ett, tangentvektor innan kollision
$V2t\prime$	Boll nummer två, tangentvektor innan kollision
$V1n\prime$	Boll nummer ett, normalvektor innan kollision
$V2n\prime$	Boll nummer två, normalvektor innan kollision
$V1t\prime\prime$	Boll nummer ett, tangentvektor efter kollision
$V2t\prime\prime$	Boll nummer två, tangentvektor efter kollision
$V1n\prime\prime$	Boll nummer ett, normalvektor efter kollision
$V2n\prime\prime$	Boll nummer två, normalvektor efter kollision
e	Konstant för energiförlusten

$$tz = (-(normal.x * 0.3) - (normal.y * 0.3))/normal.z \tag{2.24}$$

$$Tangent = (0.3, 0.3, tz) \tag{2.25}$$

Tangentiella energin bevaras [5] och energin i normalriktningen räknas ut enligt ekvation 2.30 och 2.31, dessa är härledda från ekvation 2.28 och 2.29.

Tangentiella hastighetsvektorer:

$$V1t\prime = V1t\prime\prime \tag{2.26}$$

$$V2t\prime = V2t\prime\prime \tag{2.27}$$

Normalens hastighetsvektorer:

$$e = (V2n\prime\prime - V1n\prime\prime)/(V1n\prime - V2n\prime) \tag{2.28}$$

$$V1n\prime + V2n\prime = V1n\prime\prime + V2n\prime\prime \tag{2.29}$$

$$V2n\prime\prime = e(V1n\prime - V2n\prime) + V1n\prime + V2n\prime \tag{2.30}$$

$$V1n\prime\prime = v1n\prime + v2n\prime - v2n\prime \tag{2.31}$$

Slutligen erhålls bollarnas hastigheter efter kollisionen genom att tangentiella och normalens hastighetsvektorer adderas.

Kapitel 3

Resultat och Diskussion

3.1 Förenklingar

Genom hela processen har vi försummat att simulera deformation som vanligtvis uppstår vid studs eller kollision. Implementationen av deformation ansågs för krävande och kategoriserades som en möjlig utveckling av systemet.

Ytterligare en förenkling som gjorts är att luftmotståndet försummats. Detta för att avlasta den redan beräkningstunga applikationen men framför allt för att den synliga inverkan på bollarna av luftmotståndet är knappt märkbar.

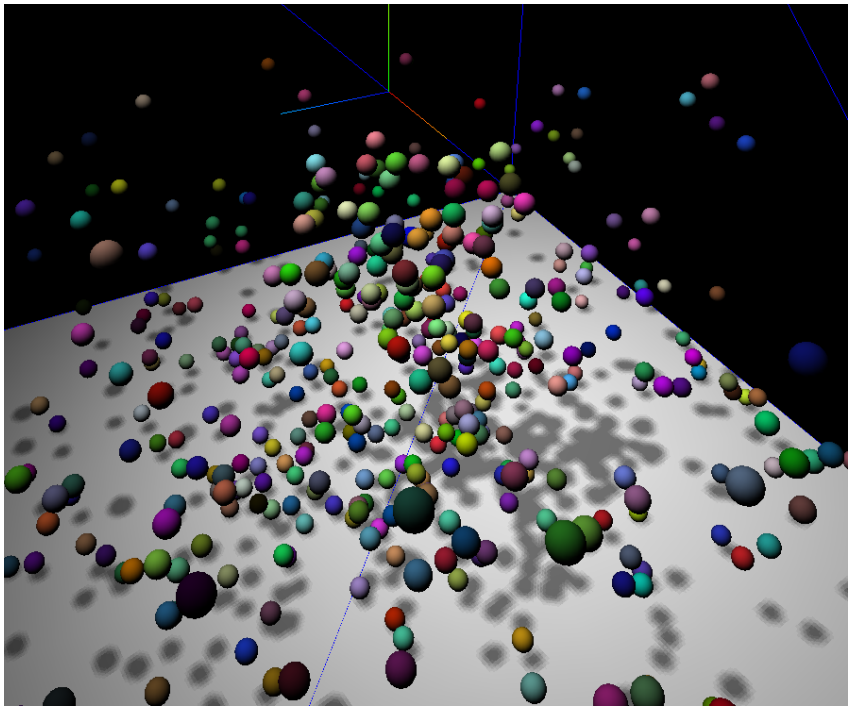
Vid kollison mellan bollar samt mellan omgivning och bollar har energiförlusten approximerats genom att multiplicera med en konstant. Denna förenkling är relativt korrekt men för att konstanten ska vara precis bör hänsyn tagits till ett flertalet variabler som till exempel elasticitet och friktion hos kolliderande föremål. Denna approximation hade liksom förenklingen med luftmotståndet ingen större inverkan på resultatet.

Grundidéen var bollarna skulle studsa ner för en gata snarlikt den bild från Sonys reklam som återfinns i figur 1. Simuleringen gjordes dock i en förenklad miljö (en box) för att fastställa att grundfunktionaliteten fungerar.

Bollarna får en initierad starthastighet, från början var det tänkt att gravitation, lutning och rotation skulle vara det som gav dem en initial hastighet och riktning. Rotationen som uppstår när bollarna krockar dels med varandra och dels med väggar och golv är en annan möjlig utvidgning av nuvarande applikation.

3.2 Resultat

Den slutliga applikationen kan köras genom att klicka här. Stommen av programmet är skrivet i javascript med stöd av three.js. Koden är uppdelad i fyra block. I det första blocket deklareras allt i scenen (kamera, sfär-objekt, plan, lampor, mus-interaktion och renderare). I block två skapas bollarna och blir tilldelade en initial hastighet och position. Block tre består av animeringen, där det i varje uppdatering anropas en funktion som kollar om kollision med väggar och andra bollar skett. I denna del av programmet sker även kontrollen om hur många studs varje boll gjort dittills i animationen. Som nämndes i Matlab-simulering implementerades en räknare för att förhindra oändligt antal studsar. Block fyra består av kollisionshanteringen som kallas om det i ett tidsintervall uppstått kollision. Koden återfinns i bilaga D och en bild från slutresultatet visas i figur 3.1.



Figur 3.1: Slutlig simulering av 200 bollar

Det krävs inga specifika hårdvarukrav för att köra applikationen. Användaren behöver dock en webläsare som stödjer WebGL.

3.3 Reflektion

Som sig så ofta inträffar går projektgruppen in med skyhöga ambitioner om vad som ska utföras under projektets gång. Andra kurser, engagemang och oförutsedda tidsdistraktioner kommer sedan i mellan och helt plötsligt är deadline runt hörnet. Det beskrivna scenariot är vad som hände denna projektgrupp. Men trots detta och det faktum att vi bara är tre personer är vi nöjda med resultatet.

Det har varit lärorikt och intressant att gå från fysikaliska formler skrivna med papper och penna hela vägen till att kunna se det implementerat live i sin webläsare. Gruppen har kämpat med att hitta gemensam tid att arbeta då alla läser olika kurser. Skriva rapport i LaTeX har även varit en stor utmaning för oss då alla är nybörjare.

Vårt mål är att hinna implementera viss interaktion mellan användaren och applikationen innan slutpresentationen. Om gruppen lyckas med detta är vi mycket nöjda med våra insatser i kursen och med slutresultatet.

Litteraturförteckning

[1] Sony. Bravaria Sony; 2009. Available from: http://www.youtube.com/watch?v=0_bx8bnCoiU.

[2] Research R. Coefficinet of Restitution;. Available from: http://www.racquetresearch.com/coeffici.html.

[3] ThreeJS. Threejs Library;. Available from: http://www.threejs.org.

[4] Matlab. Collision in Matlab;. Available from: http://www.mathworks.se/help/simulink/examples/simulation-of-a-bouncing-ball.html.

[5] Jansson PA, Grahn R. Mekanik. Studentlittertaur; 2014.

Bilaga A

1D Simulering i MATLAB

```
close all;
%Gravity free fall
grav = -0.0982;
%Step size
delta_t = 1;
%Time vector for plotting
t_vector = 0:1:2000;
%Store ball's position for corresponding t
p_vector = [2000];
%Stroe ball's velocity for corresponding t
v_vector = [0];
%counter
count=0;


for dt = 1:1:2000
    index_new = dt+1;
    index_old = dt;
    %If ball hits the floor
    if p_vector(index_old) <= 0 && v_vector(index_old) <= 0
        p_vector(index_old) = 0;
        v_vector(index_new) = -(0.6*v_vector(index_old));
        p_vector(index_new) = v_vector(index_new)*delta_t;
        count = count+1;
    else
        v_vector(index_new) = v_vector(index_old) + grav*delta_t;
        p_vector(index_new) = p_vector(index_old) + v_vector(index_old)*d

    end
    %Avoid infinite amount of bouncing
    if count>10
        p_vector(dt:2001) = 0;
        v_vector(dt:2001) = 0;
        break;
```

```
    end
end

plot(t_vector, p_vector);
title('Position of Bouncing Ball 1D')
xlabel('Time (t)')
ylabel('Position (y)')
%%hold on;
figure;
plot(t_vector, v_vector, 'r');
title('Velocity of Bouncing Ball 1D')
xlabel('Time (t)')
ylabel('Velocity (v)')
```

Bilaga B

3D Simulering i MATLAB

```
close all;
length = 6000;
%Gravity free fall
grav = [0,-9.82,0];
%Step size
delta_t = 0.01;
%Time vector for plotting
t_vector = 0:1:length;
%Position vector with time component [px,py,pz,t]
p_matrix = zeros(3, length+1);
p_matrix(:,1) = [2 , 20, 2];
p_matrix = cat(1, p_matrix , t_vector);
%Velocity vector with time component [vx,vy,vz,t]
v_matrix = zeros(3, length+1);
v_matrix(:,1) = [10, 0, 15];
v_matrix = cat(1, v_matrix , t_vector);
count = 0;
for dt = 1:1:length
    index_new = dt+1;
    index_old = dt;
    %If ball hits ground y-direction
    if p_matrix(2, index_old) <= 0 %&& v_matrix(2, index_old) <= 0
        %x-pos
        v_matrix(1, index_new) = v_matrix(1, index_old);
        p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
        %y-pos
        p_matrix(2, index_old) = 0;
        v_matrix(2, index_new) = -(0.90*v_matrix(2, index_old)) + grav(2)
        p_matrix(2, index_new) = v_matrix(2, index_new)*delta_t;
        %z-pos
        v_matrix(3, index_new) = v_matrix(3, index_old);
        p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind
        count = count +1;
    %If ball hits wall x = 0 in x-direction
```

```
elseif p_matrix(1, index_old) <= 0
    %x-pos
    p_matrix(1, index_old) = 0;
    v_matrix(1, index_new) = -(0.90*v_matrix(1, index_old))
    p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
    % y-pos
    v_matrix(2, index_new) = v_matrix(2, index_old) + grav(2)*delta_t
    p_matrix(2, index_new) = p_matrix(2, index_old) + v_matrix(2, ind
    % z-pos
    v_matrix(3, index_new) = v_matrix(3, index_old);
    p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind
    count = count +1;
    %If ball hits wall x = 300 in x-direction
elseif p_matrix(1, index_old) >= 300
    %x-pos
    p_matrix(1, index_old) = 300;
    v_matrix(1, index_new) = -(0.90*v_matrix(1, index_old));
    p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
    % y-pos
    v_matrix(2, index_new) = v_matrix(2, index_old) + grav(2)*delta_t
    p_matrix(2, index_new) = p_matrix(2, index_old) + v_matrix(2, ind
    % z-pos
    v_matrix(3, index_new) = v_matrix(3, index_old);
    p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind
    count = count +1;
%If ball hits wall z = 0 in z-direction
elseif p_matrix(3, index_old) <= 0
    %x-pos
    v_matrix(1, index_new) = v_matrix(1, index_old);
    p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
    % y-pos
    v_matrix(2, index_new) = v_matrix(2, index_old) + grav(2)*delta_t
    p_matrix(2, index_new) = p_matrix(2, index_old) + v_matrix(2, ind
    % z-pos
    p_matrix(3, index_old) = 0;
    v_matrix(3, index_new) = -(0.90*v_matrix(3, index_old));
    p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind
    count = count +1;
     %If ball hits wall z = 300 in z-direction
elseif p_matrix(3, index_old) >= 300
    %x-pos
    v_matrix(1, index_new) = v_matrix(1, index_old);
    p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
    % y-pos
    v_matrix(2, index_new) = v_matrix(2, index_old) + grav(2)*delta_t
    p_matrix(2, index_new) = p_matrix(2, index_old) + v_matrix(2, ind
    % z-pos
    p_matrix(3, index_old) = 300;
```

```
        v_matrix(3, index_new) = -(0.90*v_matrix(3, index_old));
        p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind
        count = count +1;
    %If no collision between ball and wall
    else
        %x-pos
        v_matrix(1, index_new) = v_matrix(1, index_old);
        p_matrix(1, index_new) = p_matrix(1, index_old) + v_matrix(1, ind
        % y-pos
        v_matrix(2, index_new) = v_matrix(2, index_old) + grav(2)*delta_t
        p_matrix(2, index_new) = p_matrix(2, index_old) + v_matrix(2, ind
        % z-pos
        v_matrix(3, index_new) = v_matrix(3, index_old);
        p_matrix(3, index_new) = p_matrix(3, index_old) + v_matrix(3, ind

    end

    if count>50
        v_matrix(1:3,dt:length+1)=0;
        p_matrix(1:3,dt:length+1)=0;

    end
end
%Plot a box works in 2013
%plot(plot::Box(0..300, 0..300, 0..300, Filled = FALSE,
%               LineColor = RGB::Black))

% Plot for position in (x, y)
% plot3(p_matrix(4,:), p_matrix(1,:), p_matrix(2,:));
% title('Position of Bouncing Ball 3D')
% xlabel('Time (t)')
% ylabel('Position (x)')
% zlabel('Position (y)')
% grid on
% % Plot for position in (y, z)
% figure;
% plot3(p_matrix(4,:), p_matrix(2,:), p_matrix(3,:), 'g');
% axis square
% title('Position of Bouncing Ball 3D')
% xlabel('Time (t)')
% ylabel('Position (y)')
% zlabel('Position (z)')
% grid on
% % Plot for velocity in (x,y)
% figure;
% plot3(p_matrix(4,:), v_matrix(1,:), v_matrix(2,:), 'r');
% axis square
% title('Velocity of Bouncing Ball 3D')
```

```
% xlabel('Time (t)')
% ylabel('Velocity (x)')
% zlabel('Velocity (y)')
% grid on

% Plot for position in (x,y,z) % plottar y uppåt pga gravitationen
figure;
plotcube([300 300 30],[0 0 0],.3,[1.0 .73 1.0]);
hold on
plot3(p_matrix(1,:), p_matrix(3,:), p_matrix(2,:), 'b');
axis square
title('Position of Bouncing Ball 3D')

xlabel('Position (x)')
ylabel('Position (z)')
zlabel('Position (y)')
grid on

% Plot for velocity in (x,y,z)
% figure;
% figure;
% plotcube([300 300 30],[0 0 0],.3,[1.0 .73 1.0]);
% hold on
% plot3(v_matrix(1,:), v_matrix(3,:), v_matrix(2,:), 'm');
% axis square
% title('Velocity of Bouncing Ball 3D')
%
% xlabel('Velocity (x)')
% ylabel('Velocity (z)')
% zlabel('Velocity (y)')
% grid on

% Plot for velocity in (x,t)
figure;
subplot(1,3,1), plot(v_matrix(4,:), v_matrix(1,:), 'g');
axis square
title('Velocity of Bouncing Ball 3D along x-axis')
xlabel('Time (t)')
ylabel('Velocity (x)')
grid on

% Plot for velocity in (y,t)

subplot(1,3,2), plot(v_matrix(4,:), v_matrix(2,:), 'b');
axis square
title('Velocity of Bouncing Ball 3D along y-axis')
xlabel('Time (t)')
ylabel('Velocity (y)')
```

```
grid on

% Plot for velocity in (z,t)

subplot(1,3,3), plot(v_matrix(4,:), v_matrix(3,:), 'black');
axis square
title('Velocity of Bouncing Ball 3D along z-axis')
xlabel('Time (t)')
ylabel('Velocity (z)')
grid on
```

Bilaga C

Kollision och Refakturering

C.0.1 Huvudfil

```
close all;
%length of simulation
length = 6000;
%ball radius
r = 1;
%Time vector for plotting
t_vector = 0:1:length;
%nr of bounces
count = 0;
%nr of balls
nr_of_balls = 2;

%Ball 1
    %Position vector with time component [px,py,pz,t]
    b1_p_matrix = zeros(3, length+1);
    b1_p_matrix(:,1) = [2 , 20, 2];
    b1_p_matrix = cat(1, b1_p_matrix , t_vector);
    %Velocity vector with time component [vx,vy,vz,t]
    b1_v_matrix = zeros(3, length+1);
    b1_v_matrix(:,1) = [10, 0, 15];
    b1_v_matrix = cat(1, b1_v_matrix , t_vector);

%Ball 2
    %Position vector with time component [px,py,pz,t]
    b2_p_matrix = zeros(3, length+1);
    b2_p_matrix(:,1) = [2 , 20, 298];
    b2_p_matrix = cat(1, b2_p_matrix , t_vector);
    %Velocity vector with time component [vx,vy,vz,t]
    b2_v_matrix = zeros(3, length+1);
    b2_v_matrix(:,1) = [10, 0, -15];
    b2_v_matrix = cat(1, b2_v_matrix , t_vector);

% Ball vector [bn_p_matrix , bn_v_matrix]
```

```
ball_vector = zeros((8*nr_of_balls),(length+1));
ball_vector = [b1_p_matrix; b1_v_matrix; b2_p_matrix; b2_v_matrix];

% Main loop
for dt = 1:1:length

for i = 1:1:nr_of_balls % P_temp = ball_vector(((i-1*4)+1):i*4,dt)
%     p_temp = ball_vector(((i-1*4)+1):i*4,1:length);
%     v_temp = ball_vector(((i*4)+1):i*8,1:length);

    %Extract p and v for ball number n
    p_temp = ball_vector((((i-1)*8)+1):4+(i-1)*8,dt);
    v_temp = ball_vector(((i-1)*8)+5:i*8,dt);

    %Calculate new values for p and v
    [p_temp,v_temp,count] = propagation(p_temp, v_temp, count, dt);

    %Insert new values at next time frame
    ball_vector((((i-1)*8)+1):4+(i-1)*8,dt+1) = p_temp;
    ball_vector(((i-1)*8)+5:i*8,dt+1) = v_temp;

end
    %Extract newly calculated p and v
    p1_temp = ball_vector(1:3,dt+1);
    p2_temp = ball_vector(9:11,dt+1);
    v1_temp = ball_vector(5:7,dt+1);
    v2_temp = ball_vector(13:15,dt+1);

    %Calculate distance between balls
    d = (p1_temp(1)-p2_temp(1))^2 + (p1_temp(2)-p2_temp(2))^2 + (p1_temp(
    dist = sqrt(d);


    %If collision occur
    if dist < 2*r
        ball_vector(5:7,dt+1) = v2_temp*0.95;
        ball_vector(13:15,dt+1) = v1_temp*0.95;
        count = count+1;
    end

    %Avoid infinite bouncing
    if count>100
        ball_vector(5:7,dt:length+1)=0;
        ball_vector(13:15,dt:length+1)=0;

    end
end
```

```
%Plot Bonding box and balls
 figure;
 plotcube([300 300 30],[0 0 0],.3,[1.0 .73 1.0]);
 hold on
 plot3(ball_vector(1,:), ball_vector(3,:), ball_vector(2,:), 'b');
 hold on
 plot3(ball_vector(9,:), ball_vector(11,:), ball_vector(10,:), 'g');
 axis square
 title('Position of Bouncing Ball 3D')

 xlabel('Position (x)')
 ylabel('Position (z)')
 zlabel('Position (y)')
 grid on
```

C.0.2 Funktionsfil

```
function [ p_out, v_out, count ] = propagation( p_in, v_in, count, dt )

%Gravity free fall
grav = [0,-9.82,0];
%Step size
delta_t = 0.01;
%p_out = p_in;
%v_out = v_in;

 %If ball hits ground y-direction
    if p_in(2) <= 0 %&& v_in(2) <= 0
        %x-pos
        v_out(1) = v_in(1);
        p_out(1) = p_in(1) + v_out(1)*delta_t;
        %y-pos
        p_in(2) = 0;
        v_out(2) = -(0.90*v_in(2))+ grav(2)*delta_t;
        p_out(2) = v_out(2)*delta_t;
        %z-pos
        v_out(3) = v_in(3);
        p_out(3) = p_in(3) + v_out(3)*delta_t;
        count = count +1;
    %If ball hits wall x = 0 in x-direction
    elseif p_in(1) <= 0
        %x-pos
        p_in(1) = 0;
        v_out(1) = -(0.90*v_in(1));
        p_out(1) = p_in(1) + v_out(1)*delta_t;
        % y-pos
        v_out(2) = v_in(2) + grav(2)*delta_t;
```

```
    p_out(2) = p_in(2) + v_out(2)*delta_t;
    % z-pos
    v_out(3) = v_in(3);
    p_out(3) = p_in(3) + v_out(3)*delta_t;
    count = count +1;
    %If ball hits wall x = 300 in x-direction
elseif p_in(1) >= 300
    %x-pos
    p_in(1) = 300;
    v_out(1) = -(0.90*v_in(1));
    p_out(1) = p_in(1) + v_out(1)*delta_t;
    % y-pos
    v_out(2) = v_in(2) + grav(2)*delta_t;
    p_out(2) = p_in(2) + v_out(2)*delta_t;
    % z-pos
    v_out(3) = v_in(3);
    p_out(3) = p_in(3) + v_out(3)*delta_t;
    count = count +1;
%If ball hits wall z = 0 in z-direction
elseif p_in(3) <= 0
    %x-pos
    v_out(1) = v_in(1);
    p_out(1) = p_in(1) + v_out(1)*delta_t;
    % y-pos
    v_out(2) = v_in(2) + grav(2)*delta_t;
    p_out(2) = p_in(2) + v_out(2)*delta_t;
    % z-pos
    p_in(3) = 0;
    v_out(3) = -(0.90*v_in(3));
    p_out(3) = p_in(3) + v_out(3)*delta_t;
    count = count +1;
     %If ball hits wall z = 300 in z-direction
elseif p_in(3) >= 300
    %x-pos
    v_out(1) = v_in(1);
    p_out(1) = p_in(1) + v_out(1)*delta_t;
    % y-pos
    v_out(2) = v_in(2) + grav(2)*delta_t;
    p_out(2) = p_in(2) + v_out(2)*delta_t;
    % z-pos
    p_in(3) = 300;
    v_out(3) = -(0.90*v_in(3))
    p_out(3) = p_in(3) + v_out(3)*delta_t;
    count = count +1;
%If no collision between ball and wall
else
    %x-pos
    v_out(1) = v_in(1);
```

```
        p_out(1) = p_in(1) + v_out(1)*delta_t;
        % y-pos
        v_out(2) = v_in(2) + grav(2)*delta_t;
        p_out(2) = p_in(2) + v_out(2)*delta_t;
        % z-pos
        v_out(3) = v_in(3);
        p_out(3) = p_in(3) + v_out(3)*delta_t;

    end

    p_out(4) = dt;
    v_out(4) = dt;

%     if count>50 - utanför
%         v_matrix(1:3,dt:2001)=0;
%         p_matrix(1:3,dt:2001)=0;
%
%     end

end
```

Bilaga D

index.html

```
<html>
<head>
<title>AMAZEBALLS</title>
<style>canvas { width: 100%; height: 100% }</style>
</head>
<body>
<script src="https://rawgithub.com/mrdoob/three.js/master/build/three.js"
<script src="Lib/three.min.js"></script>
<script src="Lib/TrackballControls.js"></script>
<!--<script src="Lib/stats.min.js"></script>-->
<script src="Lib/Detector.js"></script>

<script>

//Global variables
var balls = [];
var Nr_of_balls = 150;
       var primary_energy_loss = 0.9;
       var secondary_energy_loss = 0.95;
       var max_nr_of_bounces = 20;
       var time = 0;
       var radius = 1;
       var temp = 0.0;
       var change;

//Block 1 Set up Scene
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.

camera.position.x = 100;
        camera.position.y = 50;
        camera.position.z = 50;

var renderer = new THREE.WebGLRenderer({ antialias: false });
```

```
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
renderer.shadowMapEnabled = true;

//Mouse navigation
var controls = new THREE.TrackballControls( camera );
controls.rotateSpeed = 1.0;
controls.zoomSpeed = 1.2;
controls.panSpeed = 0.8;
controls.noZoom = false;
controls.noPan = false;
controls.staticMoving = true;
controls.dynamicDampingFactor = 0.3;
controls.addEventListener( 'change', render );
window.addEventListener( 'resize', onWindowResize, false );

        //Add lightning
var light = new THREE.SpotLight();
       light.position.set( 10, 100, 50 );
       scene.add(light);
       light.castShadow = true;

       //Add ground plane
       var planeGeo = new THREE.PlaneGeometry(100, 100, 10, 10);
var planeMat = new THREE.MeshLambertMaterial({color: 0xFFFFFF});
var plane = new THREE.Mesh(planeGeo, planeMat);
plane.rotation.x = -Math.PI/2;
plane.position.y = -50;

//Add wireframe for box
var cube = new THREE.Mesh(new THREE.CubeGeometry(100, 100, 100), new THRE
         wireframe: true,
         color: 'blue'}));
       scene.add(cube);
plane.receiveShadow = true;
scene.add(plane);

//Create balls and store in array
var geom = new THREE.SphereGeometry(radius,16,16);
    for (var i = 0; i < Nr_of_balls; i++) {
        var ball = {};
        ball.obj = new THREE.Mesh(
            geom,
            new THREE.MeshLambertMaterial({
                color: Math.floor(Math.random() * 0x1000000)
            })
        );
```

```
//Set initial position and velocity
        ball.x = 18*Math.random() - 9;
        ball.y = 18*Math.random() - 9;
        ball.z = 18*Math.random() - 9;
        ball.dx = Math.random();
        ball.dy = Math.random();
        ball.dz = Math.random();
        ball.count = 0;
        if (Math.random() < 0.5)
            ball.dx = -ball.dx;
        if (Math.random() < 0.5)
            ball.dy = -ball.dy;
        if (Math.random() < 0.5)
            ball.dz = -ball.dz;
        ball.obj.position.set( ball.x, ball.y, ball.z);
        scene.add(ball.obj);
        ball.obj.castShadow = true;
        ball.obj.recieveShadow = true;
        balls.push(ball);
}//End creation of balls

//Adjusted window size trigged by mouse navigation
function onWindowResize() {
camera.aspect = window.innerWidth / window.innerHeight;
camera.updateProjectionMatrix();
renderer.setSize( window.innerWidth, window.innerHeight );
controls.handleResize();
render();
}

//Navigation with arrow buttons
window.addEventListener('keydown', function (event){

     var code = event.keyCode;
     console.log(code);
     if (event.charCode && code == 0)
           code = event.charCode;

     switch(code) {
          case 37://, 65:
              console.log("left");
      camera.position.x -= 2;//((new THREE.Vector3(0, 1, 0)).normalize(),
              break;
          case 38://, 87:
              //console.log("up");
              camera.position.z -= 2;
              //camera.rotateOnAxis((new THREE.Vector3(1, 0, 0)).normaliz
              break;
```

```
            case 39://, 68:
                //console.log("right");
                //camera.rotateOnAxis((new THREE.Vector3(0, 1, 0)).normaliz
                camera.position.x += 2;
                break;
            case 40://, 83:
                //console.log("down");
                //camera.rotateOnAxis((new THREE.Vector3(1, 0, 0)).normaliz
                camera.position.z += 2;
                break;
     }
}, false); //End navigation with arrow buttons

//Update object's positions
function animate(t) {

        //Frame based instead of time based implementation at this stage
        dt = t-time;
        time = t;
        update_pos(dt)

        //Update frame
    controls.update();

        camera.lookAt(scene.position);
        renderer.render(scene, camera);
        window.requestAnimationFrame(animate, renderer.domElement);

    }; //End animate functon

    animate(new Date().getTime());

    //Update position of ball
function update_pos(dt){
var counter = 0;
     for (var i = 0; i < Nr_of_balls; i++) {
     if(balls[i].count>max_nr_of_bounces && balls[i].y < -48)
{
balls[i].y = -49;
balls[i].dy =  0;
     balls[i].dx =  0;
     balls[i].dz =  0;
}

  if(balls[i].y > -49){
balls[i].dy =  balls[i].dy - (9.82/1000);
}
balls[i].x = balls[i].x  + balls[i].dx;
```

```
balls[i].y = balls[i].y  + balls[i].dy;
balls[i].z = balls[i].z  + balls[i].dz;

wall_check(balls[i]);
while(ball_check(balls[i], i) == true && counter < 100)
{
counter++;
}

balls[i].obj.position.set( balls[i].x , balls[i].y , balls[i].z );
}
    }
    //Check collision with walls
    function wall_check(ball)
    {
     //If ball hits ground y-direction
    if(ball.y <= -49){
     ball.dy =  -ball.dy*primary_energy_loss;
     ball.dx =  ball.dx*secondary_energy_loss;
     ball.dz =  ball.dz*secondary_energy_loss;

     ball.y = -49;
     if(ball.x < -49)
     ball.x = -49;
     else if (ball.x > 49)
     ball.x = 49;
     if(ball.z < -49)
     ball.z = -49;
     else if (ball.z > 49)
     ball.z = 49;

ball.count++;

}
//If balls hit walls x-direction
else if(ball.x < -49 || ball.x > 49){
     ball.dy =  ball.dy*secondary_energy_loss;
     ball.dx =  -ball.dx*primary_energy_loss;
     ball.dz =  ball.dz*secondary_energy_loss;
     ball.dy =  ball.dy - 9.82/1000;

     if(ball.x < -49)
     ball.x = -49;
     else
     ball.x = 49;
     if(ball.z < -49)
     ball.z = -49;
     else if(ball.z > 49)
```

```
     ball.z = 49;

ball.count++;
}
//If balls hit walls z-direction
else if(ball.z < -49 || ball.z > 49){
     ball.dy =  ball.dy*secondary_energy_loss;
     ball.dx =  ball.dx*secondary_energy_loss;
     ball.dz =  -ball.dz*primary_energy_loss;
     ball.dy =  ball.dy - 9.82/1000;

     if(ball.z < -49)
     ball.z = -49;
     else
     ball.z = 49;

ball.count++;
}

    }//End of wall_check function

//Check collision with other balls
function ball_check(ball, i)
    {
     for (var j = 0; j < Nr_of_balls; j++) {
      if(i!=j && collision(ball, balls[j]) == true)
      {
      //Velocities and positions for the colliding balls
      var v1 = new THREE.Vector3( ball.dx, ball.dy, ball.dz );
      var v2 = new THREE.Vector3( balls[j].dx, balls[j].dy, balls[j].dz )
      var p1 = new THREE.Vector3( ball.x, ball.y, ball.z );
      var p2 = new THREE.Vector3( balls[j].x, balls[j].y, balls[j].z );

      var l1 = v1.length();
      var l2 = v2.length();
      l1 *= secondary_energy_loss;
      l2 *= secondary_energy_loss;

      //Calculate the normal and tangent vectors before the collision
      v1.normalize();
      v2.normalize();

      var normal = new THREE.Vector3( p1.x-p2.x, p1.y-p2.y, p1.z-p2.z );
var len = normal.length();
      normal.normalize();

      var tz = (-(normal.x*0.3) - (normal.y*0.3) ) / normal.z;
      var tangent = new THREE.Vector3( 0.3, 0.3, tz);
```

```
tangent.normalize();

//Normal and tangent projections for v1 and v2
var v1n = v1.projectOnVector(normal);
var v1t = v1.projectOnVector(tangent);
var v2n = v2.projectOnVector(normal);
var v2t = v2.projectOnVector(tangent);

//Normal velocity vectors for v1 and v2 after collision
var v2an = new THREE.Vector3((secondary_energy_loss*(v1n.x - v2n.x)
(secondary_energy_loss*(v1n.y - v2n.y) + v1n.y + v2n.y)/2,
(secondary_energy_loss*(v1n.z - v2n.z) + v1n.z + v2n.z)/2);
var v1an = new THREE.Vector3(v1n.x + v2n.x - v2an.x,
v1n.y + v2n.y - v2an.y,
v1n.z + v2n.z - v2an.z);
//Final velocity vectors after collision
var v1a = new THREE.Vector3( v1an.x + v1t.x, v1an.y + v1t.y, v1an.z
var v2a = new THREE.Vector3( v2an.x + v2t.x, v2an.y + v2t.y, v2an.z

v1a.setLength(l2);
v2a.setLength(l1);

//Assign the resulting velocoties
ball.dx = v1a.x;
ball.dy = v1a.y;
ball.dz = v1a.z;

balls[j].dx = v2a.x;
balls[j].dy = v2a.y;
balls[j].dz = v2a.z;

if(ball.count >= (max_nr_of_bounces-3))
ball.count = max_nr_of_bounces-3;
if(balls[j].count >= (max_nr_of_bounces-3))
balls[j].count = max_nr_of_bounces-3;

//Possible method to distance colliding balls from each other
/*if(len>0){
var coeff = (radius*2 - len);
console.log("Len: " + len + " <- len, coeff ->" + coeff);
ball.x += coeff*normal.x;
ball.y += coeff*normal.y;
ball.z += coeff*normal.z;

//  ball.x += coeff*v1an.x;
// ball.y += coeff*v1an.y;
// ball.z += coeff*v1an.z;
```

```
      //balls[j].x -= coeff*normal.x;
      //balls[j].y -= coeff*normal.y;
      //balls[j].z -= coeff*normal.z;

      // balls[j].x -= coeff*v2an.x;
      // balls[j].y -= coeff*v2an.y;
      // balls[j].z -= coeff*v2an.z;

      }
      else
      {
      //balls[j].x -= 1*normal.x;
      //balls[j].y -= 1*normal.y;
      //balls[j].z -= 1*normal.z;
      }*/

      ball.x += 1*normal.x;
      ball.y += 1*normal.y;
      ball.z += 1*normal.z;

      return true;

      } //End of if collision handling

  }//End of looping through all balls

  }//End of ball_check-function

    //Check distance between two balls center of mass
    function collision(ballA, ballB)
    {
     var distance = new THREE.Vector3( ballA.x-ballB.x, ballA.y-ballB.y,
     if(distance.length()<2.2*radius){
     //console.log(distance.length());
     return true;

     }
     else
     return false;
    }

    function degInRad(deg) {
    return deg * Math.PI / 180;
}

function render() {
renderer.render( scene, camera );
}
```

```
</script>
</body>
</html>
```