Developing Physics Game Engine Using JavaScript

A Project Report submitted in partial fulfillment of the requirements for the degree of Master of Computer Application of Department of Computer Science & Engineering Jadavpur University

By,

Srirupa Dey

Master of Computer Application-III
Class Roll No.: 001910503008
Registration No.:149871 of 2019-2020
Examination Roll No.: MCA226008

Under the supervision of

Prof. (Dr.) Chintan Kumar Mandal
Dr. Debajyoti Sarkar

Department of Computer Science & Engineering
Faculty of Engineering and Technology
Jadavpur University,Kolkata-700032,India

2022

FACULTY OF ENGINEERING & TECHNOLOGY
JADAVPUR UNIVERSITY

To Whom It May Concern

*I hereby recommend that the project entitled "***Developing Physics Game Engine Using JavaScript***" has been carried out by* ***SRIRUPA DEY*** *(Reg. No.: 149871 of 2019-20, Roll No: 001910503008) under my guidance and supervision and be accepted in partial fulfillment of the requirement for the degree of* ***MASTER of COMPUTER APPLICATION*** *in* ***DEPARTMENT of COMPUTER SCIENCE & ENGINEERING, JADAVPUR UNIVERSITY*** *during the academic year 2021-2022.*

Prof.(Dr.) Chintan Kumar Mandal
Project Supervisor
Dept. of Comp. Science & Engineering
Jadavpur University, Kolkata-700032

Prof.(Dr.) Anupam Sinha
Head of the Department
Dept. of Comp. Science & Engineering
Jadavpur University, Kolkata-700032

Dean, Faculty Council of Engineering & Technology
Jadavpur University, Kolkata-700032

CERTIFICATE OF APPROVAL

This is to certify that the project entitled "**Developing Physics Game Engine Using JavaScript**" is a bonafide record of work carried out by **SRIRUPA DEY** in fulfillment of the requirements for the award of the degree of ***Master of Computer Application*** in the ***Department of Computer Science and Engineering**, **Jadavpur University***. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose for which it has been submitted.

....	....
Signature of Examiner 1	Signature of Examiner 2
Date:	Date:

DECLARATION OF ORIGINALITY AND COMPLIANCE OF ACADEMIC PROJECT

This is to certify that the work in the project entitled *"***Developing Physics Game Engine Using JavaScript***"* submitted by **SRIRUPA DEY** is a record of an original research work carried out by her under the supervision and guidance of **Prof.(Dr.) Chintan Kumar Mandal and Dr. Debajyoti Sarkar** for the award of the degree of ***Master of Computer Application*** in the ***Department of Computer Science and Engineering***, ***Jadavpur University***, Kolkata-32. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Name	:	SRIRUPA DEY
Class Roll No.	:	001910503008
Registration No.	:	149871 of 2019-2020
Exam Roll No.	:	MCA226008
Project Title	:	Developing Physics Game Engine Using JavaScript

Signature :

ACKNOWLEDGEMENT

With my most sincere and gratitude, I would like to thank **Prof.(Dr.) Chintan Kumar Mandal**, *Department of Computer Science & Engineering,* my supervisor and **Dr. Debajyoti Sarkar** for their overwhelming support throughout the duration of the project. Their motivation always gave me the required inputs and momentum to continue with my work, without which the project work would not have taken its current shape. Their valuable suggestion and numerous discussions have always inspired new ways of thinking. I feel deeply honored that I got this opportunity to work under them.

I would like to express my sincere thanks to all my teachers for providing sound knowledge base and cooperation.

I would like to thank all the faculty members of the *Department of Computer Science & Engineering* of *Jadavpur University* for their continuous support.

Last, but not the least, I would like to thank my batch mates for staying by side when I need them.

SRIRUPA DEY

ROLL:001910503008

CONTENTS

Page No.

Abstract

The **Physics Game Engine** plays an important role in many types of games. While one can simply download a physics engine library and continue with the game engine development, building your own game engine from scratch has their advantage. It gives an underlying understanding of how the physics game engine works and also gives more control over the performance, flexibility, functionality, and usability of the engine itself.

In some cases, building your own 2D physics game engine, that furnishes an approximate simulation of systems such as collision detection, is a good choice especially while using **JavaScript**.

In this paper, we will discuss some of the physics simulations, define some basic properties and behaviours of rigid bodies, detect and solve rigid body collisions, and simulate collision responses after the collisions.

The physics game engine that has been built is accessible through a web browser that could be running on any operating system. The development process includes an Integrated Development Environment (IDE) that is Visual Studio Code for this project and a runtime web browser (Google Chrome) that is capable of hosting the running game engine.

Introduction

The first major step to building a physics game engine is to choose the features and the order of operations. The features may seem trivial, but features help to form the physics engine components and could indicate areas that might be difficult.

The most basic building blocks of a physics game engine are **velocity**, **acceleration**, and **gravity**. The positional attributes are used to drive the player, collision detection allows a player to reach the goal and move around the game. The collision types (Normal collisions and bouncy collisions) allow the game to have different ground types.

The physics game engines are complex for computations but simple to structure once the pattern is known. The engine object graph has four main components:

- Physics entity: It is an object or rigid body that the engine acts upon, and is the least active.
- Collision detector: It is concerned with finding the collisions between the entities.
- Collision solver: The collisions are passing over to the collision solver after checking in the collision detector.
- Physics engine: All the three components mentioned above together encompass the physics engine.

In Physics Game Engine development the speed of processors also plays an important role and it leads to designs for the game engines that give results in real-time. CPU utilization is less for simple solutions but when a complex solution is followed, it means compromising other features that may take up some of the CPU utilization of any game. Shaving off CPU usage from any of the game components, whenever possible, helps in the long run.

Game Engine

The game engine is the heart of a game and consists of five major components: the authoring tool, the physics engine, the rendering engine, the user interface (UI), and the audio engine. There are some clear differences between **games** and **Game engines**. The components of a game, its specified characters, the reason for the collision, real-world objects, and their behaviours, etc, are the components that make a real game, and on the other hand rendering, loading, animation, and collision detection between the objects, physics, inputs, GUI and AI are the primary different components of an engine.

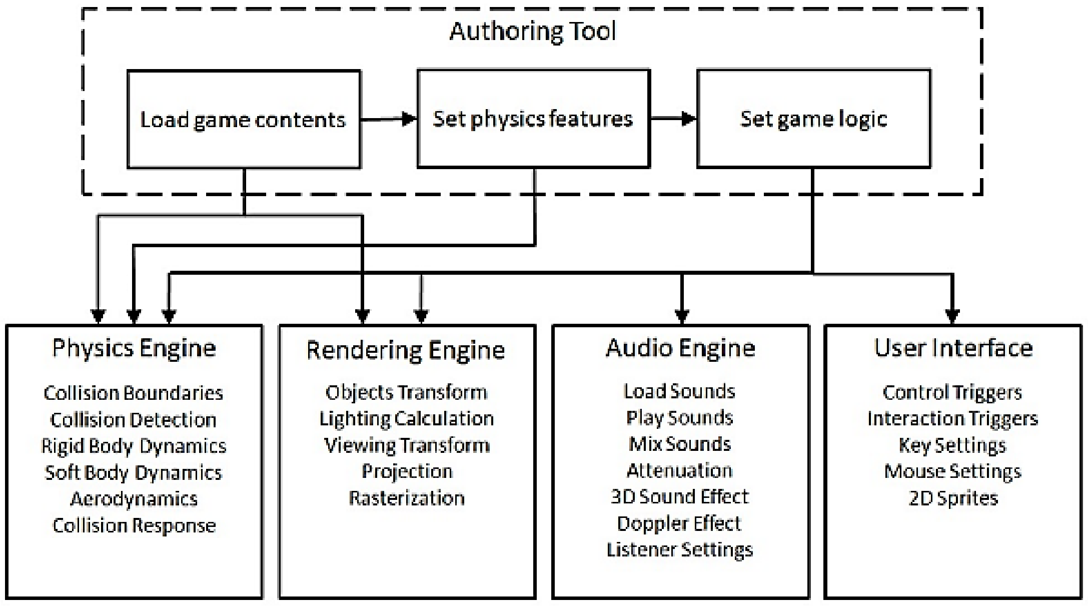


Fig: Architecture of Game Engine

The **Physics Engine** is responsible for the computation of physics. It is essentially a big calculator for dealing with the low-level details of physics simulations. It handles all the mathematical functions to simulate physics. A typical physics engine partitions the physical simulations into two phases: *collision detection* and *multibody dynamics*
Collision detection is the process that determines whether two objects touch each other or not, while *multibody dynamics* investigates the dynamical motions of objects that are connected with each other by joints or exist independently. Both are based on mathematical methods that are derived or modified from physical laws.

The **Rendering Engine** is used to draw a 3D world and portray it on the screen. The work of the rendering engine is described in three main parts. The first part is to

transform 3D data game objects into 2D screen space, which includes two transformations. In the first part of the transformation, 3D data in the game world is converted into 3D data in the coordinates of the view world using the engine that is associated with the camera in the game world. It decides the view that the player will be seen on the screen.
The second transformation converts 3D data of the view world into 2D data in the screen world and 2D data is made available to be drawn as pixels on the computer screen. This process is called *Projection*. And in the third part, the 2D data in the screen space is to be drawn onto the screen. This process is named *rasterization*. The *rasterization* process determines the displayed colours of the pixels and the calculation of the several shading effects, including materials, textures, lighting, transparency, and other special functional effects.

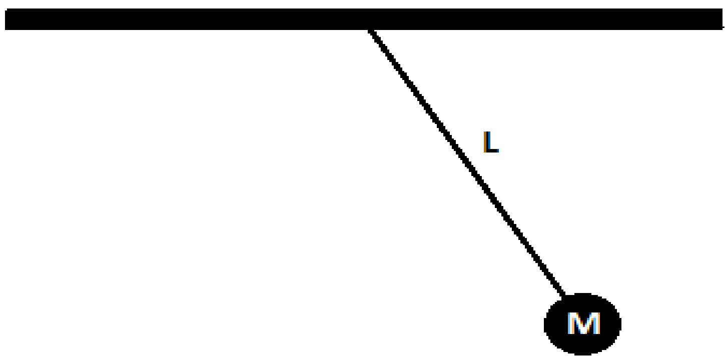
The function of the **Audio Engine** component is responsible for generating sounds while the game is running. Audio can be extremely important for a game's atmosphere and can enhance the quality of the game and also heighten the satisfaction of players.

The component that represents the interaction between game objects and players is called *User Interface (UI)* which can be separated into *invisible interfaces* and *visible interfaces*. The *invisible interfaces* have no on-screen features; for instance, control triggers using a mouse or a keyboard, move items, or communicate with NPCs(non-player characters) whereas the *visible interfaces* have on-screen features such as selective menus and any other information that is provided for the player on the screen.

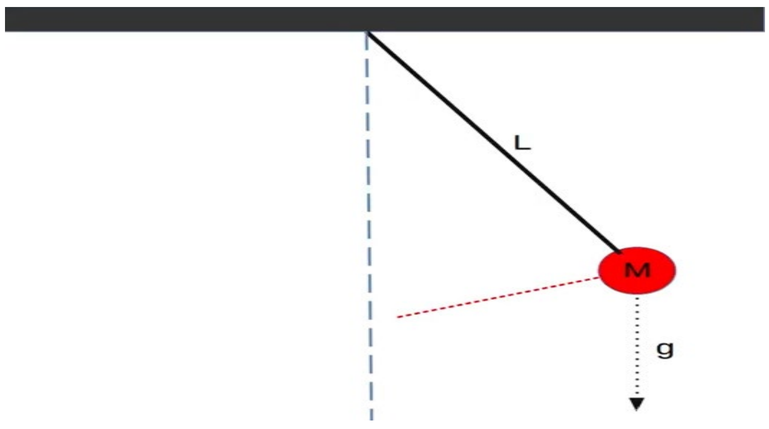
So, the **physics engine**, the **rendering engine**, the **audio engine**, and **UI** are the basic components are the basic building blocks for a realistic physics-based simulation or virtual environment, some frameworks of Integrated Development Environment such as XNA provide an integrated library to develop simulations or games utilizing these four components.
From 1989 the beginning of the simple game engines to recent 3D high-performance game engines, the goal of development remains constant i.e. to provide the developer a platform to make use of their unique ideas and create games into reality. As in engines the basic architecture, codes, and the middlewares are present, the developer only needs to tweak them as per requirement. The evolution of game engines is advancing towards more realistic and high-end games in various fields like physics sounds etc.

Simple Pendulum Simulation

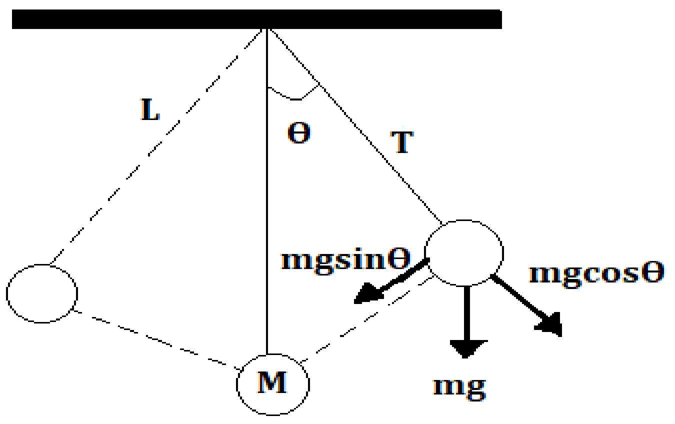
The pendulum is modelled as a point mass at the end of the massless rope. The vertical rope passing through fixed support is the mean point of the simple pendulum. That vertical distance between the point of suspension and the centre of the mass of the suspended body is known as the length of the simple pendulum. When displace to an initial angle and released, the pendulum will sway back and forth with the periodic motion.



The simple pendulum is a system that moves and sways in an oscillatory motion. The oscillatory motion is mainly driven by gravitational force **g** is constantly acting on the pendulum.
Given that a pendulum is displaced by angle θ, the formula for calculating the motion of the pendulum is generally given as,
Angular acceleration = g / L * sin (θ)
The simple pendulum is initially at rest in the vertical position. When the pendulum is displaced by angle θ and released, the force of gravity pulls it back towards its resting position. Its momentum causes it to overshoot and come to an angle -θ. The Restoring force along with the motion of the simple pendulum due to gravity is -**mgsinθ**.



Three forces are acting on the mass (M) -
i) Tension of the rope (T)
ii) Parallel component of the weight (**mgcos**θ)
iii) Perpendicular component of the weight (**mgsin**θ)
But as the mass (M) is attached the inextensible rope for the Simple Pendulum so restoring force is the Y-component of the mass.



<u>The time period of Simple Pendulum:</u>
This is defined as the time taken by the simple pendulum to finish one full oscillation. Interestingly, the period of the simple pendulum can be extended by increasing the fulcrum length (**L**) while taking the measurements from the pivot point of the suspension to the middle of the mass.
However, a period can be influenced mainly by the position of the pendulum concerning the Earth as the strength of the gravitational field is not uniform everywhere.
<u>Position of Simple Pendulum:</u>

Mean position is defined as the position of hanging mass (**M**) when it is at rest. As the mass is at its lowest point in this position, its potential energy is minimum and kinetic energy is maximum.
Extreme position is defined as the position when the hanging mass (**M**) has made maximum displacement concerning its **mean position**. As the mass is at its highest point in this position, the potential energy is maximum and the kinetic energy is minimum.
At any given point, during the course of oscillation, its total energy remains constant.
Here are the formulas for calculating the position-

```
b_location.x = Math.sin(theta) * Length + pivot_point.x;
b_location.y = Math.cos(theta) * Length + pivot_point.y;
```

Simulation of a ball on a Tilting Surface

A surface that can be tilted to make the ball roll due to gravity. The plane can be tilted up, down, to the left, and the right. At first the velocity of the ball will be zero because the ball is stationary.

Tilt the surface:

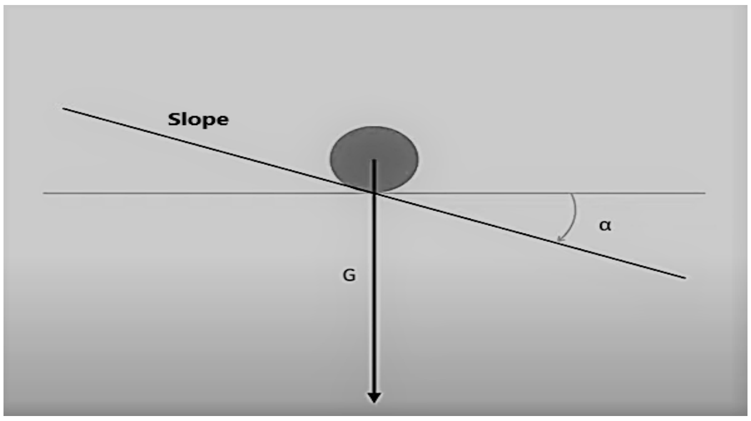
The surface can be tilted horizontally and vertically by moving a virtual joystick. This effect can be achieved by handling two event listeners, the mouse down and the mouse move. When the mouse down is pressed we register the initial position of the mouse. Then on every mouse movement, compare the new mouse position to these initial values.

We can turn the mouse movement into rotation values, and make it one to one so each pixel of mouse movement will result 1degree rotation of the surface. The mouse movement value can be multiplied with a constant to make it react quicker or slower. Here note that, the vertical movement gives the horizontal rotation and the horizontal movement gives the vertical rotation.

Moving the ball:

The ball has a position and a velocity. At the beginning the velocity is zero as the ball is stationary. But as we tilt the surface the acceleration is generated and that changes the velocity.

The surface is mostly tilted both horizontally and vertically, but we can handle these two axes separately to simplify the calculations. We can break up the velocities into horizontal and vertical components. The vertical component of the velocity will depend on the horizontal rotation of the surface and the horizontal component of the velocity will depend on the vertical rotation of the surface. The great thing about breaking up the velocity into horizontal and vertical components is that now we can calculate them independently. Let's tilt the surface vertically by 20 degrees. This will change the horizontal component of the ball's velocity.



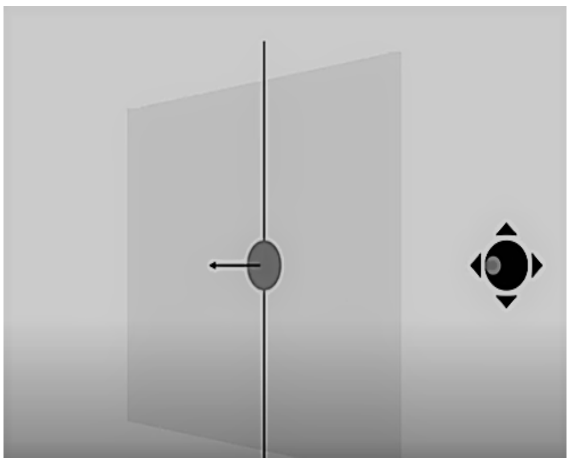
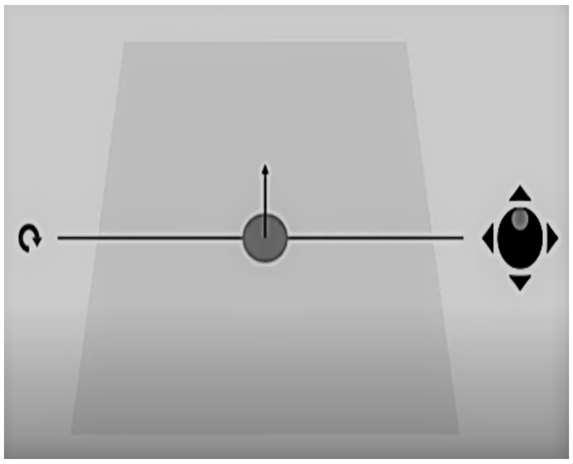
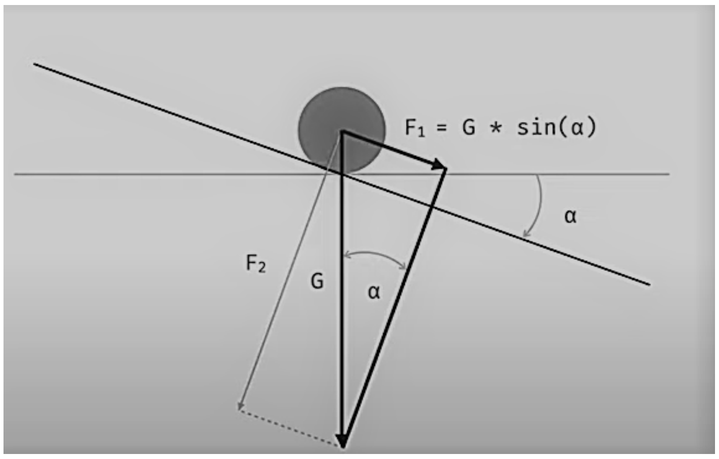
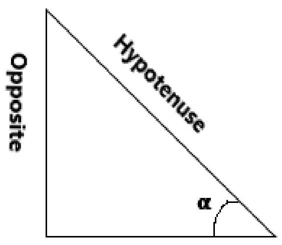



Fig: Horizontal and vertical tilt of the surface

At any point gravity pulls the ball down but it can't move that way as the surface is on the way. But it can go in parallel with the surface. To get the value of this movement we have to calculate how much force does gravity have on the ball in parallel with the slope.
We can break up gravity into two components as well. To a component that is in parallel with the slope, and to one that is diagonal to it. We can frame gravity in a rectangle where one side is parallel with the slope. The sum of these two forces (**F1, F2**) equals to gravity (**G**).
If we move by **F1** force then by **F2** force we do the same movement as we would have done with moving with **G** force. In other words, as gravity equals to the sum of these vectors we can replace gravity with these two vectors. Now we got two forces, the one in parallel with the surface will be the force moving our ball and the one diagonal to the surface we can either ignore or we can use it to calculate friction.
The gravity is splitting the rectangle into two right triangles that have the same angle as slope. We can multiply the sine value of that angle with gravity to get a parallel component of gravity.

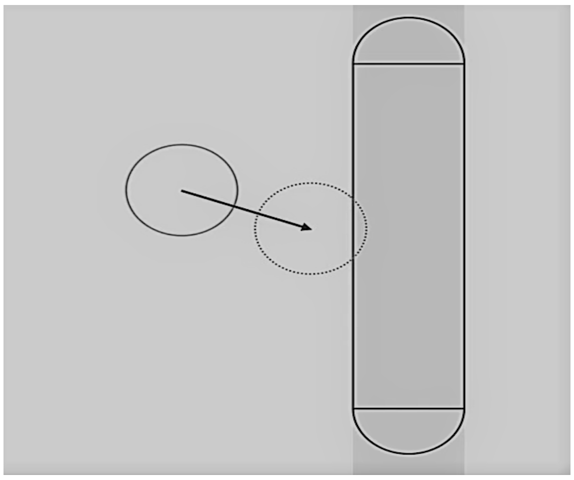




Opposite = Hypotenuse * sin(α)

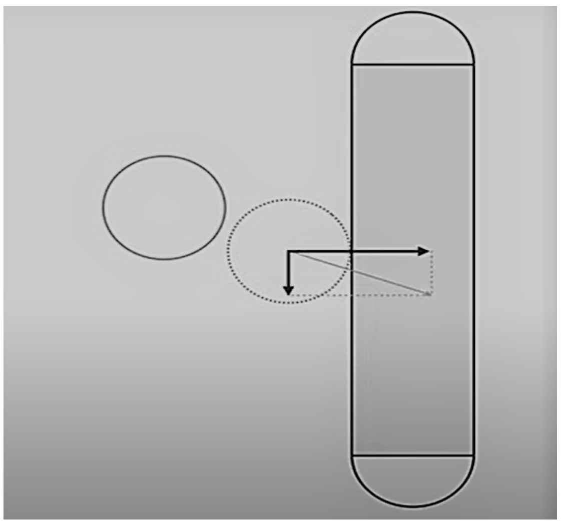
Hitting the wall:
Hitting a wall is simple at first but then it gets complicated in the loop in every cycle between calculating the next position of the ball and actually moving them to that position. We also go through each wall and see if the ball hits any of them.
Vertical and horizontal walls behave quite similarly, just the directions are changing. We calculate their starting and ending coordinates and test if the ball is about to get too close to any wall. Here we are using the future positions of the ball.

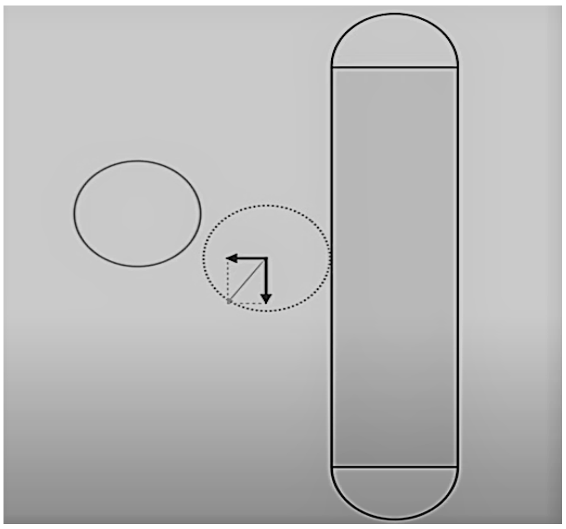
Calculate where the ball would be next if it wouldn't bump into a wall, then test if the position is okay. First check if the ball will be within the stripe of the wall, if it is, that doesn't mean that the ball will hit the wall. The Ball could be before and after it, but if the ball isn't in this stripe then it certainly is not going to hit the wall. We have to take into consideration the width of the wall and size of the ball. If any part is overlapping then we take further inspections.



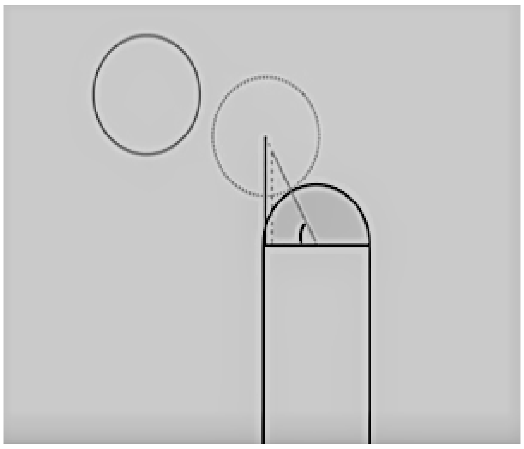
If a ball is overlapping with the main part of the wall, then simply shift the ball back till the ball is just next to the wall. Change the ball's position and also update the future position of the ball that it uses in collision detection.
Once we deal with position we have to deal with velocity. As we have seen, the ball's velocity consists of two components. Conveniently enough, one of these components is always diagonal to the wall. This component has to change as the ball can't move into the wall's direction anymore.
There are two options, one is we can either just set it to zero and the ball just keeps on rolling along the wall or we can just set it to be a proportion of its original value but opposite direction. This will result in bouncing effect, the ball keep bouncing on the wall till the component ultimately goes down to zero. If we keep moving the surface, then we still have an acceleration towards the wall, the ball still tries to go

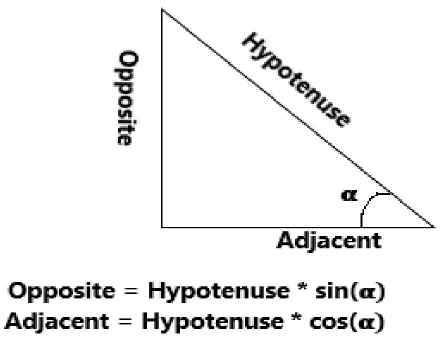
through it, but in every cycle we detect the hit, ignore the movement towards the wall, and let it roll along the wall.



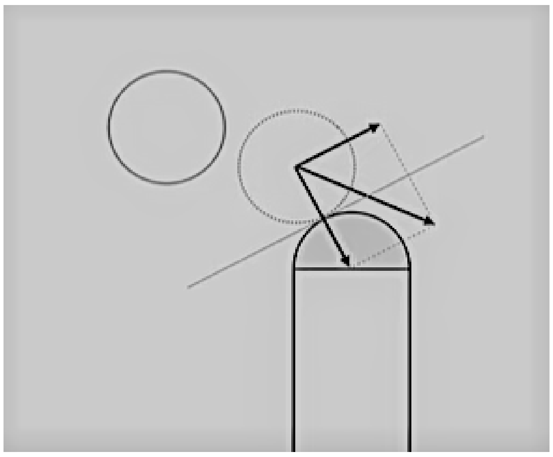
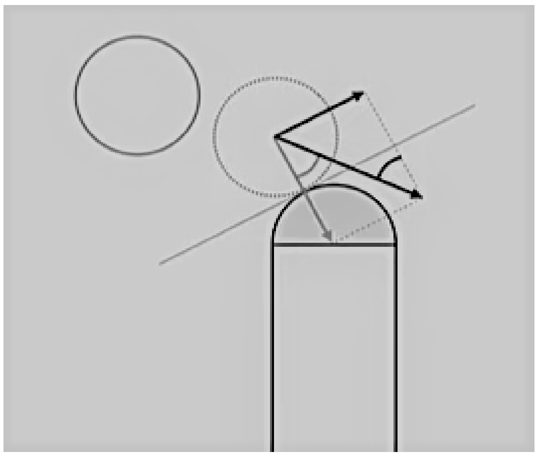


Hitting the wall cap:
In this case, we have to come up with a completely new velocity to simulate the ball rolling over the wall cap. The distance we are looking for is the one between the centre of the ball and the centre of the wall cap. If this distance is less than half the size of the ball summation of the radius of the wall cap, then we have a hit. We get the distance by using Pythagorean Theorem. If the distance is too close then first we push the ball back to the closest possible position where it would still not overlap with the wall cap.
First, we get the angle of impact with arctangent. Use the angle and distance the ball should be from the wall cap to calculate the horizontal and vertical delta from the centre of the wall cap. Then just add the values to the centre of the wall cap and get a new ball position that doesn't overlap with the wall anymore.

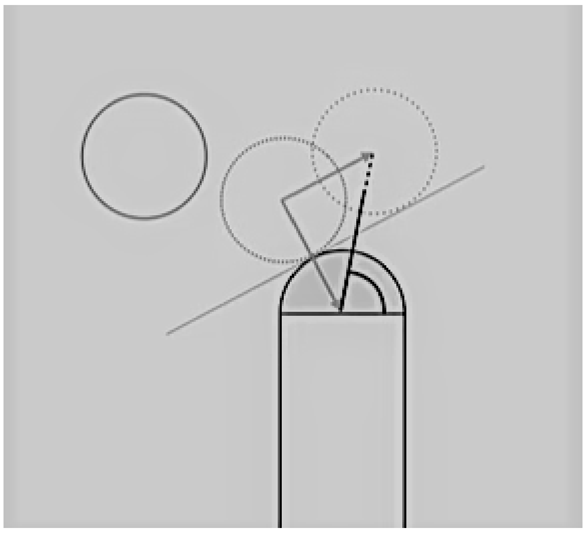
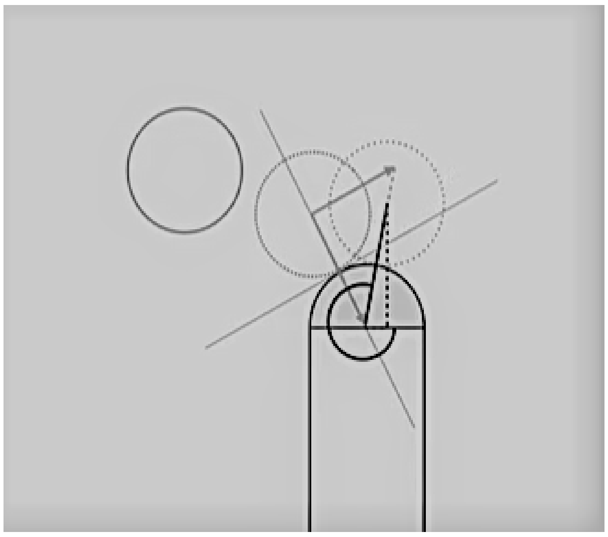



Opposite
Hypotenuse
α
Adjacent
Opposite = Hypotenuse * sin(α)
Adjacent = Hypotenuse * cos(α)

Now, need to calculate the angle of impact using arctangent under the hood and use the horizontal and vertical distance between the ball and the wall cap. After calculating the heading of the velocity and subtracting it from the angle of impact.

Instead of moving the ball away from the wall, have the rolling effect so we can pull the ball back towards the wall cap. Using the sine and cosine functions we get the horizontal and vertical positions offset of the end of velocity from the centre of the wall cap that is the new and final velocity vector. We add these delta values to the centre of the wall cap and we get the endpoint of the new velocity. Finally, we got a new ball position and a new velocity that will move the ball to a position that still doesn’t overlap the wall.

Rope Simulation

We model ropes as systems of molecules mannered to sit together in a shape that to some extent resembles the real thing more like a chain than a rope. We can visually connect the particles during the rendering process to make them look like a rope. We can use the visual ropes for dragging, pulling, lifting, and suspending other game objects.

Force and Motion

The mathematics of motion is called *classical mechanics*, which simply represents the masses as particles in space and accelerates the masses by forces as time passes. Now mass is defined as that has a position, velocity, and weight at any place where gravity exists. A mass that has a non-zero velocity in space moves in the direction of the velocity. Therefore the velocity is one reason for the change in position and another reason is the passing of time. Change in position depends on the fast movement of a mass and how much time has passed.

The velocity of a mass changes when some force acts on it. Velocity tends in the direction of the force that is inversely proportional to the mass and is proportional to the force.

Acceleration defines as the change of velocity per unit of time. The formula for acceleration:

acceleration = force / mass

So, when the force on a mass is more, its acceleration is more. And when the mass value of a mass is more, its acceleration becomes less.

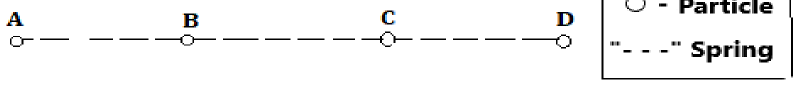
From this, we can get the famous equation:

force = mass * acceleration

Determining The Motion of Equation:

The motion of equation means determining forces that act in a physical setting. In the rope model, the forces will act on the particles that make up the rope.

The first force will be the spring tension between the particles.



The springs are the sources of the force between two particles, particle **A** is bounded to **B**, **B** to **C**, and **C** to **D**. So, formula for the spring force:

force = -k * x

Where **k** is a constant that represents the stiffness of the rope and **x** represents the distance of the mass from the point it is obliged to. So the above formula is the reason that causes the rope to wrinkle because there is force unless x is zero. Therefore all the particles of the rope would be pulled to each other until **x** becomes zero. The force should be zero when x was a positive value so somehow we have to maintain a constant length. The formula for this:

force = -k *(x-d)

k is a constant that represents the stiffness of the rope and **x** represents the distance of the mass from the point it is obliged to and **d** is a constant positive distance value that a rope stays steady.

So the rope would stretch when the value of **x** is more than **d**, and the rope would shrink when the value is less than **d**.

Now we need to introduce *friction*, if we do not consider the *friction factor* then the rope would never stop swinging. If the force can be applied to a mass in the opposite direction of the mass's movement, it makes the mass get slower. Therefore the friction force can be formulated in terms of velocity of mass:

friction force = -p * velocity

Where **p** is a constant to represent how much friction there is and **velocity** represents the velocity of the mass that is under the friction force. The friction force obtained by the relative velocities of the masses is added to the force of the rope.

To complete the simulation, we need to create an environment, that contains the rope and consider all the external forces acting on the rope. So we should have a gravitational field in this environment and masses experience the gravitational force.

A planer force is also added so that we can drag the rope on. The Equation of motion is extended. So, the formula for Gravitational force is:

force = (gravitational acceleration)*mass

Gravitation and air friction are added and will act on each particle on the rope. The planer force will act on every mass as well.

Here we have included the most frequently used concepts of advanced simulations that are true for physical simulations in game development as well.

Collision

The meeting of two objects where each exerts a force upon each other, causing the exchange of energy or momentum. As used in physics, the term collision does not necessarily imply actual contact. Collision problems are concerned with the relation of magnitudes and directions of velocities of colliding bodies after the collision to the velocity vectors of the bodies before the collision.
The conservation of momentum principle states that the system's total momentum is unchanged in the collision process when the only forces on the colliding bodies are those exerted by the bodies themselves. The velocity can change only during the collision process for a short interval period.

Classification:
The collision can be classified as **elastic** and **inelastic**.
In an **elastic** collision, mechanical energy is conserved, that is the total kinetic energy of the system of the particles after collision equals the total energy before collision.
In **perfectly elastic** collisions, both energy and momentum are preserved, which means the total momentum of the system stays the same as well as the energy due to the collision.
In **inelastic** collisions, only momentum is conserved. There is no loss of kinetic energy since it is transformed into heat, sound, etc. Most real-life collisions are inelastic collisions. However, the initial total energy is smaller than the total energy after collision sometimes can occur in classical mechanics. For example, a collision can cause an exploration that converts chemical energy into mechanical energy.

In **perfectly inelastic** collisions, the objects in the collision combine to form one connected mass this means two objects stick together after collision and move as one body. The linear momentum of the system remains preserved after this collision. One example of a perfectly inelastic collision is the ballistic pendulum. The following equation represents a perfectly inelastic collision between two objects A and B-

$$m_A v_{Ai} + m_B v_{Bi} = (m_A + m_B) v_f$$

Where v_f is the final velocity and m_A and m_B are the masses of the objects.

Separating Axis Theorem for Collision Detection

Introduction:

The separating Axis Theorem uses to detect whether two convex polygons produce a collision and it is used in a large number of game development. A separating Axis Algorithm can quickly determine whether it products a collision or not when the number of sides of a convex polygon is less. When the number of sides increases, the polygon becomes more complex and the greater the costs will be. So this algorithm has its strengths and weaknesses.

The Separating Axis Theorem by S. Gottschalk suggested that the main content is for two or more objects will collide, if one could find an axis so that these objects do not overlap on the projection of the axis, then we can believe that these objects do not intersect each other. Select theoretically axis is not fixed, as long as there is such an axis. However, non-convex polygons can be decomposed into several all convex polygons as the separating axis theorem applies to the convex polygons.

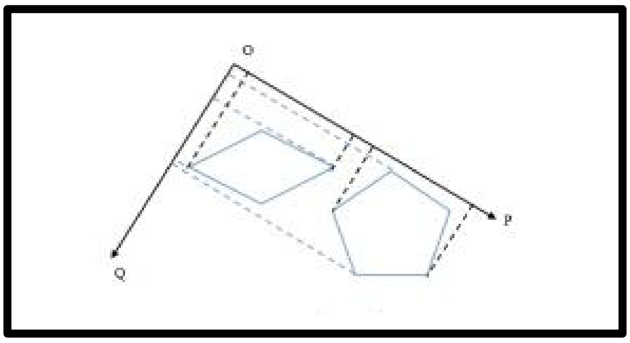


Fig: Separating Axis Theorem

For 2D, the normal of each edge of a convex polygon includes the possibility of the axis in all directions, so when we select the potential separating axis (PSA), generally we can choose the normal of edges of the convex polygon. Then the normal vector can be defined as:

$$[x\ y]\begin{bmatrix}-y\\ x\end{bmatrix}=0 \text{ or } [x\ y]\begin{bmatrix}y\\ x\end{bmatrix}=0$$

It just represents different directions of two normal vectors for the negative numbers, and when vertices are projected onto a potential separating axis, it does not need to be considered.

Separating Axis theorem collision detection algorithm is an optimistic collision detection algorithm, once you have found such a disjoint axis that detection will no longer be implemented, otherwise would have been implemented. And when the convex polygon becomes more complex, the corresponding overhead becomes greater. So for these flaws, led to the emergence of a large number of calculations, and there are many optimization schemes.
The main idea is to reduce the amount of computation for collision detection by reducing the number of Potential Separating Axis (PSA). For example, in two rectangles, because of their edges are parallel two by two, so we can manage the potential separating axis into two separating axis. Under the best of circumstances, the number of its maximum potential separating axis is two. So we just need to detect two or four axes to determine whether the collision.

Conclusion:

SAT algorithm is proposed based on a vector to the closest point where the potential separating axis (PSA) to determine whether two convex polygon collision. Most of the game engines have fairly complete physical systems, such as unity3D and cocos2d-x are relatively well-known domestic, which contain its collision detection algorithms but there are still some problems to be solved.

Circle-Circle Collision

The Circle-Circle Collision algorithm works by taking the centre points of two circles and ensuring the distance between the centre points is less than the two radii of the circles added together.

Edge Collision Detection:

Here it detects whether the bouncing circles have collided with the edges of the screen and if so it makes the circles bounce off the edge it collided with. If any of the circle's edges are located at or beyond an edge on the screen, then that's Collision. Once the collision is detected, need to reverse the velocity.

- **Bottom edge/Floor:** It is equal to the height of the screen.
- **Top edge/Ceiling:** It is located at y:0(top left corner is x:0,y:0).
- **Left edge:** It is located at x:0.
- **Right edge:** It is equal to the width of the screen.

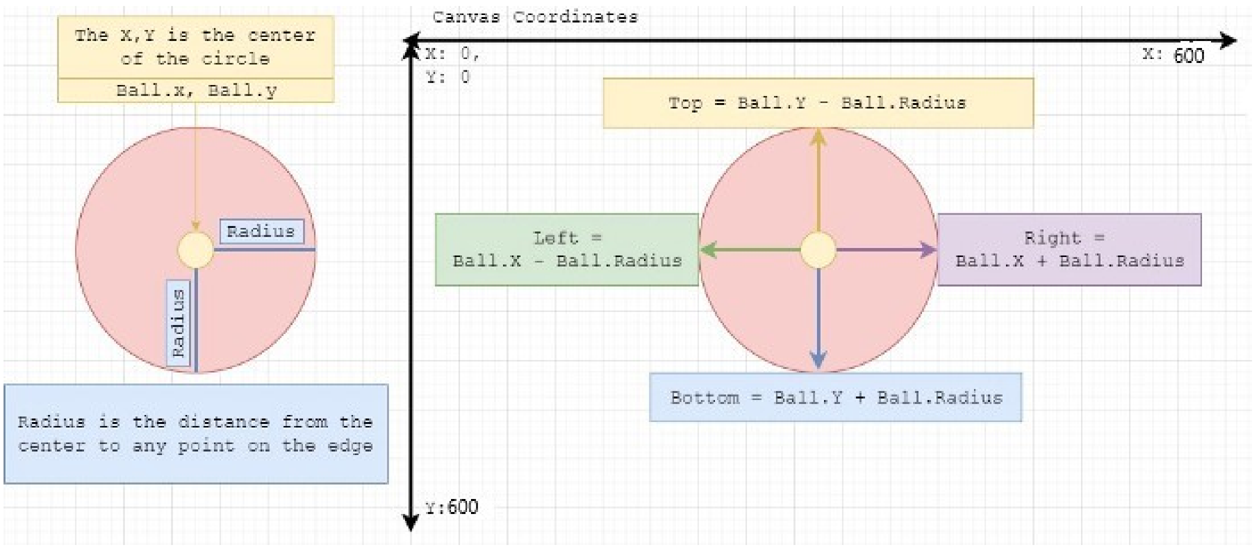
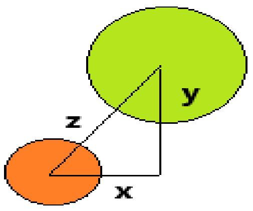

The X,Y is the center of the circle
Ball.x, Ball.y
Radius
Radius
Radius is the distance from the center to any point on the edge
Canvas Coordinates
X: 0, Y: 0
X: 600
Y:600
Top = Ball.Y - Ball.Radius
Left = Ball.X - Ball.Radius
Right = Ball.X + Ball.Radius
Bottom = Ball.Y + Ball.Radius

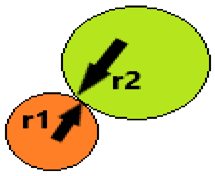
Fig: Circle Edge Diagram

Circle to Circle Collision Detection:

The basic way to determine whether two circles have collided or not is by determining whether or not two circles intersect or overlap. Conceptually, to perform a circle collision compare the distance from the centres of the two circles and check if

it is greater than the sum of the two radii of the circles. If this condition is true then a collision has been detected between two circles.
With the help of Trigonometry, one can determine the distance between the two points. Pythagoras's theorem, $\mathbf{x^2 + y^2 = z^2}$ to figure out the distance between two circles' centres.





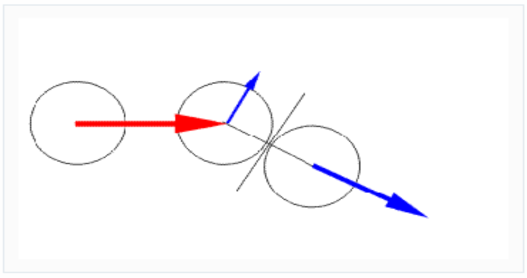
Here we get the lengths of x and y from the coordinates of both the circles. So, it's trivial to work out:

X = ball1.x - ball2.x; Y =ball1.y - ball2.y;

Now find the value of **z** with the help of algebraic rearrangement **z = Math.sqrt(**$\mathbf{x^2 + y^2}$**).**After that check this value against the sum of radii of the two circles (**r1 + r2**) whether it's less than or equal to the sum.
We will observe that no matter at what angle the circles are touching if they are touching then the value of **z** would be equal to the sum of radii of the circles.

<u>Bouncing apart</u>:

In this part, Elastic Collision comes into the picture. Two similar objects are travelling toward each other with equal speed, and they collide, bouncing off each other with no loss in speed. This is a perfectly elastic collision because no mechanical energy has been lost.



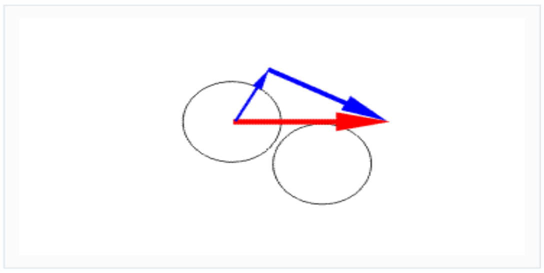


Fig: Elastic Collision

The direction each circle takes after a collision can be worked out by considering the most important factor mass. The mass of the circles would be proportional to either their area or volume.
For calculating the change in x velocity of the first circle uses the following formula:

final_velocityx = ball 1.speed.x * (ball1.mass - ball2.mass) + (2*ball2.mass * ball 2.speed.x))

Need to calculate the changes in x velocities and y velocities for both the circles after the collision occurs. The formulas are-

final_velocityx_1 = ball 1.speed.x * (ball1.mass - ball2.mass) + (2*ball2.mass * ball 2.speed.x))/(ball1.mass+ball2.mass);

final_velocityy_1 = ball 1.speed.y * (ball1.mass - ball2.mass) + (2*ball2.mass * ball 2.speed.y))/(ball1.mass+ball2.mass);

Now if the circles are heading together then the first collision will reverse their directions so they move apart and the second collision will reverse their direction again, causing them to move together. Hence we move each circle by the new velocity. This means, in principle that the circles should move apart by the same amount of speed that they moved together placing them a distance apart equal to the frame before they collided.

Conclusion

Physics game engines come with several parts to make them work. The entity, collision detector, collision solver, and engine core are the most important parts. Implementation of a simple solution for a game can work to your advantage when you build your own physics engine. If you build a physics engine where computational complexity is important then a robust solution is always more desirable and should consider a smoke-and-mirror approach.
By having the components working in accordance, and managing their own specific jobs, simple physics can be accomplished by selecting shapes and algorithms carefully. A smaller subset of the more robust physics engine implementations can be built. The more powerful physics engines can always help you to determine the best approach for your own game.

References

1. *History and Comparative Study of Modern Game Engines*, Partha Sarathi Paul, Surajit Goon, Abhishek Bhattacharya, International Journal of Advanced Computer and Mathematical Sciences, ISSN 2230-9624. Vol 3, Issue 2, 2012, pp 245-249.
2. Gottschalk, S, *Separating Axis Theorem Technical Report TR96-024*, Department of CS, University of North Carolina,1996
3. Fritzkowski P., Kaminski H., *A discrete model of a rope with bending stiffness or viscous damping*. Acta Mechanica Sinica, 27(1), 2011, 108-113.
4. *Collision Physics* by Paul W.Schmidt, the publication year 2014.
5. *The Research of Collision Detection Algorithm Based on Separating axis Theorem* by Cheng Liang, Xiaojian Liu, ISSN:1813-4890.
6. *The Development of the Simulation Modeling System and Modeling Ability Evaluation* by Jeng-Fung Hung, Jen-Chin Lin.NKN University, December 2009.
7. *Simulation of Simple Pendulum,* by Abdalftah Elbori,Ltfei Abdalsmd, Vol. 6 Issue 4, April 2017, PP.33-38.