Graphics and Visualisation

Modelling Maze Game Behavior in Physical World

Group Members:

Wilson Lim	1002810
Tay Tzu Shieh	1002946

Problem Statement

To simulate object collision behaviours in the physical world through modelling the interaction between the maze wall and the ball. The project aims to utilise computer graphics techniques to model the effects of gravitational forces acted on an object such as acceleration and bouncing off effect after collision. And discover methods for efficient collision detection computation.

Approach

Project Gravity Force onto Maze Plane and Ball Motion

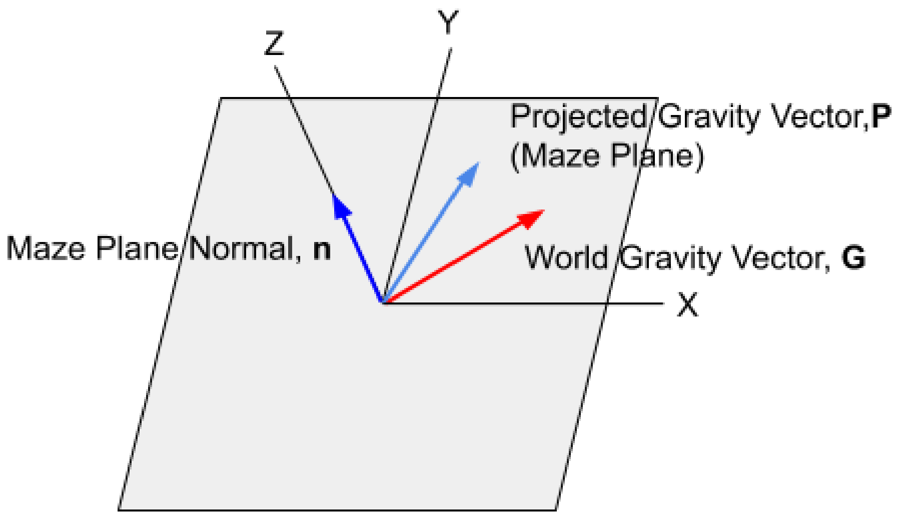


Figure 1: Vectors on the maze plane

The world gravity vector **G,** is fixed and pointing in the -Z direction but the ball is constrained to move on the plane of the maze. To compute the ball velocity under the effect of world gravity, it is required to project **G** onto the maze plane which resolves the gravity into 2D space referenced to the maze plane. The projected gravity vector, **P,** is then used to compute the new velocity of the ball. The following illustrate the process to compute the ball's new velocity vector under the effect of gravity.

1. Convert the maze plane normal vector to the world coordinate system by multiplying it with the model view transformation matrix.

$$MazeNormal_{world\ coordinate} = ModelViewTransMatrix \times <0,\ 0,\ 1,\ 0>$$

2. Calculate the projected gravity (acceleration) vector in world coordinate system using formula below:

$$GravityVector = <0,\ 0,\ -9.81>$$

$$ProjectedGravityVector = GravityVector - (GravityVector \bullet MazeNormal) * MazeNormal$$

3. Convert the projected gravity vector to the object (maze) coordinate system by multiplying it with the inverse of the model view transformation matrix.

$$ProjectedGravity_{object\ coordinate} = ModelViewTransMatrix^{-1} \times ProjectedGravity_{world\ coordinate}$$

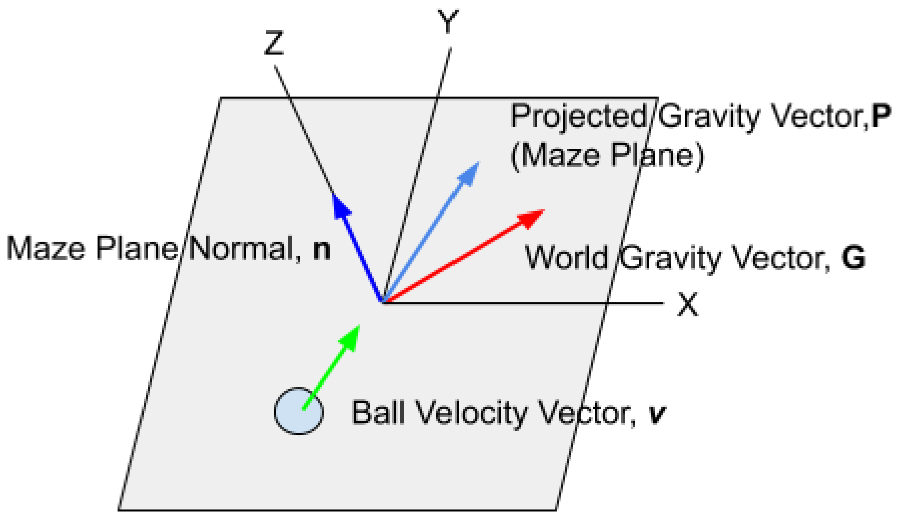

Z
Y
Projected Gravity Vector, **P**
(Maze Plane)
Maze Plane Normal, **n**
World Gravity Vector, **G**
X
Ball Velocity Vector, ***v***

Figure 2: Ball velocity vector(green) changes under the effect of gravity vector

4. The projected gravity vector is then multiplied with a fixed time step of 0.03s to get the increment or decrement velocity and added to the ball velocity.

$$FinalVelocity = InitialVelocity + Acceleration * Timestep$$

$$BallFinalVelocityVector = BallInitialVelocityVector + ProjectedGravityVector * 0.03$$

Efficient Collision Detection

The naive way of detecting the collision is to use the object position and check for collision for every face in the model space but this approach incurs excessive and redundant collision check computation. The complexity for collision detection of each rendering cycle would be O(n) and given the number of planes required to check at each cycle is large in this project multiplied by the fact this is checked every frame, it would greatly impact the program performance. By using a sorted data structure and binary search algorithm, the collision detection computation complexity is reduced to O(log n).

Further optimisation can be done by exploiting the fact that the plane normals are either in the x or y direction. Given a ball velocity, we know collision does not happen between the ball and planes with its normal in the same direction. This further reduces the required checking by at least half.

To achieve the efficient collision detection, a data structure was used to support the required operation in the below steps.

1. Data structure

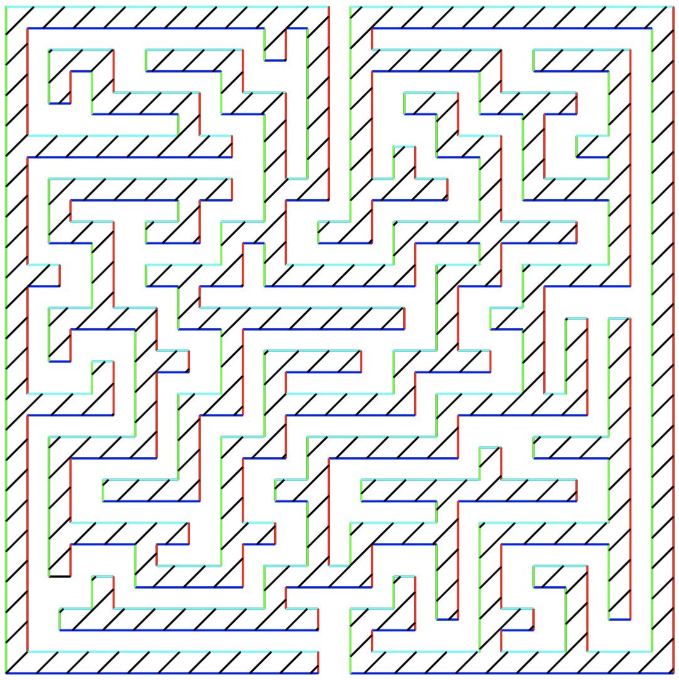


Figure 3: Maze wall color code based on plane normal vector

   a. Group the maze wall plane based on the plane normal vector
      i. Red: <1, 0>
      ii. Green: <-1, 0>
      iii. Cyan: <0, 1>
      iv. Blue: <0, -1>

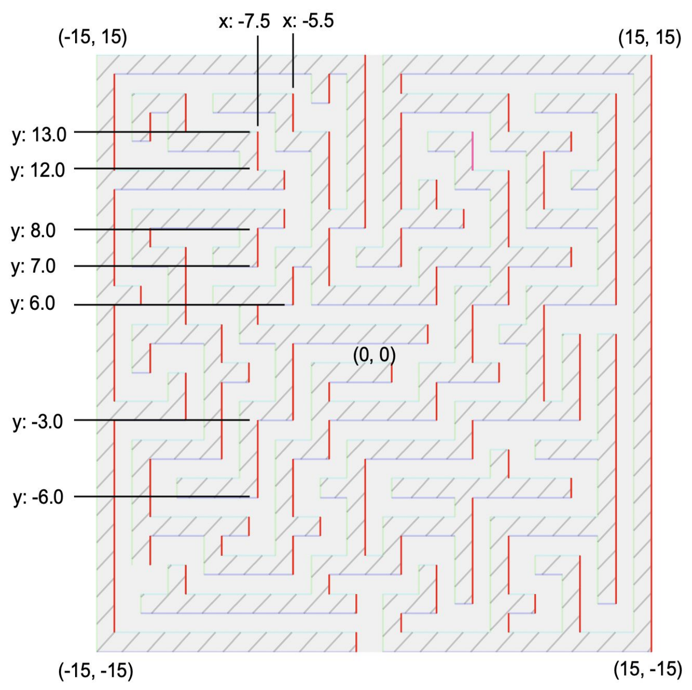


Figure 4: Distance from the origin and start and end position of the Y planes

Plane distance from origin	Plane start position	Plane end position
...	...	...
-7.5	-6.0	-3.0
-7.5	7.0	8.0
-7.5	12.0	13.0
-5.5	6.0	7.0
...	...	...

Figure 5: Subset of sorted array of the positive facing Y planes

b. Sort each group by the distance from the maze origin and the plane start and end position. Below shows the illustration for grouping the positive facing Y-plane and sorting them by the distance of the plane from the origin followed by the plane's start position and end position.
   Complexity: O(n*log n) [comparison sort]

Using the data structure, collision detection can be done as follows.

2. Finding the potential collision planes using the binary search method on the plane group determined by the ball movement direction. The plane group to search with has a normal vector in the opposite direction for the X and Y component of the ball movement vector. Below illustrate the process of finding the collision plane.

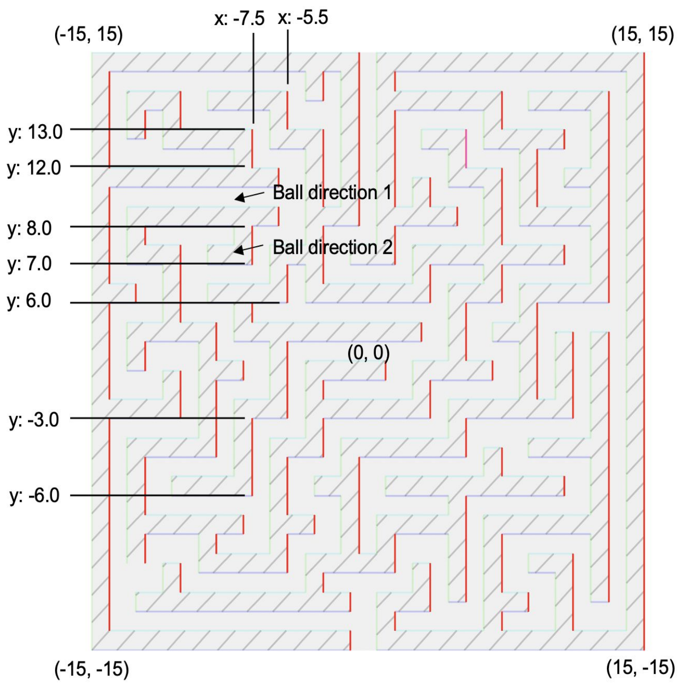


Figure 6: Different ball directions in the maze

From the diagram above, the plane groups to search with given the two ball directions will be the plane group with normal vector in positive X direction (<1, 0>) and normal vector in positive Y direction (<0, 1>) since the ball direction is facing toward negative X and negative Y direction. The binary search is then performed on each group to find the collide plane by checking if the ball radius vector crossed the current plane and within the plane range.

	Plane distance from origin	Plane start position	Plane end position
	-14.0	-14.0	-3.0
	...	...	...
mid 2nd →	-7.5	-6.0	-3.0
mid 4th →	-7.5	7.0	8.0
mid 3rd →	-7.5	12.0	13.0
mid 1st →	-5.5	6.0	7.0
	...	...	...
	15.0	-15.0	15.0

Figure 7: Illustration of the binary search mid pointer

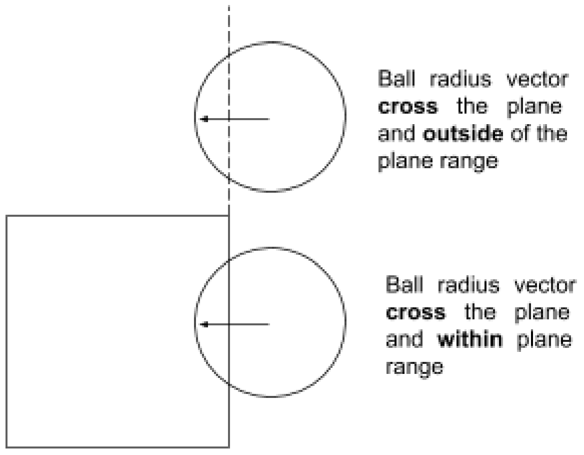

Ball radius vector **cross** the plane and **outside** of the plane range
Ball radius vector **cross** the plane and **within** plane range

Figure 8: Conditions for finding potential collide plane

Since the search range using binary search is reduced by half for each iteration, the search complexity is O(log n) which is much better than looping through all the planes in a group that has the complexity of O(n) in the worst case. The binary search works by using low and high pointers where the low pointer first points to the start of the array and the high pointer points at the end of the array. At each iteration, it will compute the mid index using the low and high pointer index and check if the value of the plane at mid index contains the given ray. If the plane at the mid index is greater than the given ray position, move the high pointer to the mid index else move the low pointer to the mid index. The binary search iteration stops once it finds the plane or the low pointer passes the high pointer index.

Collision Computation

With collision detected, collision computation is done in the steps below.

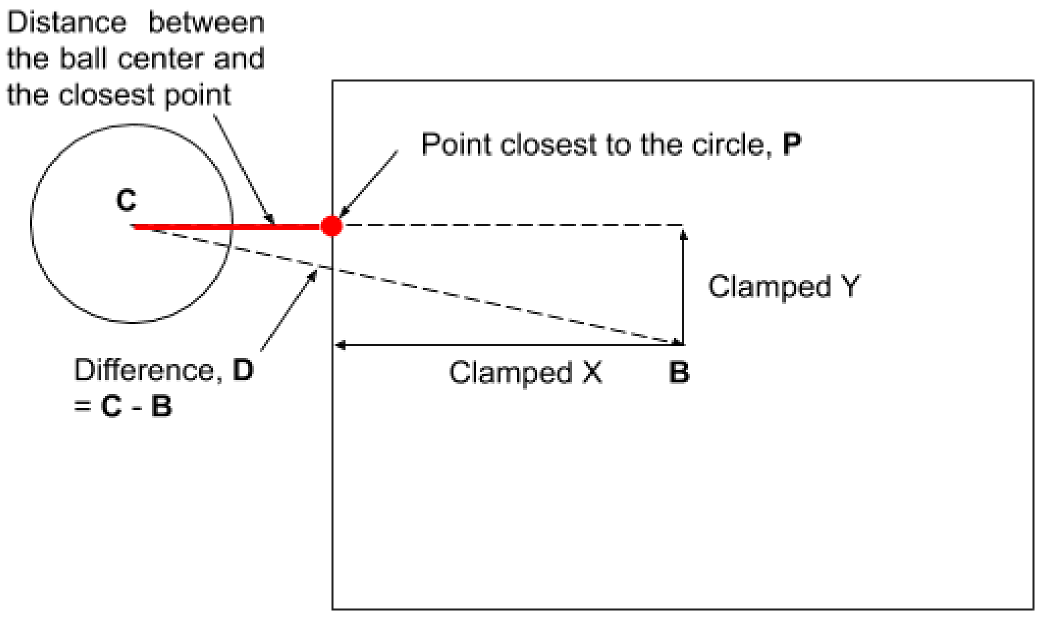

Distance between the ball center and the closest point
C
Point closest to the circle, P
Clamped Y
Difference, D = C - B
Clamped X
B

Figure 9: Finding the closest point from the plane to the ball

1. Calculate the point, **P** closest to the ball by clamping the difference between the ball center and the plane center, **D** to the plane.
2. Calculate the distance between the ball center and the closest point. The ball collides with the plane if the distance is less than the ball radius.

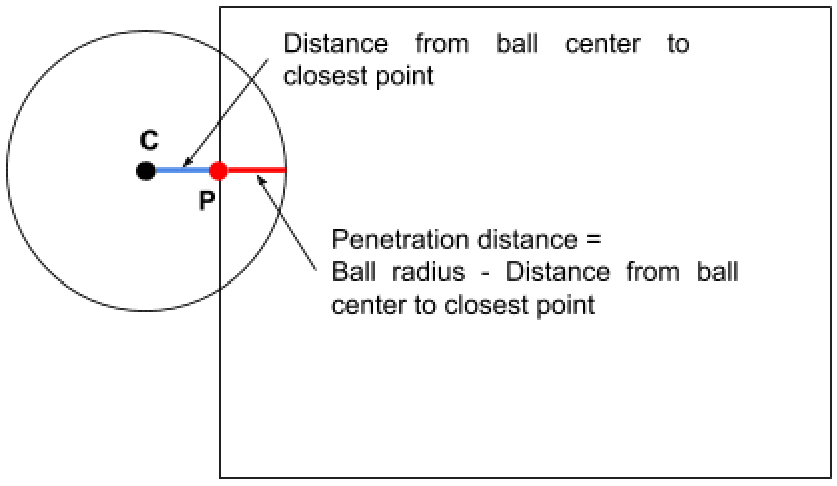

Distance from ball center to closest point
C
P
Penetration distance = Ball radius - Distance from ball center to closest point

Figure 10: Ball penetration distance

3. Calculate the penetration distance by calculating the difference between the ball radius and the distance between the ball center to the closest point. The penetration distance is used to reposition the ball away from the plane after collision.

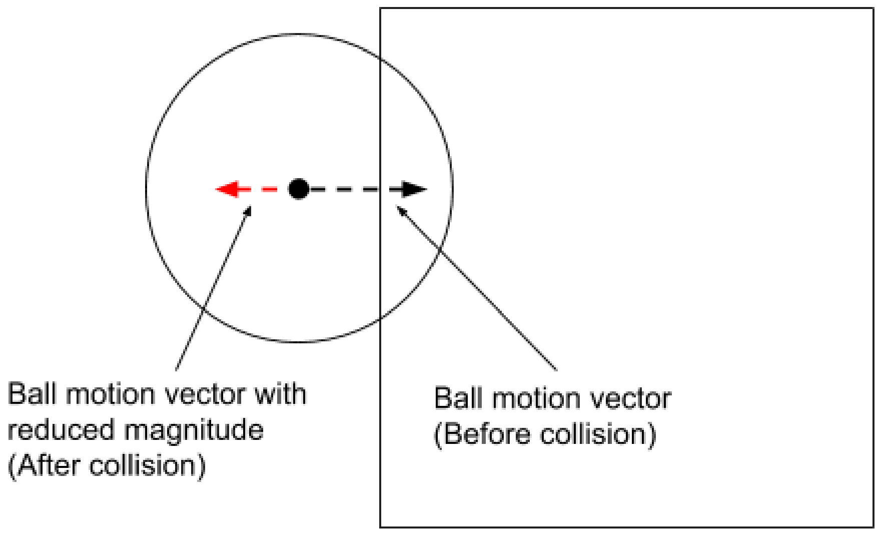


Figure 11: Ball motion vector before and after collision

4. The ball movement direction vector is reflected based on the normal vector of the collided plane and multiplied with the ball's coefficient of restitution which is 0.8 in this project. (Coefficient of restitution is a constant for a material that denotes the ratio of final velocity over initial velocity) For example, if the ball moves in the positive X direction collides with the plane, the X component of its movement direction vector will be reversed.
5. When the ball's displacement is greater than its radius, the collision detection will be computed in 0.02 travel distance step until it reaches the computed displacement. This approach is to prevent the ball from "phasing through" the wall where the ball's next position goes past the wall plane when it reaches a very high velocity and no collision is detected.

Texture Mapping

To improve the visual experience, texture is applied to the maze model, with separate texture for the wall, floor and top of the maze. The different textures bring about a contrast in the different surfaces which allows the user to quickly and clearly identify the different surfaces.

Figure 12: Image file for wall texture

The approach taken is to have separate image files, where each image file contains the texture for a surface, as seen in figure 12. The texture is stored in a bitmap image file type, with 24 bit colour which assigns 8 bits for red, green and blue colour each. This file format is chosen because it is supported by many image editing software and the image can be stored without compression, making it easier to handle.

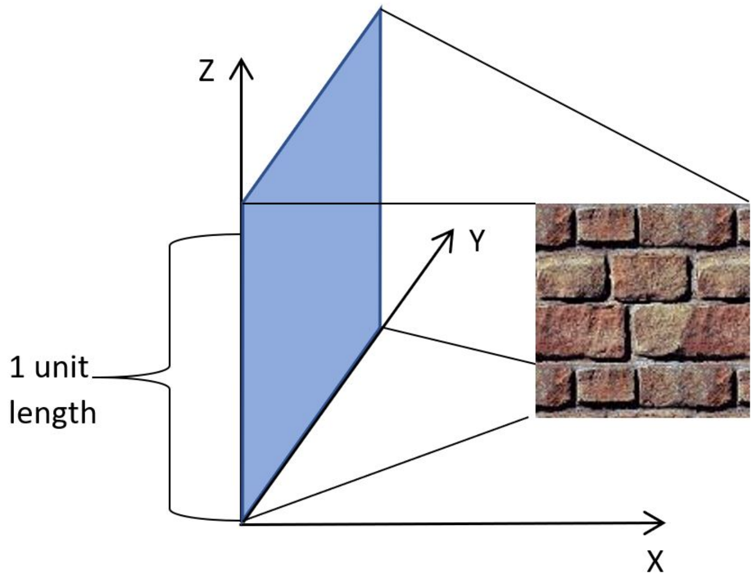


Figure 13: Mapping of wall texture

In the maze initialisation, the texture ID to be used is determined based on the face normal and position. The face normal would allow one to know if the direction it faces and the coordinate is used to determine if its the top or floor of the maze. Based on the texture ID, the mapping from the face coordinate to the texture is carried out differently.

For wall faces, their face normal is in the x or y direction, where the y coordinate is mapped to the x coordinate of the texture when the face normal is in the x direction, while x coordinate is used for faces with the normal in the y direction. The z coordinate of the face maps to the y coordinate of the texture, as seen in figure 13.

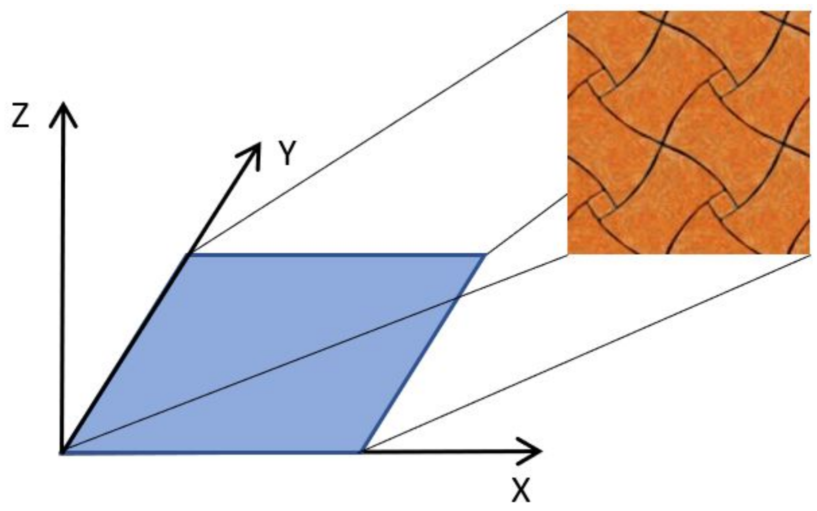


Figure 14: Mapping of floor texture

The maze floor and top faces can be determined by checking the z coordinate, where the floor is close to zero and the top is more than zero. The top and floor faces are mapped similarly, with the x and y face coordinate mapped to the x and y texture coordinate respectively, as seen in figure 14.

During texture initialisation, each texture is loaded and the texture object ID is stored. When rendering, the particular texture is bound to OpenGL based on the texture ID before applying texture coordinates. Since the texture is fixed upon loading, there is no need to reinitialise and read the texture file on the go. The texture can be loaded and stored in memory and accessed where required. This saves computation in avoiding the need to continuously read the texture file and parse it every frame.

Implementation

Platform

The modelling of the maze game is done using the GLUT library which is written in C. It is a OpenGL utility toolkit that implements a simple windowing application programming interface (API) for OpenGL. It provides features such as window and user input events and image and 3D model rendering. The maze game is written using C++ to take advantage of the high efficiency of the language which could improve the game responsiveness and better user experiences.

User Interface

The user is allowed to rotate the maze by holding down the left mouse button and moving the mouse using an arc ball rotation method to move the ball. Rotation motion is limited to ensure the user does not over rotate the maze. The “B” and “S” keyboard key are used for zooming in and out of the maze. The “T” keyboard key is used to reset the maze orientation. The “R” keyboard key is used to reset the game. Key inputs are also indicated within the game for user’s reference.

Maze Data

The maze layout is generated using an online tool “http://www.mazegenerator.net”. The layout is then converted into 3D model by extruding the layout using 3D modelling software which is then exported as OBJ file format. The maze 3D model is then loaded into the program by parsing the OBJ 3D model file which defines the vertex position of each face in the model.

Results

Basic Maze Model

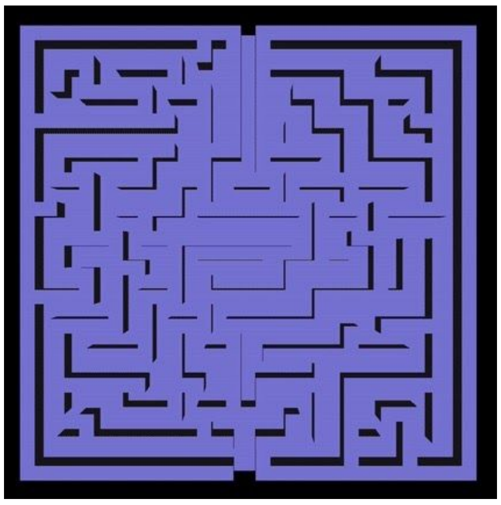


Figure 15: Maze model rendered in the perspective view

In the first step, the maze model is loaded into the world and scaling is done to fit the entire maze into the view of the user, as seen in figure 15.

Gravitational Force on the Maze Model

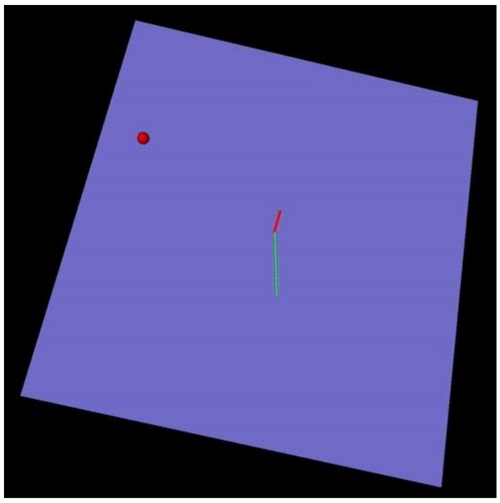


Figure 16: Rendering the ball motion under the gravitational force

The maze model structure is hidden to show the ball and the projected gravitational force onto the maze plane (green ray) as seen in figure 16. The ball is moving in the direction of the projected gravitational force as the gravitational force acts on the ball. This allowed us to visualise the projection to verify the projection direction.

Maze Model With Ball

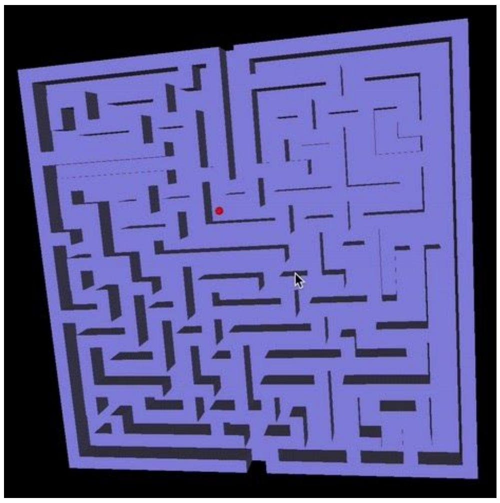


Figure 17: Ball collide with the maze wall

In the next step, ball collision was implemented with the maze wall. The ball moves while constrained to the maze walls and bounces off when colliding with the maze wall.

Maze Model With Texture

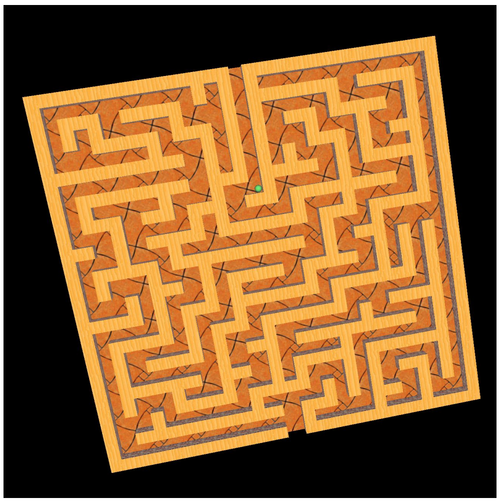
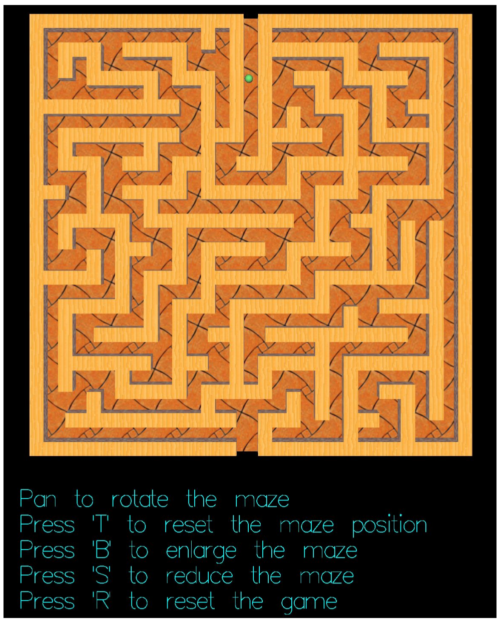


Figure 18: Maze with different texture for the wall and the floor

Texture was added to the maze. The texture helps the user to differentiate the different types of surfaces, as seen in figure 18.

Gameplay View



Pan to rotate the maze
Press 'T' to reset the maze position
Press 'B' to enlarge the maze
Press 'S' to reduce the maze
Press 'R' to reset the game

Figure 19: Game instructions displayed at the bottom for user reference

Basic instructions were added to the screen so that the user knows the functionalities of the supported key presses in figure 19.

Game Performance

Benchmarking system:
Central processing unit (CPU): Intel i7 7700HQ
Graphics processing unit (GPU): Nvidia GTX1060 6gb (mobile) and Intel HD630
Ram: 2x8gb 2133Mhz

The test is carried out with a laptop with the above hardware. The frame rate and usage is logged as the game ran while the user interacted with the game. To reduce interference between other applications, unnecessary applications are closed before the start of the test. Some basic applications remained running in the background, such as internet browser applications and messaging applications. This would provide a reasonable gauge in the usability under a typical situation.

To account for older or weaker systems, the system was limited in its processing capability by the use of Intel Extreme Tuning Utility (XTU) to limit both CPU and onboard graphics clock rate. The game was also run using just the onboard graphics card to ensure that systems without dedicated graphics cards can also run the game smoothly. Frame rates and usage was logged using MSI's afterburner tool which also showed instantaneous frame rate.
Test condition 1:
CPU maximum clock speed: 0.8Ghz
HD630 maximum clock speed: 500 Mhz

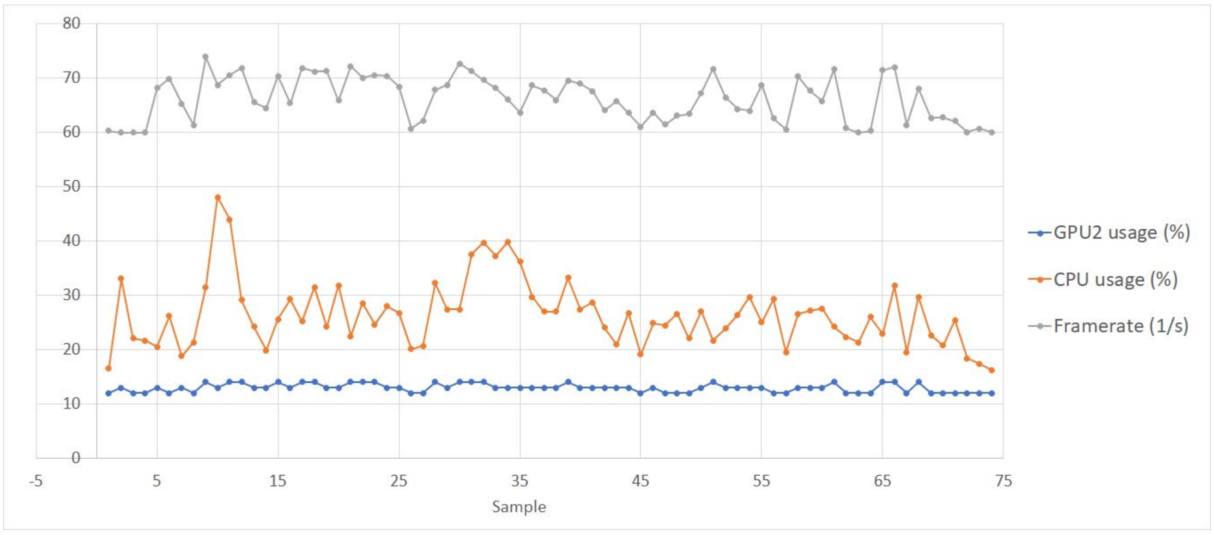


Figure 20: Graph of performance for limited system

The system even when badly handicapped can be seen to be able to achieve at least 60 frames a second as seen in figure 20. The integrated graphics card was barely used, sitting at a usage of around 10%, while the CPU had a higher load, but stayed below 50%.

Test condition 2:

CPU maximum clock speed: 2.1Ghz

GTX1060 maximum clock speed: 1670Mhz

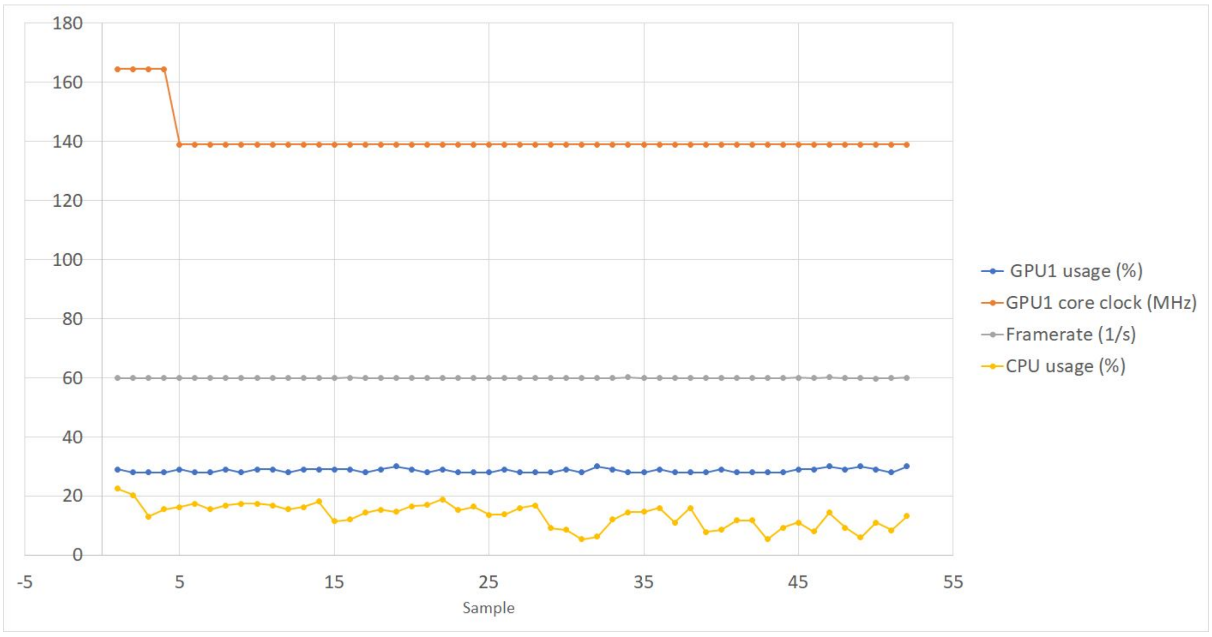


Figure 21: Graph of performance for faster system

When running the game at a higher end system, the utilisation can be seen to be lower for the CPU in figure 21. In contrast, the dedicated graphics card, GTX1060, has a higher utilisation, but at a much reduced clock speed of about 140Mhz for most of the time. This meant that while the utilisation percentage is high, it was due to the very low clock rate as the game was not demanding enough to cause it to raise its clock speed above idling speed.

Discussions

Advantages of Current Approach

The current implementation of object collision detection algorithm helps to speed up ball motion rendering by reducing the number of collision detection computation using binary search algorithm.

Limitations of Current Approach

The primary limitation for this project is the customisation and interchangeability of the maze model as it needs to update some of the configurations and recompile to use any new maze model. The user has to create the maze model and it could be inconvenient for users that do not own any 3D modelling software tools. In the future, it is possible to explore the options for auto generate random maze models for each game session thus enticing the user. Although the current collision detection algorithm helps reduce the amount of computation work but in the case where the ball is moving at high velocity, it is required to do step wise collision detection computation which increases the computation cycles.

Knowledge From Lectures

By using the model view transformation matrix between the world space and model (maze) space, it simplifies the collision detection computation as the ball coordinate is referenced to the maze plane which is a 2D plane and thus eliminates the maze rotation transformation. The collision detection approach is similar to the ray tracing where we project the ball motion ray from the ball center to check if it collides with the maze wall and update the ball position based on the collision result. The texture was mapped using planar mapping, which was also taught and implemented as part of a lesson assignment.

Conclusion

The project aims to simulate the object behaviours in the physical world, specifically the effect of gravitational force and the bouncing effect of the colliding objects. To gamify the objective, the project simulates the physical world phenomena through the maze game and allows the user to interact with it. The project improves collision detection computation performance by sorting and grouping all the maze wall planes and using the binary search algorithm to find the potential colliding planes. Although the object collision detection algorithm implemented in the project is not generalised for all the use cases, it could reduce the computation work load in the situation where the collision detection is only required in the 2D space. The project shows that it is crucial to relax the problem by eliminating the factors that are trivial for a given situation which is the Z axis movement of the ball in this project.
The project also applied unique textures to the different surface types to help the user identify the different faces. This greatly improved the ease in which the user could differentiate between the wall, floor and top. This improved the user experience and made the game easier for the user. This has also provided a chance to learn how to handle more than a single texture for a model. This is important when the model has surfaces that look different and provides more variation in the visuals of the model.

Reference

Boström, J. (n.d.). Maze Generator. Retrieved August 11, 2020, from http://www.mazegenerator.net/

Vries, J. D. (2014, June). Collision detection. Retrieved August 11, 2020, from https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection