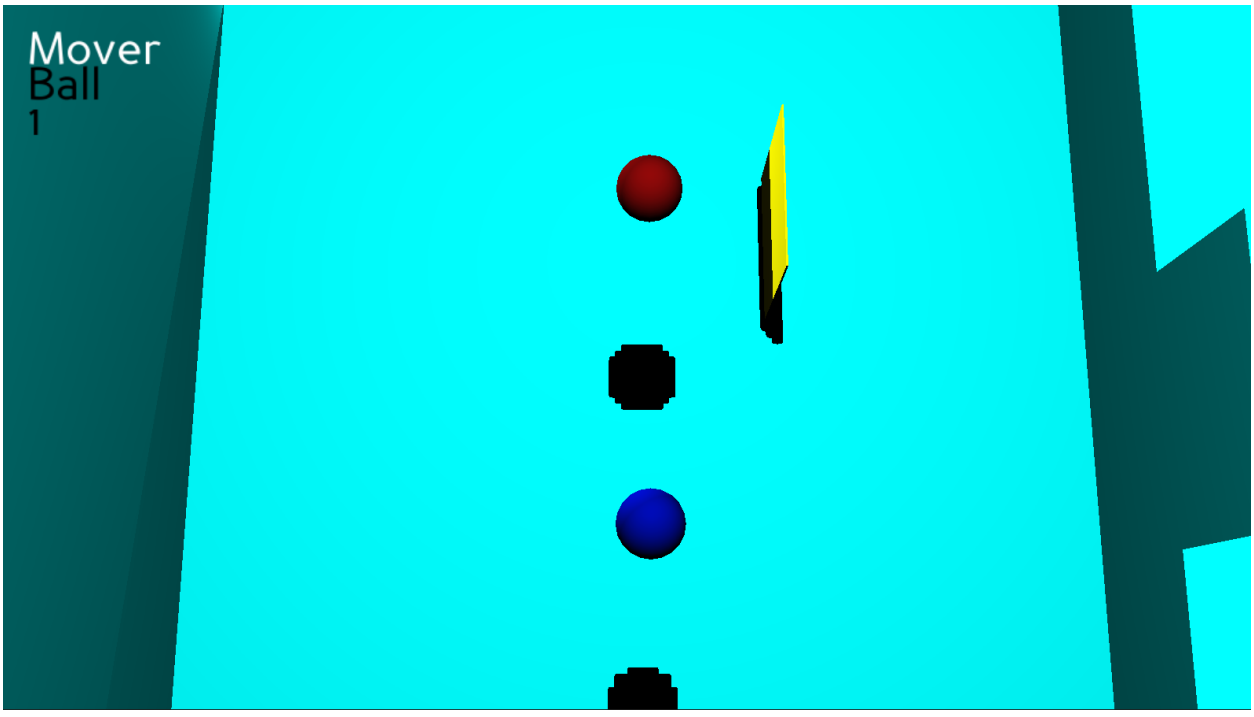
Simulating Physics and Object-to-Object Interaction with Ping Pong

Henry Cullom, Sebastian Martinez

Abstract:

In this paper we will investigate how to properly simulate rigid bodies and the interactions between them. This is a topic that we went over a little bit in class, but didn't get a chance to implement. Collision checking is a very important part of physics simulations since they allow complex object-to-object interaction, and we discussed a wide variety of methods to help achieve this goal, including Axis Aligned Bounding Boxes, KD Trees, and a few more. The way we will make this problem interesting and interactive is by simulating games, such as ping pong.



Motivation

Many modern games today utilize game engines that are written from the ground up and thus use their own physics and collision detection systems to remove any unnecessary overhead to reduce computation time or to have a tight control over every aspect of the game. Part of making a video game engine is also being responsible for the graphics rendering and what happens to all the models every frame that is rendered. Combining the models with an in-house physics system and collision detection system is a complex task that can take many months to years to refine. There are many game engines out there that are able to do all of what is needed from an engine: rendering graphics, having a physics system, and be responsible for collisions and movement; however, they can have limited flexibility to provide what a specific game needs or can be lacking in other ways to craft games, such as using an entity component system as opposed to using object oriented programming. Thus, the motivation behind this ping pong project is broken into various subsections: having practice with external engines in constructing a scene from the ground up and setting up things such as lighting and shadows, figuring out collisions and setting up a personal data structure for use, and implementing the basic kinematic equations of physics for the engine to use. The end goal is to make a simple system that can replicate balls that can bounce around a table and even collide with each other, so as to one day simulate a ping pong ball game.

Related Works

We took inspiration for our project and found help and guidance through the following works. (1) This Document discusses a way to simplify collision detection by creating oriented bounding boxes around a complex figure that completely encapsulate it with minimal empty space inside of the bounds. This allows a quick way to decide if objects did not hit each other which can really speed up computation as the number and/or complexity of objects in the scene increases. (2) Bentley’s paper discusses the

definition of the kd data structure and different uses of said structure. One advantage of the data structure is the efficiency of queries. The paper also describes the Big O notation of the kd tree operations, such as insertion and deletion. Kd trees would be a useful structure to test implementation for collision detection in objects. (3) This paper by M. Goslin and M. R. Mine describes the panda 3d engine that we're using to render our game. It helped a bit with getting accustomed to the engine and displaying the capabilities of it. (4) This paper by Cai, Panpan et al. discusses the use of axis-aligned bounding boxes and how they can be used to detect collisions. We used this paper to help with the calculations regarding the bounding boxes.

Distribution of Work

Sebastian: Working with the Panda3D engine and creating the threading system.
Henry: Working with the Euler time step and collision algorithm.
There was a combined total of around 80 hours of work put into this project evenly split between both of us.

Implementation

The Panda3D Engine

The Panda3d engine is a game engine that was developed by Disney's VR Studio for the purpose of flexibility of design; it can develop from a broad range of programs such as real time graphics applications to VR theme parks. The perks of using the Panda3d is that setting up the engine is a rather straightforward process; what makes the engine nice to use is the variety of resources that are available in setting up a scene, as there are many different ways to get started. Another advantage to using panda is its platform agnostic approach, which uses abstraction layers that allow for development on various different systems, such as linux and windows. The main reason for using the panda3d engine is for its fast development process and scriptable language; the use of Python entails simpler development time and due to its object

oriented programming paradigm, means a lot of inheritance could be used to have cleaner code.

Representing the scene:

The scene is represented by a series of classes and objects that culminate into the python file pyGameMainTest. This python file is responsible for holding all of the objects necessary for use of the scene simulation; this file holds the balls, lights, walls, and paddle. The main scene composition first sets up all the objects necessary and puts them into a list as well as a collision detection list for later reference. After all of the objects are set up and parented to the render, and all of the keyboard inputs are pushed through, the simulation can begin. One tool that we utilize that the Panda3d possesses is the Task Chain Manager System. Without this vital tool, the simulation would have a much more complicated implementation as it would be constructed with existing threading tools that Python provides. The Task Chain Manager System is a Panda3d system that runs when every frame is called of the simulation, it is responsible for executing every function that is attached to it. What makes it parallel is that every ball that is on the screen is responsible for its own physics and movement, and thus requires its own thread. One issue that arises from using chains however is the lack of critical section protection from global resources. Thus, care is taken so to not make the process repeat when it should not be.

Ball.py:

The majority of all the physics calculations and collision detection is handled by each individual ball. Each ball can have a color, a mass, and a start position and starting velocity that the physics can work off of. Once the physics simulation has begun, the balls act accordingly based on the predefined gravity of the world and whatever starting velocity it was given. Every frame each ball checks to see if it has collided with an object and to calculate the new force of the ball based on its collisions and already existing forces. The user can manually manipulate each ball to give it a new start position or even to move a ball to collide with other balls when the simulation is going under way, as to demonstrate the collisions that have been calculated.

Table.py

Static objects that do not move on their own, but can still be manually moved around by the user. However, they still interact with any balls that collide with them and thus are also added to the collision object list. They are moved with WASD and can also be rotated with the keyboard using RT, FG, and VB. One thing to note is that balls can only collide with the table and paddle when it is in either a 90 or 180 degree position as in its current state, the implementation cannot take into account the angle of the object's rotation since it utilizes axis-aligned bounding boxes.

Paddle.py

A more advanced user controlled object. This object is not affected by gravity, but is affected by force. When the user selects it by pressing 5, any attempt to move it will set the acceleration, then the rest will be handled by the calcForce method, which will accelerate the paddle around so the user can move the paddle more like a natural swing rather than just moving it slowly along small steps at a fixed speed.

Light.py, Camera.py, UI.py:

Similar to the implementation of the previous objects, the light and camera can be adjusted to the viewers liking, in which they too can also have their position and rotation changed. The UI is a static object that informs the user what object is currently being moved or when the physics simulation is currently underway.

Representing the physics

The physics system uses a method called euler timestep to calculate forces and apply movement. The Euler time step method begins by having a loop that runs iterations of the calcForce function in the ball.py and paddle.py objects. The time between the iterations is called dt, and will be kept track of to be used later. The calcForce function begins by taking in some accelerations such as gravity for the ball or the force the player applies to the paddle to move it. Next, the acceleration is multiplied

by the dt to acquire the increase in velocity, which is added to the old velocity to get the current velocity. Then, that resulting velocity is multiplied by dt and added with the previous position to get the resulting position caused by all of the forces and accelerations. This method is simple and works great for a physics simulation where there is not a lot of circular motion, which the Euler time step method does not do very accurately.

Collision detection

Each object in the scene has a bounding shape, with balls using a bounding sphere and boxes using axis aligned bounding boxes. They need to be able to interact with each other to properly detect collisions. The sphere shape is stored as a center point and a radius, and the axis-aligned boxes are stored as two points, one the minimum x, y, and z point, and one the maximum x, y, and z point.

Between spheres, collision detection is as easy as comparing the distance between their centers with the sum of their radii. The checkCollision function also returns the point at which the two spheres meet which can be used to calculate the normal to be used for collision handling.

For the axis-aligned boxes, one can compare the max point of the one closer to the origin with the minimum point of the box further from the origin to see if they collide. This works by comparing the x, y, and z of each point and if the min of the further box is less in all three x, y, and z of the max of the loser box then there is a collision. This type of collision is not used by our program, but can still be calculated.

For the Sphere to Box collision, a different technique must be used. The following algorithm can calculate the closest point to the center of the sphere that is on the box.

```
For i in x, y, z:
  if (c[i] < b_min[i]):
    p[i] = b_min[i]
  if (b_min[i] < c[i] < b_max[i]):
    p[i] = c[i]
  if (b_max[i] < c[i]):
    p[i] = b_max[i]
```

This produces p, the closest point on the box to the sphere. Once p is acquired, then we can check the distance from the center of the sphere to the closest point in the box. If this distance is less than the radius of the sphere, then there is a collision.

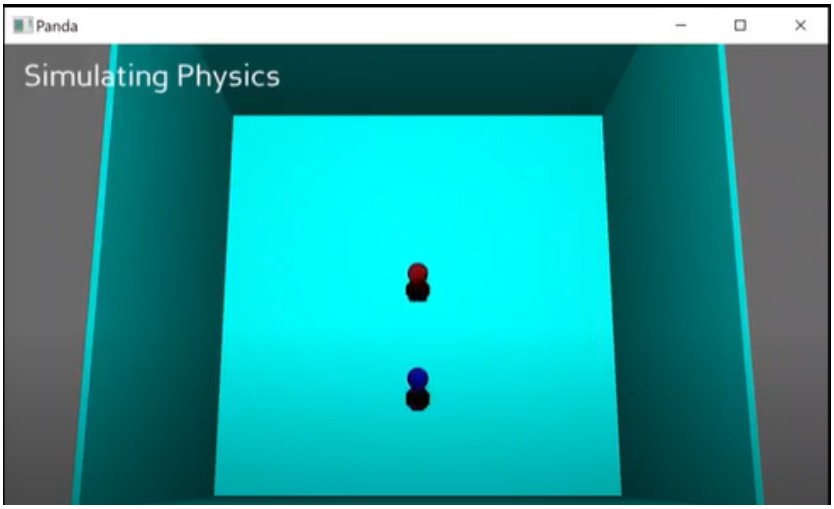
Collision Handling

Once we determine if there is a collision, we move on to collision handling. To handle collisions, we first move the object outside of the thing it collided with. Then, for a simple reflection, such as a ball against the static table or against another ball of the same mass, the velocity is reflected along the normal of the collision point. The velocity is also reduced slightly here as a simple hack to make sure the balls don't bounce forever.

For the complex collision of the user controlled paddle, we need to account for conservation of momentum for the ball. The effect of the ball on the paddle is negligible, so it is ignored, but the paddle is much more massive than the ball, and therefore imparts its momentum onto the ball when it moves.

Testing

Testing our implementation required iteration over each subsequent goal: first we needed to ensure the scene can be loaded with the desired EGG files that were converted from Maya Binary, then we needed to manipulate and position all the models in the desired location to begin implementation of our physics system. However, before the physics system could be made, we wanted to ensure that every object could be manipulated by user control as to ensure that they can be moved to



demonstrate certain test cases, such as pushing the limit of the engine to manipulate many balls or to move a table to affect the trajectory of a ball's path. Once the scene was set, the lights were added, and the shadows rendered, we then worked on implementing the physics one iteration at a time. First, by adding in the force of gravity and adding in the euler time step method we were able to implement the simulation of a ball falling straight into the ground. After that was working and we could ensure that multiple balls could fall with the help of the Task Chain System, the next step was to implement the collision detection system to then ensure that the balls would not just lay static once colliding with an object. Finally once a simple collision detection system was in place, the next step was to test out and implement the collision of the balls with other balls.

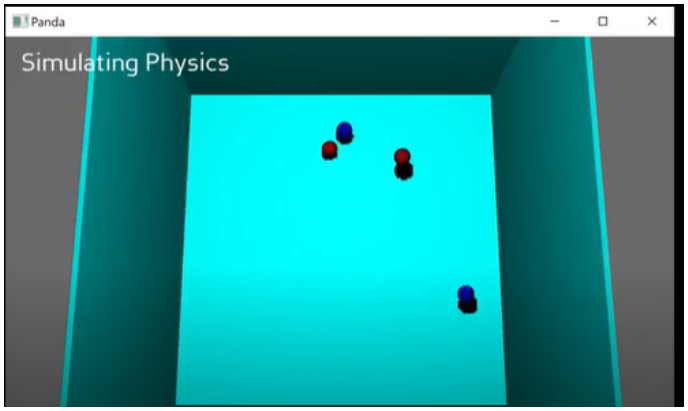
Technical Challenges

We experienced a few technical challenges throughout the process. First, and one that we couldn't figure out how to solve was an issue with Panda3D's threading. We constantly got a nonsense error from a random part of the code that was somewhere in the Panda3D library. This error had no apparent cause, but caused one of us to randomly crash with no repeatable steps to cause the crash, and caused no issue for the other.

Another challenge was deciding which objects should handle collision. With multiple balls in the scene all running on different threads, it would be hard to decide if a collision between ball a and ball b detected on ball a should move both a and b or just a. After some testing of each strategy, we decided that a collision should only be handled by one ball and not both for simplicity. This means that ball a will move itself and reflect the velocities of itself and ball b. This means that balls further up in the collision detection order will handle the brunt of the collisions while the ones lower down don't, but this didn't seem to cause any issues.

Examples

To start testing our implementation of collision detection and making a physics system, the progression of test examples were based on successful iterations of the previous examples. In the first test example we tested our forces like gravity and drag by having a ball start with some speed and let it fall through the scene. This would show the parabolic arc that the ball takes and helped us make sure that the physics and Euler time step method were working properly. The next example is used to test basic collisions. We placed a static plane into the scene and put the ball above it, allowing it to fall and test the collision and resulting bounce. Next we added walls around the table to keep the objects contained and added many balls to the scene. This is the test to see how the balls interact with each other. The final scene is interactive, where the user controls the position of the paddle and hits the ball with it.



Results

The project went more smoothly than was anticipated. Aside from the small technical challenges discussed earlier, after implementation and bug fixes, the physics look great and realistic. The performance was almost too good at points and the really small timestep that resulted made debugging at the beginning tough since the balls would just disappear if the parameters weren't set correctly. Overall, the project was a success.

References:

1. Gottschalk, S., Lin, M. C., & Manocha, D. (1996). OBBTree: A hierarchical structure for rapid interference detection. In Proceedings of the 1996 ACM SIGGRAPH Conference (pp. 171-180). ACM Press. https://doi.org/10.1145/237170.237244
2. Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9), 509-517. https://doi.org/10.1145/361002.361007
3. M. Goslin and M. R. Mine, "The Panda3D graphics engine," in Computer, vol. 37, no. 10, pp. 112-114, Oct. 2004, doi: 10.1109/MC.2004.180. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1350741
4. Cai, Panpan & Indhumathi, Chandrasekaran & Cai, Yiyu & Zheng, Jianmin & Gong, Yi & Lim, Teng & Wong, Peng. (2014). Collision Detection Using Axis Aligned Bounding Boxes. 10.1007/978-981-4560-32-0_1.