COS 426 Final Project Report: Hamster Battle

Katie Baldwin, Jeremy Kiil, Ryan McDowell

Abstract

Our final project is the creation of a survival game where the player is a hamster in a ball, and faces increasingly difficult waves of enemy hamsters trying to push them off a table. The creation of this game entailed designing a hamster with corresponding animations, simulating the physics of the environment, developing a strategy for non-player-controlled (NPC) hamsters, and creating an appropriate level progression. Though the game has an outstanding performance issue where the frame rate capabilities of the local computer determines the speed at which the game is played, the game runs smoothly and progresses as intended. Potential future work includes adding power-ups and an ability purchasing system, which would improve player retention and increase the probability that the player could last beyond the first few rounds.

Introduction

When considering our project, we took inspiration from a few notable sources. The first was the traditional children's game "King of the Hill," in which players attempt to stay atop a hill or other elevated area while pushing others off. The second was high school physics, specifically homework problems involving elastic collisions between billiard balls and other such objects. The final was the movie *Bolt*, specifically the character Rhino, who is a fearless anthropomorphic hamster whose primary means of locomotion is a hamster ball.

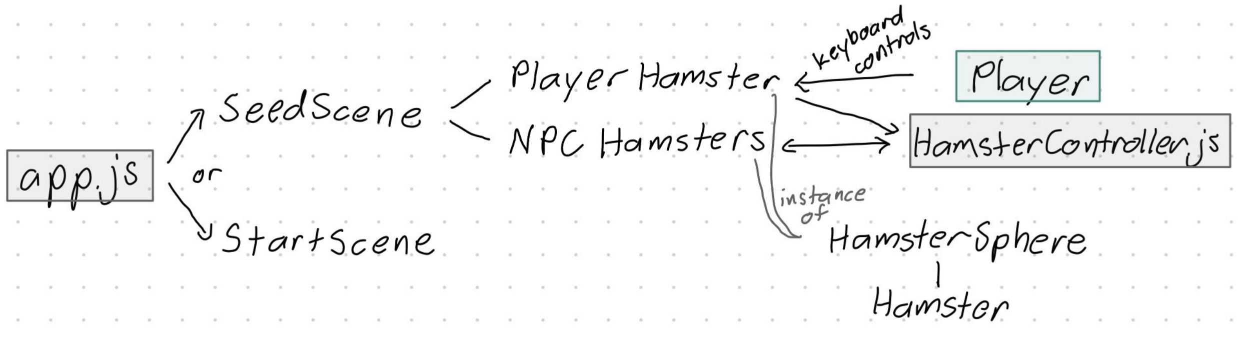
Putting these ideas together, our goal was to create a fun game involving animated hamsters and object collisions. The player operates a hamster ball and is in competition with other hamsters (whose behavior is hard-coded) to try to push them off the edge of a table before they do the same to the player. It is a survival-based game where, as the player pushes the whole field of enemy hamsters off the table, more rival hamsters of increasing intelligence and strength appear. Eventually, the player will be overwhelmed by the caliber of hamsters who are trying to push them off and be forced off the table.

We implemented this by starting with the course-provided template code. We used Three.JS libraries for our implementation and Blender for the hamster design and animation.

Methodology

<u>Overview</u>

We had the following object hierarchy for our project, which integrated all upcoming methodologies in a logical way.



Hamster Design and Animation

Although various elaborate Blender hamster models are available for purchase on the web, we went with a simple hamster design which we created ourselves, made mostly of cubes, with spheres for the head and nose and toruses for the ears. We attempted to create an armature and bones and create the animation using pose mode, but this proved too difficult to figure out, so we went with a simple repositioning of the legs to simulate the hamster's running.

Physics Simulations

The primary moving components in our game were the hamster balls, who experience the following phenomena:

1) Gravity
2) Collisions with the table
3) Collisions with each other
4) Internal propulsion (i.e. the hamster "running")
5) Friction with the table

Our philosophy for implementing these was that we wanted the game to behave with a semblance of realism, as opposed to being as realistic as possible. Accordingly, some of these forces were modeled in ways that are not entirely true to life.

Our physics simulation is based upon the equations from which Verlet integration is derived, as presented in the Assignment 5 specifications. Thus, the position of each hamster ball at time *t* + *dt* is given by

$$\vec{x}_{t+dt} = \vec{x}_t + (1 - D)\vec{v}_t dt + \vec{a}_t dt^2$$

Like in Verlet Integration, we calculate the acceleration vector for each hamster ball at each timestep based on the forces it is experiencing. Unlike in Verlet integration, we store the velocity vector of each object explicitly, and update the velocity using the formula

$$\vec{v}_{t+dt} = \vec{v}_t + \vec{a}_t dt$$

We chose to calculate and store velocity explicitly because this makes it easier to implement elastic collisions, during which there are sudden changes in velocity that are not easily represented as forces applied only in a given timestep.

Gravity is important for the game so that we can spawn new hamster balls in the air and so that they can properly knock each other off of the table. Based on the above, it is straightforward to implement gravity by applying a force in the negative y direction on each sphere at every timestep that is proportional to its mass.

The table in our game is a rectangular prism, and we implement collisions with the table by implementing collisions with an axis-aligned rectangular prism. To do this, we check whether the sphere overlaps with the rectangular prism. If it does, we move it outside the prism and set its velocity in that direction to 0. For the most part, the hamster balls only intersect with the top half of the table, but this implementation allows for the possibility of expansion. We tried modeling the normal force being applied to balls that touch a box, or treating it as an elastic collision, but this led to undesirable bouncing and other effects (see implementation hurdles for more details about this decision).

Collisions with other hamster balls are implemented as elastic collisions between spheres, meaning that the total kinetic energy is conserved, meaning that the balls bounce nicely off each other. To do this, we find the velocity of each of the spheres in the direction toward the center of the other sphere. Then, we use the equations for one-dimensional elastic collisions:

$$v_1 = \frac{m_1 - m_2}{m_1 + m_2} u_1 + \frac{2m_2}{m_1 + m_2} u_2$$

$$v_2 = \frac{2m_1}{m_1 + m_2} + \frac{m_2 - m_1}{m_1 + m_2} u_2$$

where u_1, u_2 are the initial velocities of the spheres and v_1, v_2 are the final velocities of each of the spheres.[1] We subtract the 1D initial velocities from and add the 1D final velocities to the current velocity to get the final result.

When the hamsters are instructed to move forward, either by the player pressing the ‘w’ key or by the HamsterController that governs the NPCs, a force is applied to the sphere in the direction that the hamster is facing, accelerating the sphere forward. This magnitude of this force differs between hamsters, which each have a certain “power” level that determines their strength. As the game progresses, the new NPCs that spawn are increasingly strong.

The friction with the table allows hamsters to slow down by not applying a forward force. We implemented this as a force that is proportional to the hamster’s mass that acts in the opposite direction of the sphere’s velocity. In real life, friction (at least with a solid surface) only applies when the object is touching the surface. For simplicity, and since the hamster balls are almost always touching the table, we made friction always active. We tuned the strength of friction so that the balls still slide around a little bit, but still slow down quickly enough to allow for a large degree of control.

Hamster Control

At each time step, a hamster ball can do one of four things: move forward, turn left, turn right, or do nothing. As described above, moving forward applies a force in the direction that the hamster is facing. Turning left or right rotates that hamster as would be expected. Two notes are in order: first, the hamster can take at most one of these actions, it cannot go forward and rotate simultaneously. This makes sense in terms of how hamsters inside balls run in real life - to our knowledge, they can’t really rotate and run forward at the same time. Further, as we understand it, hamsters cannot run backwards inside a ball, so we have no method for any hamster to move backwards. This means that all hamsters must rely on friction to slow themselves down, and, for the player, this means that it becomes an art to control the hamster effectively.

NPC Strategy

The programming for the “enemy” hamsters could have been approached many different ways. The strategy that we chose was simple, but suboptimal. There are two levels of random choice to the strategy. First, the hamster chooses whether to pursue the player or pursue a random point on the map. This choice is made after 200 frames of pursuing the player or after reaching the most recent randomly chosen point on the table. Each hamster has a “randomness” level that corresponds to how likely they are to choose to pursue a random point. The NPCs at the

[1] “Elastic Collision,” Wikipedia, accessed 12/15/23, https://en.wikipedia.org/wiki/Elastic_collision.

beginning of the game have randomness 0.9. Having chosen a strategy (either pursuing a random point or the player), the NPC then decides whether to use this strategy or do nothing. All NPCs choose to use their strategy in each frame with 80% probability. If they choose to use their strategy, they then either rotate towards or move towards their target, depending on whether or not they are facing (approximately) towards their target.

If the NPC is pursuing the player, it wishes to collide with the player such that the player will be propelled towards the nearest edge. Therefore, the point of reference p on the player that the NPC wishes to collide with is the point on the sphere opposite the point closest to the edge. The angle between the direction d_i of NPC i (which extends from the center c_i) and the vector v_i formed between c_i and p is defined as θ_{v_i, d_i}, where $\theta_{v_i, d_i} \in [0, \pi]$. If $\theta_{v_i, d_i} < \pi/6$, then NPC i moves forward to potentially collide with the player. Otherwise, NPC i must turn. To determine the direction to turn, we compute the cross product $x_i = d_i \times v_i$. If $x_i > 0$, the instruction for this time step is turn left; otherwise, turn right. When an NPC is near the edge, it is especially resourceful, and chooses the best strategy with 100% certainty.

There is a key oversimplification in this strategy, in that it disregards the position of the other NPCs. This selfish pursuit of the player leads to hapless collisions from time to time.

Given far more time, reinforcement learning would have been a more ideal solution to allow for intelligent NPCs with optimal strategies. However, given the complexity of such an implementation, it was infeasible for this project. It may be for the best that it was infeasible, too, because having four completely optimal hamsters trying to push you off the table would be far too difficult. As is, even with our suboptimal NPC strategy, the game ends up being quite hard as the levels progress, though it is possible for a crafty player to lure later-level NPCs into chasing them at high speed near the edges of the table, tricking them into falling off (this can also lead to the player's own demise!). If we did use reinforcement learning, to balance NPCs who pursue a near-optimal strategy, more randomness in which choice they pursue would likely need to be inserted.

Level Design

The player can begin the game by pressing the spacebar. The game starts at Level 1, Round 1. Each round, new hamsters spawn from the sky. There are three rounds per level, containing one, then two, then three hamsters that the player must knock off. Every 3 rounds, the level increases, and the following changes are applied to the NPC hamsters: the color changes, the mass of each hamster is multiplied by 1.1, the power of each hamster is multiplied by 1.5, and they become "smarter" in that the randomness level is multiplied by 0.8 (so they pursue the player more often). This means that the difficulty of the game increases rapidly! Once the player dies, they can restart the game by pressing the spacebar.

Miscellaneous Implementations

We attempted to use Three.JS text display interfaces, but as we were unable to get those to work, we resorted to plain HTML inserted into the div created by dat.GUI. The start and restart scenes pages are created by the same Scene object, and the text corresponding to the state is hidden or shown accordingly. App.js keeps track of the current state using a global variable.

So that the player could see more easily which direction their hamster is facing, we rotate the camera with the hamster. Specifically, in every frame, we move the camera to be 8 units behind and 5 units above their hamster, looking in the direction of the center of their sphere. This position was chosen such that you were fairly close to the hamster but still able to see some of the surroundings.

Results

Overview

The resulting game runs as intended using the methodologies listed above. Collisions and friction feel realistic, the hamster design and animations are superb, the NPC hamsters effectively push the player off of the table if given the chance, and the level progression and the corresponding difficulty leads to a challenging but fun game.

Known Issues

There are a few known issues which could not be worked out, though. The game performance is unfortunately dependent on the graphics processing capabilities of the machine it is running on. Though our respective machines all allow for a playable game, we found that we could not tune it to perform optimally on all of them, and it was thus tuned to the following laptop specifications: a Dell Inspiron 5593 with an Intel i7 processor, 8 GB of RAM, and an Intel Iris Plus graphics card. Notably, our game seems to run much slower on some of the Apple laptops we tested our finished product on. This is unfortunate because this can make some of the aspects of the game less intuitive - gravity, friction, and rolling forward apply at slower rates than would be expected which can make the game hard to play.

One potential fix to this issue would be to cap the frame rate of the game and tune it to the lowest-performing laptop we have access to, but such a solution is not ideal because there are no guarantees that any given laptop would meet that minimum specification. The best solution would be to base our updates on the wall time, not on frame generation. Unfortunately, that would require a full redesign of our update loop, and was not feasible for this project.

An additional known issue is that, because of how we handled key presses, if you are holding down a key to rotate or move the hamster and press another key slightly later, the hamster stops rotating or moving. This is unfortunate behavior and could be solved with a bit of a redesign with how we handle key press events (specifically, we would have to check for key-up as well as key-down events). Since we realized this issue fairly late in the process, we unfortunately did not have the time to address it.

Discussion

Implementation Hurdles

One implementation hurdle was getting the hamster to stop moving at the proper times. We first tried to stop the animation if the velocity was 0, but this proved more difficult than anticipated. The work-around we implemented was to have a counter in each hamster object that restarts each time the “w” key is pressed and the goForward() method is called, and is incremented each time the update function is called. Once the counter reaches 50 (arbitrarily picked, because it “works well in practice”), the hamster stops moving. This also works for non-player-controlled hamsters because they also use the goForward() method

Animation of the hamster was also difficult because none of us had experience with Blender. We tried to implement the animation using bones and pose mode, but our rudimentary understanding of Blender was not up to this task.

Another implementation hurdle was tweaking Verlet integration to get our method of modeling physics. As was mentioned above, we wanted to explicitly store the velocity of each sphere to make elastic collisions easier. The major challenge we ran into when doing this was correctly implementing the collisions between the spheres and the floor. Initially, the spheres would sit on the floor and then get pulled through the floor by gravity - this was because we were still increasing their downward velocity vector even though we were holding their position constant. Eventually, this downward velocity would grow so great that they would travel below the table in a single timestep. This was undesirable behavior, so we then tried to model the normal force of the table against the balls, but this was hard to get right because of floating point number precision issues that caused balls with certain radii to bounce up and down. In the end, we decided to simply zero out the velocity of spheres in the direction of a box they were touching, which accomplished our purposes well but could be changed if we wanted to add more features (such as other boxes acting as walls) to the game.

A final implementation hurdles was fine-tuning the logic behind the movement of NPCs. Specifically, when we started making the NPCs move towards random points on the table, they would sometimes end up orbiting around those points because of the angle they were accelerating in, never quite reaching the point itself. Further, sometimes the point they were targeting was near the edge of the table and they would fall off since they wouldn’t slow down near the edge. To address these issues, we instituted the following policies in the NPC logic:

1) If an NPC is pursuing a random point and it is within 1 unit of the edge, it turns and moves towards the center (this almost always prevents NPCs from drifting off the edge of their own accord).
2) If an NPC is pursuing a random point and it is within 2 radius-lengths of the point, it slows down by not always accelerating (this prevents NPCs from getting stuck in an orbit around the target point).

These policies largely addressed these two problems.

Future Work

There are many features which could be added to improve and expand the game. A fun addition would be power-ups that randomly spawn on the table that increase the mass, power, or maximum speed of the player as the game progresses. Once a player picks up a power-up, their appearance would change in some way (e.g. radius increase, color change), and their attributes would temporarily change to align with the effect of the power-up. One consideration to make is that, in order to make the power-ups a bit more game-changing, the NPCs should be able to pick them up as well. A random subset of the NPCs should temporarily direct their attention to a power-up once it spawns, and then focus back on the player once it is picked up. To facilitate this change, the area of the table would likely need to be increased to avoid incidental intersections with the power-up, and perhaps some barriers on one or two sides of the power-up would introduce an interesting dynamic.

We also considered expanding the table into a larger obstacle course, where the player would try to get as far as possible while being attacked by rival hamsters and possibly other hazards. Our current implementation of sphere-box intersections would likely have made this possible if we had more time to design sensible obstacles and a way to make them randomly spawn as the player progressed.

Also, we had dreams of instituting a deeper progression system that grants you improvements as you reach certain levels, or better yet having a purchase system where you can buy certain abilities or improvements once you acquire enough points (1 point = 1 hamster forced off the table). Finally, the following small quality-of-life improvements would be welcome: tracking high scores, displaying the round and level in the UI, and displaying how many hamsters you "killed" after you die.

Reflections

We are very pleased with the outcome of Hamster Battle, and learned a great deal in the process of making it. In addition to an improvement to our software engineering skills and our intuition for creating a sensible object hierarchy, we learned a lot about physics simulation and even Blender!

Conclusion

Overall, this game represents a great success in our eyes. Not only does it play smoothly with no visual glitches or unexpected behavior, but it is precisely the game we imagined implementing at the start of this project, with very few compromises made. We are especially pleased with the physics simulations that result in a realistic and satisfying feeling when collisions occur. Though there is a known performance issue (mentioned in the Results section), we are very pleased with the game we have created and we have had a great time playing it!

Contributions

Ryan designed the physics simulations involved in the project and helped implement some of the random NPC choices. Jeremy worked on the strategy of the non-player-controlled hamsters and the NPC integration into the scene at large. Katie designed, animated, and integrated the hamster animations. She also implemented the "start" and "restart" game screens and game state transitions.

Works Referenced

Williams, Chris, and Byron Howard, directors. *Bolt*. Walt Disney Animation Studios, 2008. 96 minutes.

https://tympanus.net/codrops/2019/10/14/how-to-create-an-interactive-3d-character-with-three-js/ - character animation tutorial, linked from project page https://www.youtube.com/watch?v=cHI2hzL-pcQ - A video of a hamster, referenced for animation purposes.

https://www.youtube.com/watch?v=Rqhtw7dg6Wk - A blender tutorial

https://www.makeuseof.com/how-to-animate-in-blender/ - Another blender tutorial

http://blog.nuclex-games.com/tutorials/collision-detection/static-sphere-vs-aabb/. Referenced when formulating a method of checking whether a sphere and axis-aligned box are overlapping.

The THREE.Js documentation