Collision Mania

An Interactive Game to Catch Falling Objects

Ramla Ijaz, Nguyen Phong Le, Brett Phelan, Longfei Qiu

Abstract

We present in this report the game *Collision Mania*. The goal of this game is to control the player character to catch and break randomly falling cubes to get as high a score as possible before exhausting all lives.

The game's aesthetics were inspired by the music video for the song *Hyperballad* and crafted using Three.js's existing geometry classes and functions. The gameplay and the various visual effects present were implemented using Three.js (rendering) and Cannon.js (physics). We detail in the following sections of this report our game concept and the making of the game's background aesthetics, as well as the gameplay and our implementation of the physical effects like falling and shattering cubes.

Gameplay

In our proposal presentation, we pitched a game based around *avoiding* falling objects. Our gameplay has since evolved: the goal of the game is now to *catch* the falling cubes.

The player controls a stick figure in a square area using the keyboard arrow keys and starts each game with 0 points and 5 lives. Each time a cube falls and hits the ground, the player loses one life. Each time a cube falls on the player character, the score is increased by. The cubes are generated randomly, and the game will continue until the player exhausts all of their lives.

The game will also get progressively harder the longer the player survives. This is done by gradually increasing the random cube spawn rate, up to a maximum of one cube per second. The player is eased into the game in the beginning with a spawn rate of one cube every 5 seconds.

Related Work

We were inspired by multiple three.js games both in aesthetics and gameplay. For aesthetics, many games use blocky, polygonal shapes for their world building such as "The Frantic Run of the Valorous Rabbit" [2] and "The Endless Runner" game [3]. Many of these games also have an element of collision detection, which is a core part of our game as well. Our inspiration for

exploring the shattering effect in our game was the “Smash Hit” game where a ball hits shards of glass and they break with realistic effects [5].

Game Concept and Aesthetics

Our game is inspired by the music video for Icelandic singer Björk’s song Hyperballad. The video contrasts computer generated cityscapes, pylons, and screens with the dirt and paper landscapes over which they are projected. The lyrics describe the singer’s morning ritual of tossing objects off the side of a cliff and, as she watches them crash on the rocks below, imagining herself in their place to make herself feel happier with her lover. In our game, we adopt the aesthetics of the video, and the perspective of an inhabitant of the valley below who must catch Björk’s seemingly endless ammunition of “whatever she finds lying around” - this morning, cubes. In Figure 1, we show two screenshots taken from the Hyperballad music video [1] which we use as inspiration for creating our game scenery.

Figure 1: Screenshots from Bjork’s Hyperballad music video that we took as inspiration for our game aesthetics.

Our first task was to create the background cliffs. We chose to create a low-poly plane. We generated a rectangular backdrop and displaced each vertex of the mesh by a random amount in the z-direction to give it a crumpled appearance. This is a common method to create low planar scenery [2, 3, 4]. The other option we considered was to use a crumpled texture on a plane to obtain a realistic look. Another option was to use Blender to create a low-poly planar 3D environment and import it into our code. We decided using Blender was not as useful in our game since the stick figure was going to be moved in a small area in front of the cliffs. So, we didn’t need to put effort into creating a perfect environment that could be viewed from all sides, which blender would help with.

The soil look on the ground was added by creating a plane with a texture taken from an image of pebbly soil. In addition we added gray fog to give the dusky look as seen in the music video. We added an additional spotlight focused on the areas where the actual gameplay occurs, to mimic the look of moonlight in the music video and make the player boundaries clearer. Lastly,

we added outlines of buildings with fluorescent blue windows. Again, we went for a polygonal rather than a very realistic look. For the buildings we created a group of planes of size 1x2. Each plane represents a “block” of the building and was shifted to its correct position. Each plane was colored either dark blue or a fluorescent blue. The opacity for each color was set randomly to get a non-uniform slightly more realistic look.

Finally, we added clouds from the music video as a repeated texture on a plane in front of the scene. The transparent background of the texture made only the clouds visible. We repeatedly offset the UV map of the texture to make the clouds move.

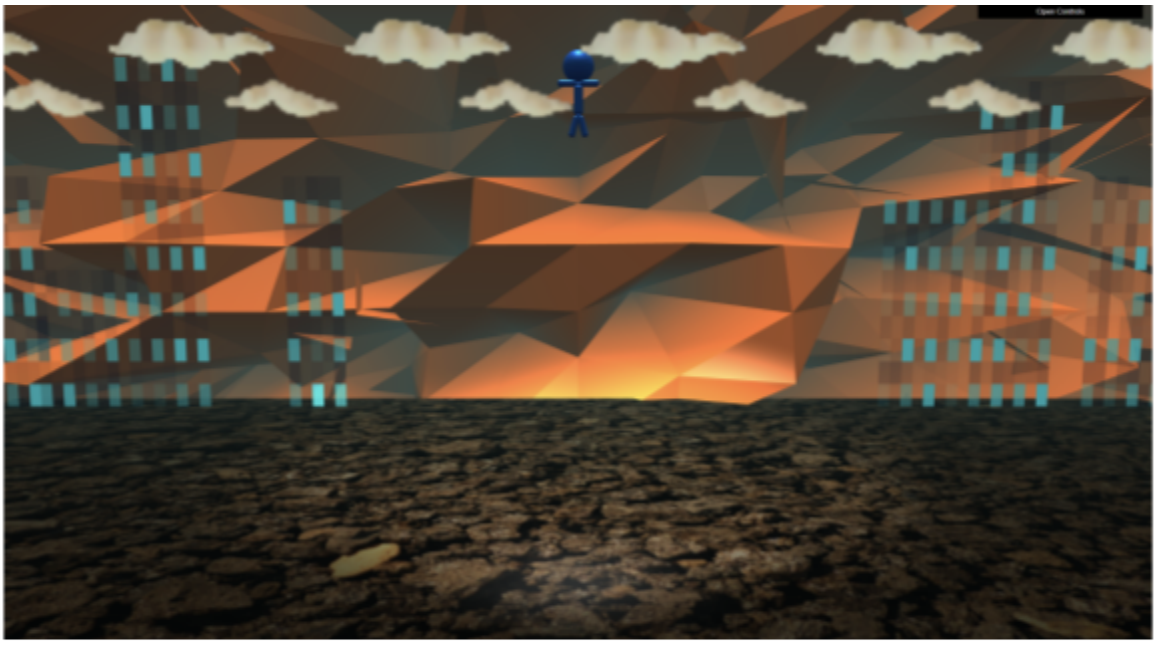

Figure 2: Scenery of the Collision Mania game. Almost all aspects are inspired by the Hyperballad music video

Game Physics

Our game required three physics simulation functionalities: realistic falling animations, collision detection, and interaction between rigid bodies. Each of these is easily implemented using the Cannon.js physics engine (the original Cannon.js package is no longer maintained - we used `cannon-es` instead), which animates each body by considering the forces acting on it at discrete time-intervals. Additionally, it allows us to set the initial conditions of each body as they enter the scene. In our project, this allowed us to easily tune the difficulty and gameplay nuances of the game by, for instance, increasing the speed at which spawned objects fall, or adjusting the acceleration of the player character.

A realistic falling animation was obtained by setting gravitational acceleration to 9.81 m/s, the actual Earth gravitational acceleration. Collisions are simple to detect between Cannon.js

bodies, but are clearly dependent on the object's defined hitbox. For simple shapes, such as cubes or spheres, there is a clear identity between the rendered (visual) and simulated (physical) object. More complex objects, such as the player figure, however, are approximated by one or multiple bounding volumes. For the purposes of our game, this approximation has virtually no artifacts on the player side; objects are large and fall from above, so if an obstacle enters the bounding volume, it likely intersects the true player figure as well. Interactions which result from collisions are used to simulate a realistic shattering effect.

To make generating physical objects easier and more streamlined, we created custom classes for our objects extending from the existing Three.js Mesh class. For these classes, we added extra properties to store the Cannon.js physical body, as well as some helper functions to update the mesh position - as we have stated earlier, the rendered and the physical objects are separate and the rendered objects' positions and rotations must be updated after each simulation step. These custom classes are especially useful when we are generating random infinite cubes.

The choice to use a physics engine for our game is ultimately a stylistic one. Simulation of basic falling objects and detection of collisions between cubes hardly requires an entire library - all this can be done by simple quadratic equations and pairwise distances. The simulation becomes more demanding only once we consider how shattered pieces of an object interact with the environment and one another. From a gameplay perspective, the simulation of these interactions is inconsequential; the player must only catch objects as they fall, not once they have hit the ground. A physics engine simply enables a realistic simulation of these "shatterings", which we will detail in the next section.

Collision Detection

The Cannon.js physics engine allows us to easily detect collisions between the player character and the falling cubes. We handle collisions by attaching an event listener to the cube that waits for a "collide" event to call a collision handling function. This function checks that the cube collides with either the player or the ground and shatters it when appropriate.

Shattering Effect

Within the context of a realistic simulation, deviations from expected behavior can be glaring. This can of course draw attention to undesired errors in the simulation, but can also be exploited to emphasize certain events and make them appear and feel more impactful. This principle is used ubiquitously in film and video games, and we apply it in our game to emphasize cubes colliding. We coded 2 different "shattering" effects to be invoked when a cube hits the player character (the user can toggle between the two different effects with the "ShatterAnimation" checkbox in the menu).

The first, more dramatic-looking effect is to spawn a number of smaller cubes traveling in random directions. Implementing this was quite straightforward, as we were able to leverage our existing code infrastructure to create the extra cubes.

For the realistic shattering effect, we attempted several iterations. We first attempted an effect which partitioned the cube into 6 square pyramids by face and created 4 randomized shards from each. The randomized component and irregularity of the shapes lended this effect a sense of realism, but we encountered difficulties with the physics simulation. In the end, we adopted a simpler method which split cubes up along the faces of the mesh (2 triangles on each face), connecting each face to the center to create 12 triangular pyramid shards. To obtain their bounding boxes for the physics engine, we used Cannon.js's Trimesh body, as our previous attempts at using ConvexPolyhedron were unsuccessful. This allowed us to obtain shards that bounce realistically against the ground. Nonetheless, the shards still did not interact realistically with each other and would sometimes accelerate in unexpected directions. This is something we can strive to fix if we were able to work on this project again in the future.

For both effects, we also implemented a delayed destruction system where each object is assigned a time where it needs to be destroyed. Each cube/shard contains a destruction time, and with each frame update the app checks in the list of rendered objects whether there are any that need to be removed.

Interactive Component / Controls

We went through a few iterations of the player character movement before settling on the current one. Initially, we simply displaced the player character by a set number of pixels every time the game detected a keypress. However, as we incorporated the physics engine into our game, we strove to make moving the player character more realistic and challenging.

What we ended up with is a system where every time an arrow key is pressed, we accelerate in one direction by adding incrementally to the player character's velocity. When the key press is released, the player character doesn't stop immediately, but slows down gradually. This yields a "slippery" feeling when controlling the player character and adds an extra layer of challenge for players (for example, they might rush to catch a cube but must also consider that they take a few moments to slow down). To make sure that the player character can move unimpeded, we also had to play around with the friction between the player's hitbox and the ground / stage boundaries.

Graphical User Interface

We implemented a basic Graphical User Interface (GUI) that allows players to control the game, including buttons to start, pause, and reset the game. We also have 2 graphical toggles that the user can click on to enable or disable the shatter effect for the falling cubes, and to enable or disable the texture mapping on the ground. We also used the GUI to display variables such as

the Score and Lives of the player. The GUI library does not update these values automatically, so we have to manually update the labels each time our scene is updated.

We implemented this GUI using the dat.GUI library that was included with Three.js. Three.js does not seem to support more advanced interfaces out of the box, so we went with the option that would make the implementation as painless as possible.

Future Directions

Due to a lack of time, we were not able to go further than the content described before the deadline. If we were to continue working on this project in the future, we would look into incorporating the following:

- An external backend/database to store players' scores; implement a leaderboard system.
- More realistic shattering effects.
- Greater variety in falling objects. This will require working on separate shattering/exploding animations for each.
- Animation for the player character.
- A better menu system. For instance, a 3D menu instead of the current basic one made with dat.GUI.

Conclusion

In our game Collision Mania, we were able to complete our two major goals for the minimum viable product: implement a basic collision detection system between a moving object and falling cubes, and introduce the basic cliff aesthetics from the Hyperballad music video. We were also able to work on additional goals, namely shattering effects and improved aesthetics including clouds and buildings. We have become familiar with many components of three.js that are useful for game building as well as the cannon.js physics engine.

Contributions

All members of this group actively participated in testing and checking all code. Our development process was to work on a separate branch, and then submit pull requests that could be reviewed by other members. We kept an active discord server where we all asked for help, and received feedback and ideas from our group members. Specifically, the areas of focus for each member were:

Ramla Ijaz: Game aesthetics including cliffs, ground, buildings, and lights; camera perspective and placement of objects in the global space; experimenting with mass, velocity and frequency of falling objects to increase difficulty of game play.

Nguyen Phong Le: Implemented physics for the game using Cannon.js. Written code for the GUI, the randomly generating cubes and the template for custom class for objects incorporating both a Three.js mesh and a Cannon.js physical body. Contributed code to the cube shattering effect.

Brett Phelan: Game aesthetics, especially clouds, lighting, and shadows. Experimenting with cube shatter effect, attempting to tune physical parameters to make effect realistic. Created shard class for shatter effect.

Longfei Qiu: Making stick human model in Blender, keyboard movement control, small cube version of shattering effect.

Links

Our source code and live demo can be found at the following links:

Github Repository: https://github.com/CharlieQiu2017/cpsc578-final-proj
Live Demo: https://charlieqiu2017.github.io/cpsc578-final-proj/

References

[1] Bjork, “Hyperballad”, *https://www.youtube.com/watch?v=te7PgEGj0z0*, Accessed: 2023-05-09

[2] Karim Maaloul, “The frantic run of the valorous rabbit”, https://codepen.io/Yakudoo/pen/YGxYej, Accessed: 2023-05-09

[3] Juwal Bose, “Endless Runner Game”, https://gamedevelopment.tutsplus.com/tutorials/creating-a-simple-3d-endless-runner-game-using-three-js--cms-29157, Accessed: 2023-05-09

[4] Elena, “Low Poly Plane”, https://codepen.io/carrot-e/pen/WGZJBe, Accessed: 2023-05-09

[5] mediocregamestudio, “Smash Hit”, https://www.youtube.com/watch?v=XL5as-w7Q1g, Accessed: 2023-05-09