Aliens in Cahoots

Roma Bhattacharjee and Zachary S. Siegel

Click here for a video demo of our project.
Click here for a live demo of our project.

Abstract

Introducing Aliens in Cahoots, a new puzzle-based 3D video game! Drawing inspiration from “Run”, “FireBoy and WaterGirl,” and “Baba Is You,” our key game mechanics allow you to switch between controlling two players, and *change gravity*. We use ThreeJS for rendering and CannonJS for the physics interactions, with the player and level models designed in Blender. Implementing the camera and player movement in particular proved to be an interesting challenge, with our final product teaching us about quaternions and transformations between coordinate spaces.

Introduction

The goal of our project was to build a puzzle-based video game that combines interesting game mechanics, strategic thinking, and physics. From our experience, modern video games have trended towards prioritizing interesting plots over interesting game mechanics, to the detriment of the video game experience as a whole. We wanted to build a game that focused on interesting combinations of simple mechanics that drastically change the gameplay experience. We drew inspiration from three games: Run, FireBoy and WaterGirl, and Baba Is You. In addition, we wanted our game to be in 3D to make it more realistic.

From Run, we enjoyed the platformer style game where the character has to jump across platforms and avoid falling into the void due to the force of gravity. In Run, if the player runs on to the side of the wall, gravity flips in that direction. We also incorporated switching gravity when the player runs into certain interactive elements, as in Run.

From FireBoy and WaterGirl, we implemented dual-character control, where you are in control of two characters. In FireBoy and WaterGirl, you control each of the two characters with a different set of keybinds, whereas in our game since it is in 3D, you press a keybind to change perspectives between the two characters. We designed our levels intentionally so that to beat the game, both characters need to work together and get to their respective landing pads.

From Baba Is You, we took inspiration from their puzzle approach of beating the game. We tried to add interactive elements that players can push around and leverage, in a puzzle environment, to solve the level. For example, we frequently utilize cubes that can be pushed around, and the character needs to jump on the cubes in order to climb up to new platforms in each level.

Since we also wanted to make our game three dimensional, we needed to use libraries that could handle the rendering and physics collisions and interactions. We elected to use ThreeJS for the 3D rendering of the bodies, and CannonJS to implement the physics interactions. Since the player is moving around, interacting with the ground, pushing objects, and standing on other players, it is essential that we have a model for physics. Finally, to make the models for the players and levels themselves, we used Blender and exported levels and object models to .glb files, which was easier than designing the levels in ThreeJS and CannonJS themselves.

We think that these three components work well together as the mechanics we propose – switching the direction of gravity and controlling multiple players – are conceptually simple, which made it easier to implement and also easier for the player to control. And yet, they dramatically change the gameplay making things more interesting.

At a high level, there are several features that we needed to implement for our game to run smoothly. We divide the methodology into three sections. In the first section, we will talk about implementing CannonJS and the broad physics interactions that take place within our game. In the second section, we will talk about the player controls, including how we move the player about the environment, especially given that our game allows for changes in the direction of gravity. In the third section, we talk about how we position the camera relative to the player and the direction of the player.

Methodology

1: Physics Interactions/Cannon JS/ThreeToCannon

We felt that including CannonJS as one of our dependencies would be useful as we built out the core functionality of the game. This is because we wanted to build levels in a software like Blender, to then import into the game, and we wanted a library to handle basic physics simulation tasks (like applying forces, simulating friction, and detecting collisions) for us. CanonJS is a physics library that automatically handles object movement and interactions, whereas if we used ThreeJS alone, we would need to manually compute the object interactions.

Notably, a ThreeJS Mesh is not the same as as CannonJS Body. A mesh is what is visually shown on the screen, but for purposes of interaction and physics, we need to specify a CannonJS body so the code knows how to handle collisions. When building our levels, it would have been a hassle to have to manually specify the collision shapes for each Three.js mesh that we created, especially because we were building them in Blender, so we would have needed to manually get the Blender coordinates for each shape and re-create it as a Cannon.Js body. This would also have been prone to mistakes, as it is difficult to visualize where Cannon.js bodies and shapes are in space since they are invisible. We found a library called three-to-cannon that solves this problem for us—it can take a Three.js mesh and return a Cannon.js shape that defines a

rough collision body for that shape. The options are bounding box, cylinder, sphere, convex hull, and mesh (though the latter option is inefficient and therefore not desirable). Choosing a box or convex hull collision shape was sufficient for our purposes, since we designed most of our levels to be made of blocky components like walls, platforms, or cubes. With this library, the Cannon.js bodies were automatically created for us – all we had to do was import the Blender file, which contained the Mesh that Three.JS could use.

For the physics interactions, we had to differentiate between dynamic and static bodies. Static bodies are used for objects that cannot move, such as the land or the pads that the player needs to stand on to complete the level. Dynamic bodies are meant for movable objects, such as the player and the red cubes that can be manipulated in the environment. We define each level's land as a .glfs file, (e.g. 'level6.glb'). Our code enumerates through every object in the levelX.glb file, and creates a Cannon Static body for that object. This is because we assume the land objects are not moveable.

Regarding physics interactions, we need to ensure that objects also have mass. For instance, when we create the cube, which is a dynamic body, we need to assign it mass so that it can be pushed around by the player. We abstract each object into a class which contains both the Three.Js Mesh and Cannon.Js body for modularity. In certain cases, we need to manually define a bounding-box around the object, such as cases of the player and the landing pad, when the mesh is too complicated for three-to-cannon to convert the Three.Js mesh into a Cannon.js body. In this case, we just manually draw a box around the Mesh which serves as a good approximation.

2: Player Controls and Gravity

As discussed above, one unique feature of our game is the fact that each player can have a unique local gravity force that is not necessarily the same as the world's gravity. This meant that we couldn't use the global gravity setting in our Cannon.World object. So instead, we disabled global gravity and simply added a distinct gravity force to every dynamic body that we instantiated into our levels. This allowed us to control gravity on an object-by-object basis and even change its direction in response to certain game events or triggers. The way the code is set up allows us to set any generic direction of gravity—we are not limited to just six axis-aligned directions. While most of our levels do stick to these six directions for ease of level design, we do demonstrate this functionality on our ending scene, which allows the players to walk around on a sphere.

One downside of disabling global gravity was that Cannon.js no longer properly applied friction upon contact between materials. To mitigate this, we found a frictionGravity property on the world object, which is meant to be a guide to Cannon.js when applying friction (but is not actually applied to objects as a true gravity force). This seems to have resolved our issue.

We also had to choose between applying movement as a force versus directly updating the player body's position vector on each update call. There were benefits and drawbacks to each approach. Applying movement as a force resulted in cleaner collisions with walls, but it was also

harder to control the unbounded acceleration the player would experience while walking (it felt similar to walking on ice), and increasing friction too much would make the player skip along surfaces instead of smoothly walking along them. Since we wanted to focus our efforts on the more unique game mechanics instead of tinkering with physics contact materials and friction values, we went with a different approach—we read the input from 'WASD' into an input vector, apply a transformation to account for gravity and orientation (which we dive into in more depth in the camera movement section below), and then update the position vector directly. The downside to this is that players can walk slightly into collision bodies, and upon letting go of a movement key they are pushed back. But the upside is that movement on the ground is much snappier, which is ultimately more useful for a platformer-type game.

3: Camera Movement and Player Orientation

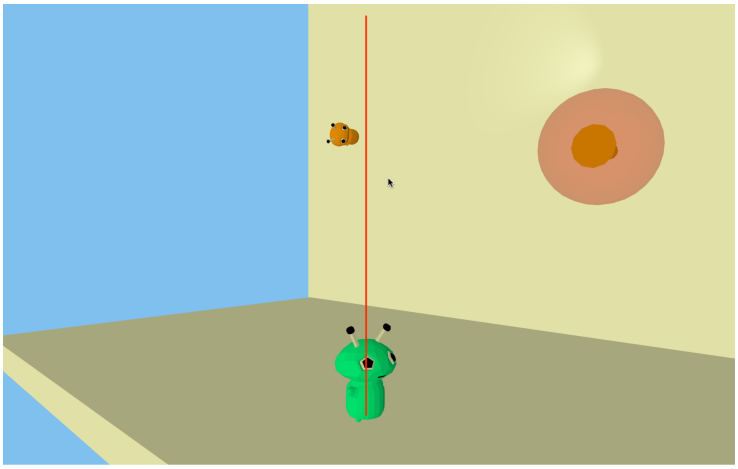
Because there are no restrictions on the orientation of the player (gravity can take on any direction), the task of translating 'WASD' input into player movement and controlling the camera were non-trivial—in particular, we made extensive use of quaternions and transformations to achieve our smooth controls, no matter the orientation of the player.

We first discuss camera movement. In a normal game where gravity is fixed, the camera can be controlled by updating spherical coordinates using the mouse movement. In our case though, we first have to rotate the camera in world space according to the quaternion that defines the orientation of the player. In particular, we use the quaternion that would rotate the "up" vector (0, 1, 0) to the opposite of the gravity direction (also referred to as the local normal vector) for the active player. This quaternion can then be applied to every transformation we do in the future—in practice, this is done by simply multiplying the quaternions together.

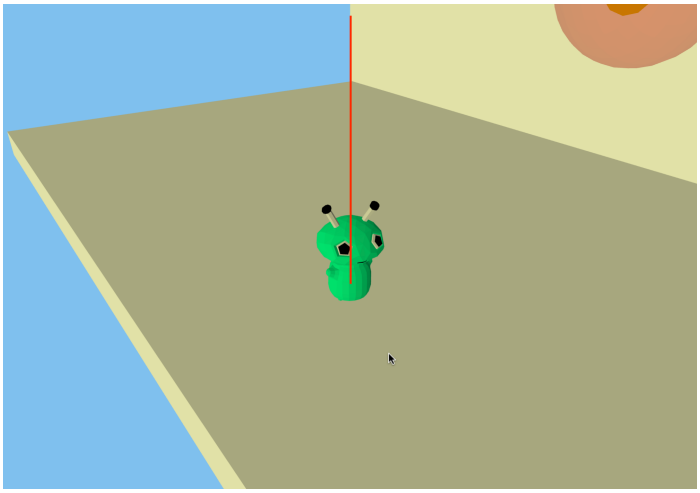
Once the camera is positioned, we have to point it at the player. Again, in a normal game, we could use the standard Quaternion.lookAt() function, but this function fails when players are upside down or sideways. Thus, we re-implement the behavior by breaking it into two parts—first, we have to rotate the camera about the axis of the player's local normal vector, and second, we pitch the camera down until it is viewing the player.

The first step is achieved as follows: we define a `camToBodyOrth` vector, which is the component of the vector from the camera to the player that is orthogonal to the local normal axis of the player. We then calculate the signed angle between this vector and the direction that the camera is looking, to determine how to rotate the camera to face the local normal axis of the player.

The next step involves finding the axis of rotation in world space, which can be found by taking the positive x direction of the camera in its own local space and applying the camera's quaternion to convert it to global space. Then, the desired angle to look at the player is calculated similarly to in the previous paragraph, and we rotate the camera accordingly.

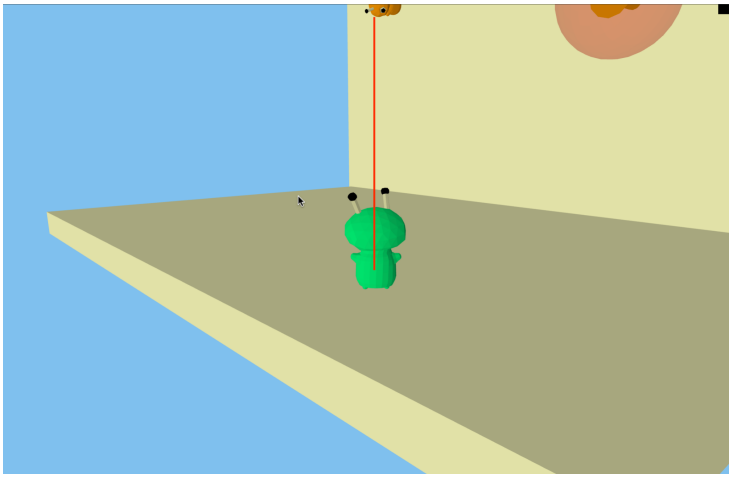


After positioning the camera using spherical coordinates, we must make it face the player. The above figure shows the alignment of the camera to ***face the axis*** *defined by the Player's local normal direction (which is the inverse of its local gravity vector). Here, the active player (green) is standing on a wall—the axis defined by its normal direction is drawn in red. Note that the camera has not yet been pitched down to center the player.*



We then pitch the camera on its local +x axis, so that the player is in the center of the screen.

The final step is to rotate the player mesh around its normal axis, so that it is facing away from the camera at all times. This orientation is then used to transform the local input velocity vector that we retrieve from the player's "WASD" input into global space (in order to let the player walk diagonally on walls just by pressing the "W" key, for example).



Finally, the player is rotated about its normal axis to be facing away from the camera. This is the orientation that is used when converting the local input velocity vector grabbed from "WASD" into a global vector used to displace the player's Cannon.js body.

Results

Overall, we considered our project to be a success. We had some of our friends play the game during the development process to solicit feedback, and they all seemed to enjoy the game very much. For example, after hearing that the game controls weren't clearly defined, we added text to the game to teach the player how to play. After hearing that one of the levels was difficult to figure out, we added a hint in text to hint the user to jump on the other player, which aided success. It is hard to develop a formal notion of success in a video game, but we informally surveyed the playing experience of our friends to see what we could do to improve the game mechanics. We also tried to add some levels that were particularly frustrating, such as the last level, which requires that you restart the entire level if even one of your characters falls into the void, as we think this made the game a bit more challenging and added some stakes to your performance.

In addition, throughout the design process, we executed multiple experiments to isolate that individual components of our project were working. For example, we started by testing Cannon.js and ensuring that if we dropped the player from high up, it would hit the floor and bounce. Then, we added player controls to see if the movement worked (but without gravity). Then, we tested the combination of player controls and gravity. By building up our project in this iterative process, we were able to more easily debug individual components.

Discussion and Conclusion

Overall, we believe our approach is promising due to the modular design principles we used to make the game. Adding additional levels is easy: they are designed in Blender, you import them into ThreeJS, and create a new level file. Adding components to a level is easy, since we encapsulated objects to contain both the body and mesh, and we defined custom objects such as gravity launchers, blocks, and pads to start and end the level on. In addition to making the code more readable, this allows the game to be easily expanded both by us and other developers.

Having built the game, we can think of some ways to improve our modularity approach. We noticed that certain components are abstracted to a degree that they don't need to be, and other components could use more abstraction. For example, our implementation for the Landing Pads are somewhat over-engineered—they always advance the level, look the same, and simply require coordinates to place it. They could be of their own class to simplify things, rather than making them a LevelObject, which is generic. On the other hand, more basic objects such as cubes or buttons don't need their own class since we want to preserve the ability to customize them. Different buttons have different functions throughout the game, and we want to keep that more abstract.

For follow up work, we think of three distinct directions that would improve the quality of the game:

a) *Improving the camera mechanics:* Currently, the camera can clip through objects and walls as it remains a fixed distance away from the player. Ideally, we should project a ray from the player and see the distance of intersection and place the camera right before that intersection to prevent it from running into walls. We did experiment with this approach but ran into bugs which we were not able to resolve before submission.

b) *Improving the graphics of levels:* Currently, the levels are relatively barebones with flat floors and walls. It would improve the aesthetic quality of the game to add more decorations, colors, and objects to make the game more enjoyable. In addition, it would be great to make more objects interactive to give the game more of a puzzle feel rather than a parkour feel, which some of the levels seem to have.

c) *Adding additional levels and game mechanics:* Our game only has six levels so far, so additional work should explore adding in new levels. Some ideas that we have for levels are having "lava" that if you touch sends you back to the beginning of the level, have the player pass beneath the blades of a windmill so you have to time your travel perfectly, and a maze that both players need to navigate where the landing pads are in different areas of the maze.

We took away several lessons when designing this game.

a) *Importance of Modularity:* As the game became more and more complicated, we realized the importance of modularity, abstraction, and efficient code design. As it became impossible to keep our codebase organized, we needed to make the code modular and have classes for levels, objects, and utility functions.

b) *Difficulty of Level Design:* Designing challenging but not over-challenging levels was far more difficult than we thought. To start, physically placing objects at appropriate distances away from each other to make the parkour mechanics was difficult, but the creative ideation process was also difficult – designing levels that were not repetitive and required original thinking, like a puzzle, was more difficult than we imagined.

c) *Quaternions:* We learned about the considerations of dealing with orientation and quaternions, and the types of computations that are involved in doing those alignments, involving linear algebra.

Contributions

Roma Bhattacharjee worked on camera orientation and player movements, designed levels and models that we used, and wrote the writeup.

Zachary Siegel wrote the writeup, designed levels for the game, and added CannonJS physics interactions to the ThreeJS environment.

Works Cited

https://threejs.org/docs/

https://schteppe.github.io/cannon.js/docs/

https://www.coolmathgames.com/0-run

https://fireboyand-watergirl.com

https://store.steampowered.com/app/736260/Baba_Is_You/

https://pmndrs.github.io/cannon-es/