COS 426 Final Project: Marble Hero

Nicolas Schmidt   Dharit Tantiviramanond   Lulu Zhong [*]

May 12, 2015

1 Introduction

1.1 Goal

In this project, we set out to build a simple, yet fun, interactive game. We were inspired by games like Marble Madness that present a game environment that allow for totally abstract mechanics and imagery. Like in Marble Madness, our game is about steering a ball through a physical simulation. We used abstract imagery produced for the unreleased game Starrider to background our game's colorful arena.

The mechanics of the game are straightforward. The camera follows the movement of a sphere as it falls and interacts with it by adding trampolines along its trajectory that cause it to bounce in different directions. The ball moves in two dimensions along the X-Y plane, and is bounded in all four directions by walls. The environment itself is 3D. The user adds trampolines by clicking and dragging in the direction of the plane's normal. The bounciness of the trampoline is proportional to the distance that the user dragged.

The goal of the game is to accumulate points by collecting orbs that are floating around the environment. The player wins by gathering 426 points!

1.2 Previous Work

We chose to build our project on top of the framework given to us in homework 4. This was because we wanted to utilize the particle engine framework to incorporate particles into our game, and decided that the code structure could be outfitted to suit our purposes. The framework offers an excellent interface for defining the behavior of particle systems, and already had a scene, camera, and renderer set up that we could access. Also, we were familiar with certain parts of the framework already, and took the opportunity to dive a little deeper into the three.js API.

One part of the framework that was less intuitive was handling interactions between the particles and the scene objects. Particles are only able to collide with a *collidables* array specified in `SystemSettings.js`,

[*]nschmidt@princeton.edu, dtantivi@princeton.edu, luluz@princeton.edu

and not with objects added to the scene in `createScene()`. We wanted the collidables to also be a part of the visual scene, and decided that it would be easier to define collisions with the scene objects themselves instead of having to define corresponding elements in the collidables array. The framework does not provide any functionality to support this.

1.3 Approach

As previously stated, we outfitted the particle engine from homework 4 for our game. This approach was useful because it saved us the work of implementing a WebGL scene, renderer, and physics engine from the ground up. Our implementation should work well under most circumstances as it is a relatively lightweight game in terms of CPU usage and memory footprint. However, we are uncertain of the scalability of the game, and the addition of too many scene objects may cause the game to freeze.

2 Methodology

We modified two main aspects of the homework 4 framework in order to build our game: the scene and the particle engine mechanics. The scene modifications included defining the physical space within which the game would exist, adding a orb collection system, creating the particles orbiting the sphere, and making the trampolines. Changes to the particle system included adding a new type of emitter for the bouncing sphere, adding a colorful particle system orbiting the sphere, modifying the collision code, and mapping the camera's position to particle coordinates.

2.1 Scene Modifications

- Game Environment: The sphere is free to move along the X-Y plane in 2 dimensions, but is bounded on all four sides by walls. The walls (seemingly) extend infinitely forwards and backwards in the Z direction. They were given a checkerboard texture to help the user perceive their depth. The background of the scene is a plane with a StarRider run-through video mapped onto it.

  These modifications were mostly implemented by adding objects to the scene within `createScene()` in `SystemSettings.js`. This is the most straightforward way to add objects to the scene.

  We also considered adding obstacles inside of the game space that consisted of planes and spheres, but decided that coming up with creative obstacle placement was less interesting to the project than developing the mechanics of the game in other ways. However, this is an easy way to add variation to the game which will come in handy if we want to create multiple levels.

- Trampolines: Trampolines represent the only way through which the player can interact with the game, and therefore their behavior must be intuitive and precise while still offering enough options to the player to keep the game interesting. The trampolines themselves are comprised of two plane meshes facing opposite directions. We decided to allow the user to create trampolines by clicking and dragging–the trampoline is added at the click position and its normal is in the direction that the user drags out. The trampoline's bounciness is also defined by the distance that the user drags, and its color changes as a visual cue corresponding to the bounciness. The bouncing behavior is implemented as a new collision method that incorporates the sphere's radius as well as the trampoline's bounce factor.

We implemented this behavior by adding event listeners to the jQuery `mouseup`, `mousedown`, and `mousemove` events in `main.js`. The `mousedown` listener adds the trampoline to the scene, and the `mouseup` and `mousemove` listeners set the trampoline's rotation and bounciness factor. We implemented a `getCursorPos` function that returns the position of the cursor on the *canvas* as opposed to the *screen*, in order to find the position at which to add the trampoline inside the scene.

One bug we encountered was that a newly created trampoline would unpredictably spin for a few revolutions before rotating to face the player's mouse in an intuitive fashion. This was because we had been using the mouse position on the canvas to calculate the trampoline normal instead of the position on the screen. Since the camera's position always tracks the ball's position, the normal calculated by the `mousemove` listener followed the direction of the camera for a short period instead of the direction that the user intended. We fixed this by using mouse screen coordinates instead.

- Orbs: The orbs are spinning spheres with a wireframe texture placed randomly in the game space within `createScene()`. We implemented a new collision function that updates a global score variable and removes the orb from the scene upon collision. The score of each orb is inversely proportional to the orb's radius.

2.2 Particle Engine Modifications

- Sphere Emitter: We added a second emitter to the particle engine, as we noticed that this functionality was already built in to the framework. We changed the `_drawableParticles` member of the emitter to be a sphere mesh instead of a point cloud, and set maximum particles to 1 and set a very long lifetime. This allows us to use the particle engine's pre-existing code to update the sphere's position and velocity, and to define collision behavior.

  In retrospect, this may not have been the most logical way to add the bouncing sphere to the system. Since there is only ever one sphere in the game, we did not need the particle engine mechanics to implement the sphere's behavior. Instead, we could have added the sphere to the scene in `createScene()`, and written functions that describe the ball's movement and collision behavior to add to the game's execution loop.

- Rainbow Orbiting Particles: To make the particles emitted from the sphere rainbow colored, we simply took the sine of the position and set that as the color in the updater. The position was scaled by a factor of 10 for aesthetics.
- Camera Movement: To make the camera follow the movement of the sphere, we accessed the camera in `updaters.js` and set its position to be equal to the position of the bouncing sphere particle. The camera was accessible as a member of the global renderer variable.
- Scene Collisions: Instead of defining a separate collidables array that the particles collide with, we wanted to collide directly with the scene objects for clarity and simplicity. To accomplish this, we loop through the scene objects in our custom updater's collision function, and separate the types of objects by their `name` member, which is set during object initialization. This is certainly not the most efficient way to accomplish this, but our game is small enough that this overhead does not get in the way of responsiveness.

3 Results

We did not perform computational evaluations of our game, but decided to focus instead of the playability of the game–specifically its intuitiveness and responsiveness. We found that for the most part, the sphere and the trampolines interact with each other in an intuitive manner, and the addition of trampolines is handled in a straightforward fashion. Directing the ball's trajectory is intuitive as it follows the basic laws of physics, but also not so easy as to make gameplay trivial. The allure of the game lies in becoming accustomed to the camera's movement and the behavior of trampoline placement, as well as predicting the ball's path of travel. Even though it is the purpose of the game, collecting orbs is not difficult and the user can win easily enough so that the game is not frustrating.

4 Discussion

Overall, the homework 4 framework lends itself well for our purposes. However, we may want to change the bouncing sphere to be just a scene object instead of part of the particle system in order to avoid overhead. This involves deeper framework modifications than the ones we performed.
One important technical discussion involves our implementation of collisions with scene objects. For each frame, we currently loop through all objects in the scene to detect collisions. This is inefficient, and only works because our game is small enough. If we were to extend the game to incorporate multiple levels and more complicated scenery, we would have to perform optimizations. One possibility is adding a hashmap of object positions to the objects, and looking up objects for every position that the sphere passes thorugh.

Another minor bug is that if the sphere intercepts a trampoline as it moves perpendicular to the trampoline (it collides with an edge of the trampoline instead of its face), undefined behavior can occur depending on the ball's position in relation to the trampoline, and the ball will bounce off in unpredictable directions. To solve this, we would have to look more closely at our collision function and perhaps disallow the sphere mesh from overlapping with the plane mesh.

Apart from the above, further work would include adding more playable levels to the game, or increasing the size of the game space. Levels could be made more interesting with the addition of scene obstacles that obstruct the ball's movement, or attractors that make it harder to control the ball's trajectory. Also, more visual effects could be added, such as particle emittance during collisions and fun animations after the player has won. Furthermore, audio effects could be incorporated into the game.

To complete this project, we had to delve deeper into the three.js framework. Specifically, we had to learn about the renderer, scene, and camera interactions that define the basic setup of the environment, the connection between three.js objects and HTML DOM elements, and inheritance model between Object3D, Mesh, and Geometry. We also had to understand the difference between Geometry and BufferGeometry and how/when to use each. Furthermore, we had to understand the homework 4 particle engine and emitter framework in order to modify it, for example, we had to understand the difference between scene elements and the collidables array available to `updaters.js`.

5 Conclusion

We successfully implemented a small game using three.js and the homework 4 framework. Although the game is simple, it is visually stimulating and incorporates enough gameplay elements to keep it interesting. During the process of development, we learned a lot about three.js and the practical considerations involved in the development of a game. We also learned to value and focus on the interactivity aspect of gaming, as the feel that the player is left with depends almost entirely on the intuitiveness of gameplay.