Spheres vs. Cubes

Griffin Schneider
Northeastern University
gcs@ccs.neu.edu

TJ Higgins
Northeastern University
thomashiggins10@gmail.com

ABSTRACT
In this paper, we describe the implementation of our 3D Computer Graphics final project, *Spheres vs. Cubes*. *Spheres vs. Cubes* is a 3D third-person 'platformer' video game with physically-accurate physics simulation.

Categories and Subject Descriptors
I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism – *animation, color, shading, shadowing, and texture, virtual reality.*

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
Keywords are your own designated keywords.

1. INTRODUCTION
For our 2D project, this group created *Circles vs. Squares* a 2D physics-based platformer. We found this to be an excellent learning experience and an enjoyable project, so we decided to extend the idea into three dimensions for our 3D project, and *Spheres vs. Cubes* was born.

While creating *Spheres vs. Cubes*, our goal was to gain some appreciation of the work that goes into creating 3D games, an area in which our group did not have experience prior to this project. Although *Spheres vs. Cubes* is obviously a much smaller project than even the simplest commercial 3D games, many of the general concepts and architecture decisions in our project are also applicable to the majority of 3D games.

We will discuss previous related works which we drew inspiration from to create *Spheres vs. Cubes*, as well as various technical aspects of the game and the techniques that we used to implement those features.

2. RELATED WORK
In creating *Spheres vs. Cubes*, we were influenced by a number of pre-existing 3D third-person platformers. When examining the 3D platformer genre, one cannot help but discuss the influence of *Super Mario 64,* the game that essentially defined the genre in 1996 with the launch of the Nintendo 64.

2.1 Super Mario 64
Super Mario 64 was one of the first games to expand the platformer genre into three dimensions [2]. This was made possible by the analog control stick of the Nintendo 64 controller, which allowed the player to move in any horizontal direction, and an excellent dynamic camera system that enabled free-roaming exploration without the player being burdened by direct camera control.

Super Mario 64's camera system is widely lauded as one of the most revolutionary features in the game [1]. We considered implementing a similar automated camera system for *Spheres vs. Cubes*, but decided not to due to time constraints. Also, the camera in *Super Mario 64* often snaps to certain pre-set angles when Mario enters certain points in a level that the normal camera algorithm would not handle well. Implementing this feature would have required adding some sort of camera-snapping-zones to our level editor, which we felt already had more than enough hotkeys. Instead, we decided to leverage the classic WASD+mouse controls found in a number of PC games that allows the player to directly control the camera at all times. If we were to attempt to move *Spheres vs. Cubes* to a console platform, we would likely need to implement some sort of more automated camera system.

Figure 1. *Super Mario 64* - Mario about to get hit by a Bullet Bill

The 'special feature' that *Spheres vs. Cubes* brings to the classic 3D platformer genre is its accurate physics simulation. All 3D

Conference'13, Month 1–2, 2013, City, State, Country.

platformers generally require some type of physics simulation, since they must at least enable the player to jump and move around, but the focus of these simulations is generally not on physical accuracy. In *Spheres vs. Cubes*, we used a physics engine to make world objects move similarly to how they would in real life. This is a relatively simple way to add a sense of realism to a game without necessarily increasing the graphical complexity. One of the most well-known games to make use of physics-simulation-driven gameplay is the extremely popular game *Angry Birds*.

2.2 Angry Birds

Angry Birds is a 2D game in which the player flings various birds into elaborate, flimsy structures in an attempt to destroy pigs [3]. The dynamic destruction of these structures is powered by Box2D, the same 2D physics engine that we used for *Circles vs. Squares* [5]. *Angry Birds*'s use of Box2D enabled the implementation of visually-pleasing, physically-accurate destruction with a minimum amount of effort from the development team. Physical accuracy was an important factor in *Angry Birds*'s extreme popularity and eventual spawning of seemingly endless new versions.

Figure 2. *Angry Birds*'s physics-simulation-based gameplay

The inspiration that we drew from *Angry Birds* and the massive amount of Box2D-driven games that followed it was that people love physics simulations. After our previous experience creating a physics-driven 2D platformer, it was a natural next step for us to bring physical simulation to the 3D platformer genre.

3. METHOD

3.1 Physics Integration and Control System

Spheres vs. Cubes uses the Bullet physics engine [6] – specifically JBullet [7], a port of the C-based Bullet to the Java programming language. Since we were already planning to use Processing, which is Java-based, to create the graphics for *Spheres vs. Cubes*, it was easier to use JBullet rather than deal with JNI to interface between Java and C code. We considered that using the Java port rather than the C-based version could lead to performance issues; however, preliminary testing showed that using JBullet was unlikely to detract from our ability to obtain 60 frames per second for the relatively simple physical situations that we planned on simulating.

Bullet is a professional-level physics engine containing a vast array of advanced features, including large numbers of optional optimizations that were generally out of scope for this project. Due to relatively tight time constraints for the project and our group's lack of experience with Bullet and 3D physics simulation in general, we decided to use only a small subset of Bullet's features for *Spheres vs. Cubes*. The main use of Bullet in our game is for simple rigid-body simulation of spheres and rectangular prisms, plus some basic collision detection for determining when the player can jump and when the player has been been hit by an enemy bullet.

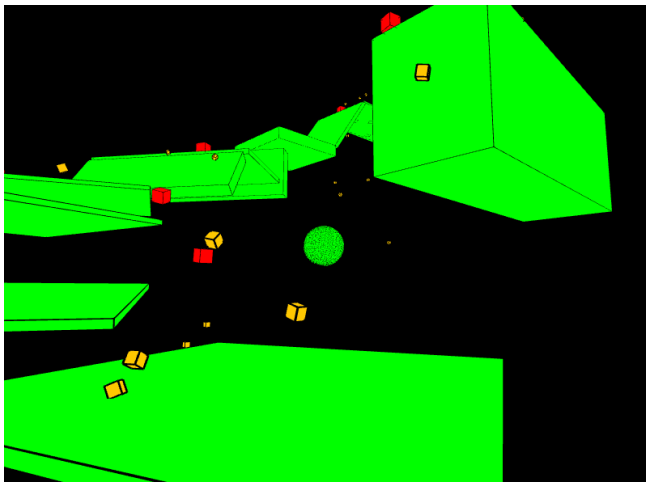

Figure 3. Example Spheres vs. Cubes gameplay screenshot

Since the sphere controlled by the player is an object in a physics simulation, we had to take care not to disrupt the simulation when the sphere is moved by the player's controls. The naïve implementation of movement controls, which is often sufficient in games without complex physics simulation, is to simply translate the player's body when a directional key is held. When the player's body is an object in our Bullet simulation, however, translating the player's body can often have adverse effects on the physics simulation. Since actual physical bodies do not move in discrete increments, abruptly shifting the player by a fixed amount every frame can lead to inaccuracies when the player collides with other objects, or when an external force is applied to the player by the physics simulation. So instead of translating the player's body, we apply a small impulse to the body for every frame during which the directional key is held down. Impulse is the integral of a force over time, so our application of a given impulse each frame is equivalent to applying a small force to the player's body for the entire time period represented by the frame during which the movement key was held down.

This repeated application of impulse still does not produce an acceptable control system – the player can accelerate without limit, and when the movement key is released at high speeds, the player continues to move since there are no forces slowing it down, except perhaps a small amount of friction from the ground.

Fixing the first problem is quite easy – we simply do not apply the movement impulse if applying that impulse would cause the player to move over some pre-set maximum velocity. Solving the second problem, however, presents more possibilities. We first tried using Bullet's built-in linear damping – bodies in Bullet have a linear damping attribute, and setting this attribute to a positive non-0 value will cause the linear velocity of the body to be scaled downwards each frame. Unfortunately, applying a linear damping to the player body affects velocity in all directions, when we really only want to decrease the player's velocity in the horizontal plane. Applying a linear damping to the player body causes jumps

to feel too 'floaty' – the damping causes the player to spend an unnaturally large time at the peak of the jump. To fix this problem, we implemented our own variation on linear damping. For each frame during which the player is not holding a movement key, we find the player's velocity, project it onto the horizontal plane, normalize that projection, reverse its direction, multiply it by a small constant damping factor, and apply the resulting vector as an impulse to the player's body. This has the effect of counteracting any motion in the horizontal plane whenever the player has released all movement keys.

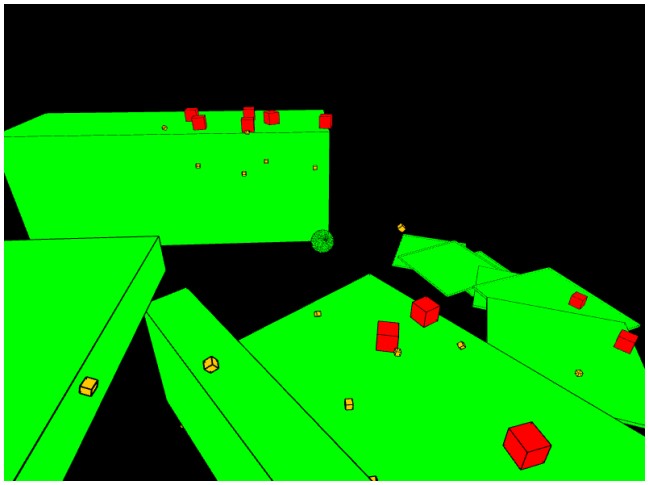

Figure 4. Another gameplay screenshot. The player is now smaller due to loss of health.

Implementing the player's jump presented a different set of problems than the implementation of horizontal movement. It is easy to implement the jump itself – just apply an upward impulse to the player – but the difficulty lies in determining when the player is capable of jumping. If we simply apply an impulse whenever the jump button is pressed, the player can essentially fly by jumping repeatedly in midair. To figure out when the player should actually be able to jump, we implemented a collision handler that listens to collision events from Bullet involving the player and uses this information to keep track of all the bodies that the player body is currently touching. Then, when the jump button is pressed, we apply the jump impulse only if the player is currently touching some other body.

This jump implementation is still not perfect, although our group decided that it was sufficiently robust for our project. For more accurate jump ability determination, we could check if any bodies are intersecting a sphere slightly smaller than the player sphere and slightly below it. This would do a better job of only enabling the player to jump if it is standing on something – our current method allows the player to jump even if the body it is touching is just the side of a vertical wall, which allows the player to climb walls of any height.

3.2 Camera Control

Spheres vs. Cubes uses a third-person camera that is always directly controlled by the player. Many third-person games have a camera that attempts to move automatically to give the player the best possible view of the game, but we anticipated that such a system would be too difficult to implement for this project. Instead, we attempted to give the player enough manual control to attain a satisfactory camera view in a variety of situations.

It seems unlikely that the player would ever want to look away from their own body, so we decided to always have the camera centered on the player. We also decided that the camera should always have a vertical up-vector, so that it is always clear which direction gravity is pointing in. These two restrictions are the only restrictions on the player camera, however – the camera can be positioned at any point in space, as long as it is pointing at the player and oriented with the same up vector. Given these restrictions, we decided that spherical coordinates were the natural choice to represent the position of the camera.

3.2.1 Spherical Coordinates

Spherical coordinates are a method of addressing any point in a 3D space. Each point is defined uniquely by a radius r and two angles, θ and φ. Spherical coordinates essentially extend the 2D polar coordinate system by adding an additional angle. The meaning of each of these angles can be seen in figure 2 below.

We chose spherical coordinates to represent our camera position because of the convenient correspondence between the parameters of spherical coordinates (r, θ, and φ) and the control inputs used to move the camera. The radius r corresponds to the zoom level of the camera, and is controlled by moving the mouse wheel up or down, or by pressing the "+" and "-" keys to zoom in and out respectively. φ represents the horizontal rotation of the camera, and is incremented or decremented by moving the mouse left or right. Note that when moving the mouse to the left, the camera actually moves to the right so that it will be angled more to the left. Finally, θ represents the vertical angle of the camera and is controlled by moving the mouse up or down. Again, the camera moves in the opposite direction of the mouse, so that its orientation will change as the player expects.

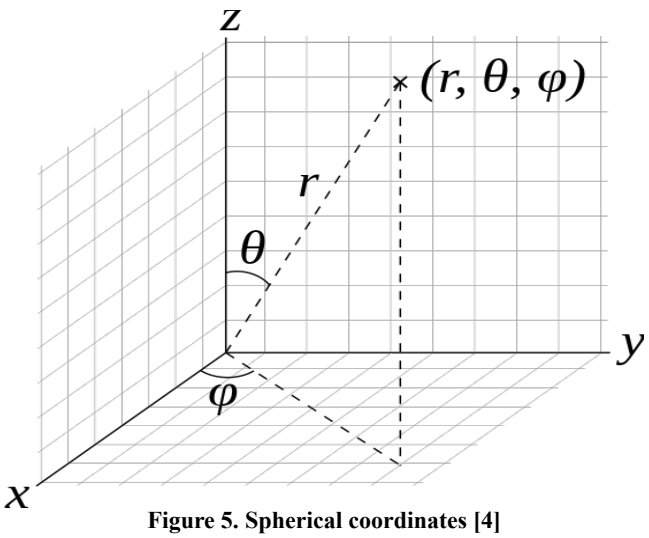

Figure 5. Spherical coordinates [4]

3.3 Game Logic

Most professional games have extremely complex game logic to maximize performance. While this is crucial for production games it is time consuming to implement. Since the game is fairly simple it was not necessary to add further multi-threading. All of the logic is handled in one main thread in the game scene.

Since *Spheres vs. Cubes* utilizes Processing it takes advantage of its draw method. This in turn is the main loop of the game. It simulates the physics of the world. It then checks for collisions

between objects and notifies them. Then it updates and draws all of the objects that existed prior to the start of that loop tick.

The only other threads are Processing's key and mouse callbacks. There is a input manager that handles these callbacks, and allows the information to be used in the main loop without having to worry about thread safety. This is used to determine which keys are currently being held down.

3.3.1 Cubes and Bullets

Each object in the world also has its on update method. The red cubes in the game are the enemies of the player. These enemies are constantly trying to destroy your sphere. Currently they do not move, but are completely effected by physics. The player can push them off edges, but must avoid their bullets. Bullets that a shot towards the player are smaller orange cubes. Each enemy has its own timer that shoots a bullet on a delay. If a bullet comes in contact with a player it decreases their radius or life.

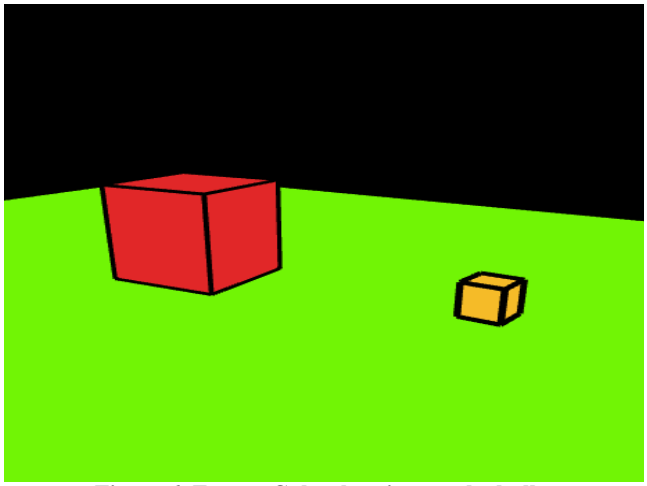

Figure 6. Enemy Cube shooting a cube bullet

3.4 Level Editor

Level design is often a crucial factor in the enjoyability of platformer-style games. If *Spheres vs. Cubes* were being produced commercially, it would likely have a full stand-alone level editing suite that could create arbitrary solid polygons to be used as the walls and floors of our levels. Since *Spheres vs. Cubes* is actually a game being produced by two people in a few weeks, we decided to hack our level editor into the game as quickly as possible. To decrease the amount of work required to implement our level editor, we decided that the editor need only enable the editor to place arbitrarily-sized, not necessarily axis-aligned rectangular prisms at arbitrary points in 3D space. In short, our goal for the editor was to be able to create any possible arrangement of boxes. To enable this, there are a few parameters that our editor must use to place each rectangle – the rectangle's three-dimensional size along each of its sides, the 3-dimensional position of the rectangle's center, and some way of representing the rotation of the rectangle. To represent rotation, we chose to use a system similar to our representation of the camera's rotation with spherical coordinates. We use two angles, θ and φ, to determine the direction of the normal vector of one of the faces of the box being placed.

Our choice of parameters for box placement was based on the correspondence between these parameters and the parameters used to represent the player's position and camera's rotation. When the player places a box in the level editor, we derive its position by finding a point slightly in front of the player's position. We take the θ and φ angles for the box's normal directly from the θ and φ angles of the camera's spherical coordinates. Thus, when the player places a box, the box will show up slightly in front of the camera, with one side of the box parallel to the view plane of the camera.

Once the box has been placed, the level editor provides keybindings to increase or decrease the size of the last placed box along each of its 3 axis. Thus, the editor provides a way to set each of the previously mentioned parameters for each placed box. Also, for convenience, the editor has keybindings to move the last placed box in the positive or negative direction along each axis, and to increment or decrement θ or φ.

Once we created our level editor with these specifications, we realized that there was no easy way to place perfectly axis-aligned boxes. The continuous nature of the mouse-controlled camera makes it difficult to place multiple rectangles so that the walls form right angles with the ground. To overcome this defect, we added additional hotkeys to the level editor that set θ or φ to various pre-defined settings. The editor has keybindings to set φ so that the camera is facing along either of the horizontal axis in either direction, and to set θ so that the camera is looking straight ahead, directly up, or directly down. This allows the player to easily create axis-aligned boxes that intersect with one another at right angles.

3.5 Scene Manager

Spheres vs. Cubes like most video games has a scene manager. With the time constraints it is a simple implementation that can only cycle scenes in and out. It does not handle transitions although that could be a possible future improvement. All stages of the game are scenes. The main menu is a scene that transitions to the game that then proceeds to the game over scene. At initialization of the game the menu scene is set to be the current scene. To change scenes a new scene is created, and set to be the next scene. Then in the main update loop it checks to see if there is a next scene and swaps out the current scene.

A problem that had to be overcome was the initialization of the important elements on the transition as apposed to the creation of the scene itself. Since the scenes are swapped during the update loop it caused issues if the scene was initialized before it got swapped in.

4. RESULTS AND DISCUSSION

In our project proposal for *Spheres vs. Cubes*, we discussed a number of features that were not implemented in the final version due to time constraints. Our project proposal failed to account for the necessity of a level editor, a fact that we realized only after attempting to begin hard-coding our levels by hand. Before beginning development, we assumed that creating a three-dimensional level editor would be too difficult, and that it would be less work to just create a few levels by setting the coordinates of each object manually. Unfortunately, we soon realized that interesting level design would be crucial to the overall enjoyability of the game. We also realized that creating a moderately-sized interesting level requires a surprisingly large amount of ground objects with a wide range of sizes and rotations, making it extremely difficult to create good levels without some graphical interface. We were forced to drop a few graphical features (most notably, particle effects) mentioned in our project

proposal in order to put together a workable level editor. We also decided to restrict our level editor to placing only rectangular prisms as ground bodies to reduce its complexity, as mentioned above. While it is unfortunate that other features had to suffer for the sake of the level editor, the editor allowed us to create levels that showcase our game much better than anything we could have created by hand in the time allotted. Also, our level editor implementation allows us to create a *Matrix*-like 'bullet time' effect which makes our demo video much more interesting.

Our project proposal for *Spheres vs. Cubes* was also hopeful that we would be able to implement a lighting system using Processing's light API. Unfortunately, we experimented with Processing's lights and did not achieve a satisfactory effect, so we decided not to use lighting effects for our game. The main problem with Processing's light API was that it did not implement shadows, which made it difficult to create rooms with distinct lighting effects (since the lights from one room would simply go through the walls into another). We decided not to implement our own shadow system using shaders, which would have been necessary to achieve the lighting effects that we were looking for. If we had additional time to continue working on *Spheres vs. Cubes*, a shadow implementation is likely one of the first things that we would attempt to add, as it would allow us to add a wide range of interesting lighting effects to the game.

5. CONCLUSION

For our 3D project, we created *Spheres vs. Cubes,* a 3D, third-person, physics-simulation-based platformer. We drew influence from third-person platformer games such as *Super Mario 64* and from the relatively recent wave of popular physics-simulation-powered games exemplified by *Angry Birds*. While we did not reach all of the somewhat lofty goals set forth in our project proposal, we did manage to create an enjoyable, relatively good-looking game under difficult time constraints, in addition to implementing an un-anticipated level editor mode for the game. While our level editor may be somewhat lacking in usability due to its vast amount of hotkeys, it was still a great improvement over hard-coded level creation and allowed us to create the levels that come with *Cirles vs. Squares*.

Overall, we believe that we have successfully solved the problem set forth in our project proposal - “I am bored.” *Spheres vs. Cubes* has managed to alleviate my boredom on a number of occasions, despite the seemingly endless number of times I've had to play the first level during testing.

6. REFERENCES

[1] "Everything Old-School is New Again". *Nintendo Power* (Future Publishing) (Winter Special 2008): 42. Winter 2008.

[2] “Super Mario 64”. *Wikipedia.* Accessed April 19, 2013. http://wikipedia.org/wiki/Super_Mario_64

[3] Rovio. “Angy Birds”. *Angrybirds.com.* Accessed April 19, 2013. http://www.angrybirds.com/.

[4] “Spherical Coordinate System”. *Wikipedia.* Accessed April 19, 2013. http://wikipedia.org/wiki/Spherical_coordinates .

[5] Humphries, Matthew. March 2, 2011. *Box2D Creator Asks Rovio for Angry Birds Credit at GDC. Geek.com.* http://www.geek.com/games/box2d-creator-asks-rovio-for-angry-birds-credit-at-gdc-1321779/

[6] Game Physics Simulation. “Bullet Physics Library”. Accessed March 23, 2013. http://bulletphysics.org/wordpress/

[7] jzek2. “JBullet – Java Port of Bullet Physics Library”. Accessed March 23, 2013. http://jbullet.advel.cz/