CoronaTime

COS 426 - Final Project Writeup

Audrey Cheng and Christine Lu

https://github.com/Audrey1656/coronatime

https://audrey1656.github.io/coronatime/

Abstract

CoronaTime is an "endless runner" game in which the player is a white blood cell, moving through a blood vessel. Along the way, the player must kill as many viruses as possible by colliding with them, while avoiding getting stuck in blood clots, which kill the player. Collisions with small antibody objects provide power-ups: either a temporary reduction in speed or invincibility to blood clots. Building off the simple "endless runner" style, we achieved 3D motion for all our objects and implemented randomly generated bends in our blood vessels. Moreover, we utilize the post-processing Bloom effect to create exciting visuals and add audio to create an immersive gameplay experience that has relevance to the current global pandemic caused by the coronavirus (COVID-19).

Introduction

Goal

Just as the brainstorming phase of this final project began, COVID-19 dramatically affected the lives of everyone in the University and around the world. Given the magnitude of the current pandemic and how COVID-19 has taken so many lives and made many of us feel helpless, we wanted to create a game that gave people the chance to fight back against the virus and take control in a virtual sense.

After deciding our theme, we landed on the popular "endless runner" game genre for our style. In this kind of game, a player has limited control over a character that is continuously moving forward. The relatively simple controls of such a game would enable us to quickly implement an intuitive framework and then iterate on it to add challenging variations of motion and physics, as well as more realistic graphics and gameplay.

Combining this with our thematic inspiration and our knowledge of the immune system, we created an endless runner game that allows the player to help an immune system fighting against coronaviruses. Making the player feel like they are a white blood cell moving through a blood vessel and achieving accurate motion through curving tubes was a nontrivial task. We also used our knowledge from the course to implement the geometries, textures, materials, and interactivity of the gameplay scene. Our enhancements, such as antibody powerups and sound effects, helped create a fun game that players can enjoy in this pandemic.

Previous Work

There are many endless runner games, and the most famous one we have encountered is Temple Run.[1] We also took inspiration from the Collideoscope[2] project completed in Spring 2019 of COS 426. Both of these games provide a 3D endless running format, which we wanted our game to also have. Collideoscope's endless runner travels down a straight infinite cylinder, which is similar to our vision of traveling through a tubular blood vessel. However, to create a more realistic effect, we decided that the blood vessels in our game should be curved and winding, as real blood vessels are in the body. There are bends in Temple Run's infinite path, but these bends are sharp 90 degree angles, whereas our blood vessels would need to curve smoothly.

We were not able to find any other games that achieved the curving effect or graphics of a blood vessel and the immune system. Thus, we implemented an original method to allow for curvature of the blood vessels. We referenced previous assignments for basic rendering and event-handling but added many more features and complex geometries in our game. We also used and built off of the starter code provided by the COS 426 course staff, which helped us structure and organize our code when we began the project.

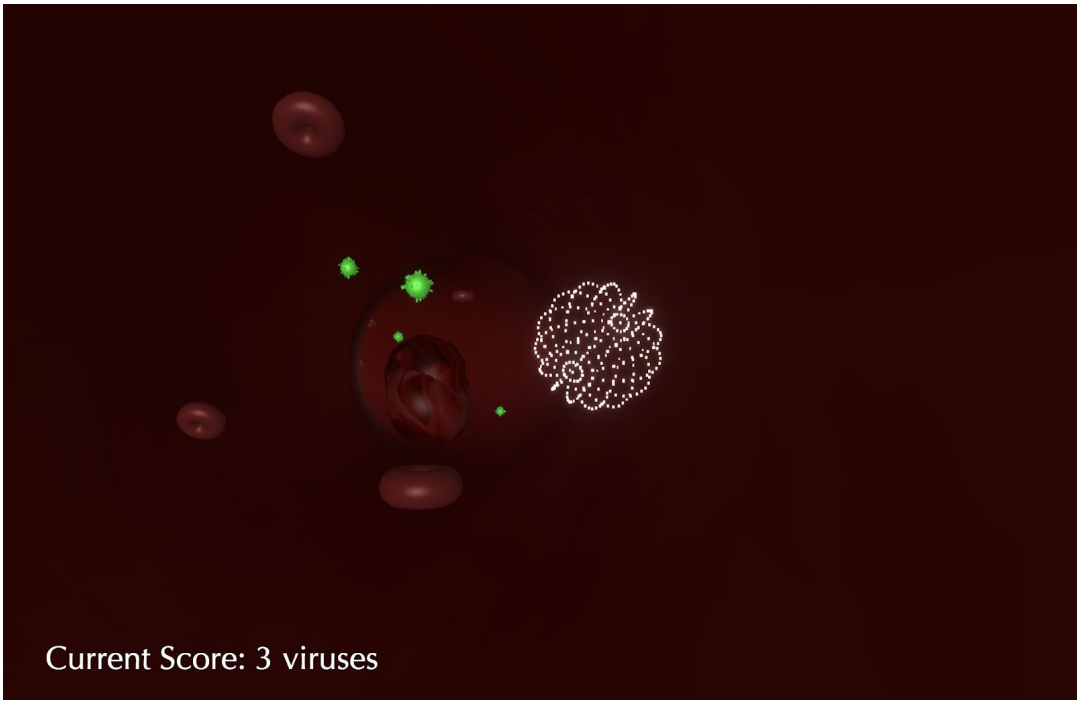
Approach

The approach we took for our game setup was to create a curving tube geometry that rotated as it moved past the player and the camera. Since we implemented our game from scratch, we began with a simplified setup that had only straight tubes to move through. We initially moved objects and the enclosing Cylinders from the ThreeJS library past the player and

[1] https://en.wikipedia.org/wiki/Temple_Run
[2] https://github.com/ewilden/collideoscope

camera to get a working MVP. Once this was completed, we created tubes that could bend at random intervals. Other game play objects were grouped with the tube geometry, and they rotated and moved with the enclosing tube. This created the illusion of the player moving forward through the tube and past objects in the tube, without losing calculation precision at the far boundaries of the scene. Our approach of having all the gameplay objects moving together past the player worked well for our game because the player tries to hit or avoid objects throughout the length of the tube (see image below). This was the simplest solution for achieving the complex geometry of a curving path.

However, this approach would not work as well for games in which multiple game play objects move at different speeds relative to the player. Due to the nature of how the enclosing tube must constantly rotate as it moves past the player, attempting to move other objects in the tube at different speeds causes them to move outside the walls of the tube at certain points. This kind of movement would require a different approach, but we decided it was not necessary for our game, since we were simulating movement through a blood vessel, in which objects typically move at similar speeds.



Methodology

The pieces that had to be implemented were:

- Basic scene setup (camera, lighting)
- Tube geometry and motion
- Object geometries, generation, and collision detection
- Player physics and controls
- Start/end menus and scores
- Gameplay (speed, difficulty ramp-up, powerups, audio)

Basic scene setup (camera, lighting)

We decided to leverage the ThreeJS library to build our game since it is an intuitive API with many capabilities that we have become familiar with throughout the course. Our basic scene consists of a player moving through a tube with the camera following close behind the player. Since objects and the enclosing tube move past the player and the speed is determined based on how far a player has made it through the game, we limited player actions to the xy-plane. The camera is set a certain distance behind the player and follows the player. With this, movement in the xy-plane would be easy to see for the player.

There were several possible implementations of the forward motion of the player through the tube. The most intuitive implementation would be to continuously move the player forward through the tube and in the world coordinate system. However, a limitation of this approach would be the loss of precision of the objects' positions as the player moves further from the scene's center. This would make the player's interactions with other gameplay objects appear unrealistic or inaccurate. To avoid this, we chose a different implementation which would move the enclosing tube around and past the player. The advantages of this method were that we could keep the player near the center of the scene, retaining precision while still creating realistic movement in which the player appears to move forward. However, the challenge of this was that the geometry of the movement of the tube past the player is much more complex than movement of the player forward through the tube, as described in the next section.

For lighting, we added AmbientLight and HemisphereLight to create a subtly lit scene that reflected a realistic blood vessel. We also added two PointLights, one behind the camera so that objects would be easier to see as they came closer to the player and one far in front of the camera so that objects in the distance could still be seen.

Tube geometry and motion

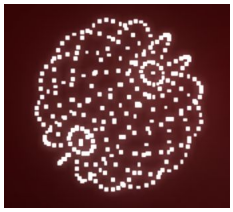
There were several possible implementations for the geometry of the tube. The most obvious geometry for a tube, and the one used by Collideoscope, is a cylinder. However, even combinations of multiple rotated cylinders would not give us the rounded bends we envisioned for our blood vessels. Therefore, we decided to implement the geometry of the tube using ThreeJS's TubeBufferGeometry. This enabled us to define the smooth curves that the axis of the blood vessel would follow. We also found an image that was visually representative of what the inside of a blood vessel would look like and used it to texture the tube to make it look more realistic.

Once we decided to use TubeBufferGeometry, we had the option of loading each segment of the tube into the scene separately or grouping them together into one object. Loading each segment separately with its own curve and mesh property would be cleaner in terms of the organization of the code, but proved to require more complex rotational and movement calculations when moving the tube past the player, as described below. Therefore, we created a larger Tube group with an array of tube segments, which enabled us to rotate the tube and all its segments as one seamless unit. Rotation is required to create the illusion of forward movement through the tube instead of just movement of the tube around the player.

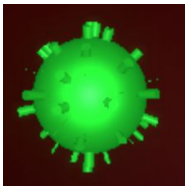
As noted in the previous section, there were also multiple possible implementations for the motion of the scene. Moving the player forward through the tube would be geometrically simple; the player's next position would simply be the next point in the curve of the tube geometry. However, to avoid loss of precision at the far edges of the scene, we decided to move

the enclosing tube past the player. To achieve this, we used the equation for the curve of the tube, which we defined when creating the tube geometry, to find the vector from the player's current point in the tube to the point in the tube the player would appear to be in at the next time step. We moved the tube in the opposite direction of this vector to place the player at the correct position in the tube. To ensure that the player and camera were always looking forward in the tube, and not, for example, at the walls of the tube, we also had to rotate the tube so that its tangent at the player's position in the tube was pointing in the same direction as the camera. The larger Tube object always stayed in the same position as the player, but the tube segments were continuously moving and being generated in the local coordinates of the Tube group. When a segment moved past the player, it was destroyed.

Object geometries, generation, and collision detection



The player object represents a white blood cell, which is the protagonist of our game. We initially created this using a sphere geometry and a basic MeshPhongMaterial. Since the camera follows the position of the player object, we made the material slightly transparent so that the player could see what was in front of the "white blood cell," However, when we applied the UnrealBloomPass post-processing, all opacities were set to 1. We ultimately used ThreeJS's PointsMaterial to create a spherical "white blood cell" that gives the player visibility of the entire scene in front of the camera and is even more visually interesting than the initial transparent material.



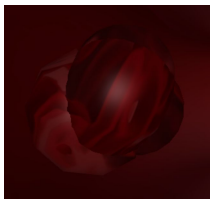
The virus object is a group of meshes that makes use of ThreeJS's sphere and cone geometries. Most graphical representations of viruses appear as spheres with bumps and spikes emerging from their surfaces. To achieve this, we centered a sphere and many cone geometries at the same point and rotated the cones so that they stuck out from the sphere's surface at random points. We found that simple randomization of the cones' rotation angles did not appear uniform, so we used the uniform random sampling method described in precept 7[3] of this course to determine the points at which the cones should stick out from the sphere body of the virus. We also randomize the radii and sizes of the viruses. Collision with a virus is detected by calculating the distance between the player and virus positions; they have collided if this distance is less than the sum of the player's and virus's radii. In this case, the virus is removed from the scene and the score of the player is increased and displayed in the bottom left corner of the screen.

Each antibody is represented as a group of 5 cylinders rotated and shifted relative to each other in a Y-shape. This is anatomically accurate and based on the organization of polypeptide chains that make up antibodies.[4] We created two types of antibodies, one that is blue and associated with the invincibility powerup, and one that is orange and associated with the speed powerup. Both powerups are described further in the gameplay section. Collision is detected by checking if the distance between sphere and antibody positions is less than the sum of the sphere's radius and the antibody's width or height.

[3] https://www.cs.princeton.edu/courses/archive/spring20/cos426/precepts/Precept-7.pdf
[4] https://en.wikipedia.org/wiki/Antibody#Structure

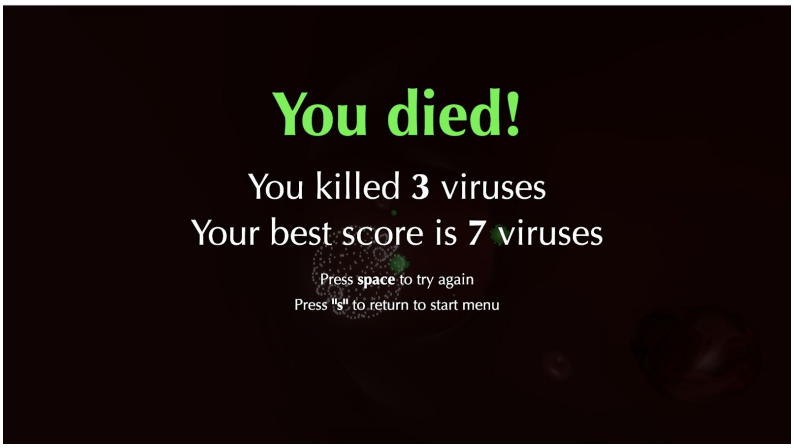
Using ThreeJS's TorusBufferGeometry, we created red blood cells that have the characteristic "donut" shape in most graphical representations of these kinds of cells. We included the red blood cells to enhance the feeling of being inside a blood vessel. Since they have no gameplay functionality, instead of waiting for player collision with red blood cells, we detected when the distance between the positions of the player and red blood cell is less than the sum of their radii plus 0.1; when this is the case, we move the red blood cell away from the player as if it were floating away from the player in a fluid environment.

We originally planned to represent blood clots with a group of red blood cell meshes. However, we found that the large number of these individual small meshes required to create a convincing "clot" slowed down the frame rate of the game. Therefore, we decided to use ThreeJS's TorusKnotBufferGeometry with an image texture to represent blood clots. By setting the tube radius of the torus knot to be twice its geometry radius, we created a knot that was visually complex but also closely approximated its bounding sphere. Therefore, we could use the radius of the clot's bounding sphere to detect collision with the player in the same way for which we implemented virus collision detection. If a collision with a clot is detected, we end the game.



Player physics and controls

We implemented our own physics for the player and gameplay objects. Movement of the player is determined by arrow key presses. When a player holds an arrow key down, force is accumulated in a ThreeJS Vector3, which is used to update the player's velocity and position. We also multiply the velocity by a drag factor to slow down player movement as we want to simulate that we are traveling through a blood vessel filled with fluid. The force is added in each frame until the arrow key is released. This approach allows for smooth and intuitive movement, as the player moves as long as the arrow key is pressed. We ensure that the player stays within the bounds of the enclosing tube by limiting movement past the boundaries of the tube.



Start/end menus and scores

To easily explain our game, we use a 2D overlay to show a start menu (shown on the first page) when a user first loads our game to explain our controls. During gameplay, we display the

number of viruses a user has killed so far. We keep track of this number as well as the highest score the player has attained in a browser session. Once a player dies in the game, we display an end menu, also using a 2D overlay, that shows the user's most recent score and high score. The end menu (shown above) gives the player the option to start the game again or return to the start menu. The overlays were implemented using HTML and CSS, and they are conditionally shown based on what state the player is in the game.

Gameplay (speed, difficulty ramp-up, powerups, audio)

The game starts off at a slow speed, to allow the player to get used to controls and gameplay environment. To make the game more interesting, we increase the speed by a small fraction in each frame as the player makes it further along the blood vessels. The speed increases linearly until a maximum speed, which we set to prevent the movement from getting too disorienting. This setup allows players to be more challenged as the game goes along but so much that the game becomes impossible; our speed antibodies allow the player's speed to be slowed down again.

Without invincibility *With invincibility*

We implemented powerups using the antibodies we described above. Upon collision with an antibody, the player experiences the effect of that antibody. Blue antibodies provide "invincibility" to the player, and the player will not die from colliding with blood clots for awhile. We indicate invincibility by adding a stronger Bloom effect when this property is present (see image above). We implement this powerup by keeping track of the distance a player has traveled and using a boolean to indicate invincibility. Once a certain distance has been covered by a player, invincibility is turned off. Orange antibodies affect the speed of a player. Upon collision, they decrease the speed of a player by a set amount to make gameplay easier. We limit the minimum speed of the player to ensure that we don't stop moving in the blood vessel.

We also added audio to our game to make it more interesting for players. The audio can be turned on/off by pressing the "m" key. When the start and end menus are displayed, the intro song for the "Osmosis Jones"[5] is played. During gameplay, there are various sound effects upon collision with different objects. Colliding with a virus or either of the two antibodies will cause a different sound to be played.

[5] https://www.youtube.com/watch?v=wsdSh_qpiNA

Results

We measured our success based on the scope of our initial proposal. We set out to build a game that would provide intuitive gameplay with realistic physics and interesting graphics. Visually, our game is compelling and interesting. We implemented accurate and realistic blood vessels to create an immersive environment for gameplay. The many different kinds of objects that players can interact with make the game fun and challenging. We were able to implement all of the stretch goals we initially made and even added features, like powerups, that were beyond the scope of our proposal. Our end product is a polished game that can provide entertainment for others.

In addition to testing our game ourselves, we solicited feedback from other users to iterate on our game and make it more appealing. We decided to allow simple movements using arrow keys to let players quickly get the hang of the game, and we found that test users were able to figure out how the controls worked quickly. Once the game ended, they continued to play additional rounds because they were intrigued by the different features in the game. Our results indicate that we were successful in creating a game that was fun and empowering in the wake of the current global pandemic.

Discussion

We took an entirely original approach to implementing our game and created all the geometries and physics of the game from scratch. In creating the realistic geometry of the winding blood vessels, we believe that we developed the best possible approach that we could think of. There is no source code for a smooth bending curve moving past a player that we could find, so we do not know of any other approaches that ours could be compared to.

However, there is still follow-up work that could be done in the future. Our current game allows only for movement of all other gameplay objects past the player at the same speed. To the player, this looks as if the player is moving forward in the winding tube past stationary objects, other than the red cells that move away from the player. Future work might strive to achieve movement of gameplay objects past the player at different speeds while keeping the objects within the winding tube.

Unlike previous assignments, in which most of the code was laid out for us and we filled out functions almost entirely relating to the physics of the geometries, this final project taught us about the whole rendering and game production process. We learned how to load pages for start and end menus, manipulate logic in a render loop to simulate motion in a scene, and create and load geometries into these scenes. By creating an interactive game, we not only developed our ability to reason about stationary three-dimensional geometry, but also about graphics in motion. Finally, we learned about the costs and challenges associated with creating such realistic and complex scenes, such as loss of frame rate and precision.

Conclusion

CoronaTime realized our goal of creating a relevant game that realistically simulates infinite motion through a bending blood vessel and provides exciting gameplay. The next steps would be to implement more powerups and motion of objects at different rates relative to the player. We were able to polish our game into a state that we are proud to share and hope that it will lift our players' spirits during these trying times.