Snake Game with a Twist

Josh Glazer and Ryan Zeng
{Josh.Glazer001@umb.edu},{Ryan.Zeng001@umb.edu}
University of Massachusetts Boston

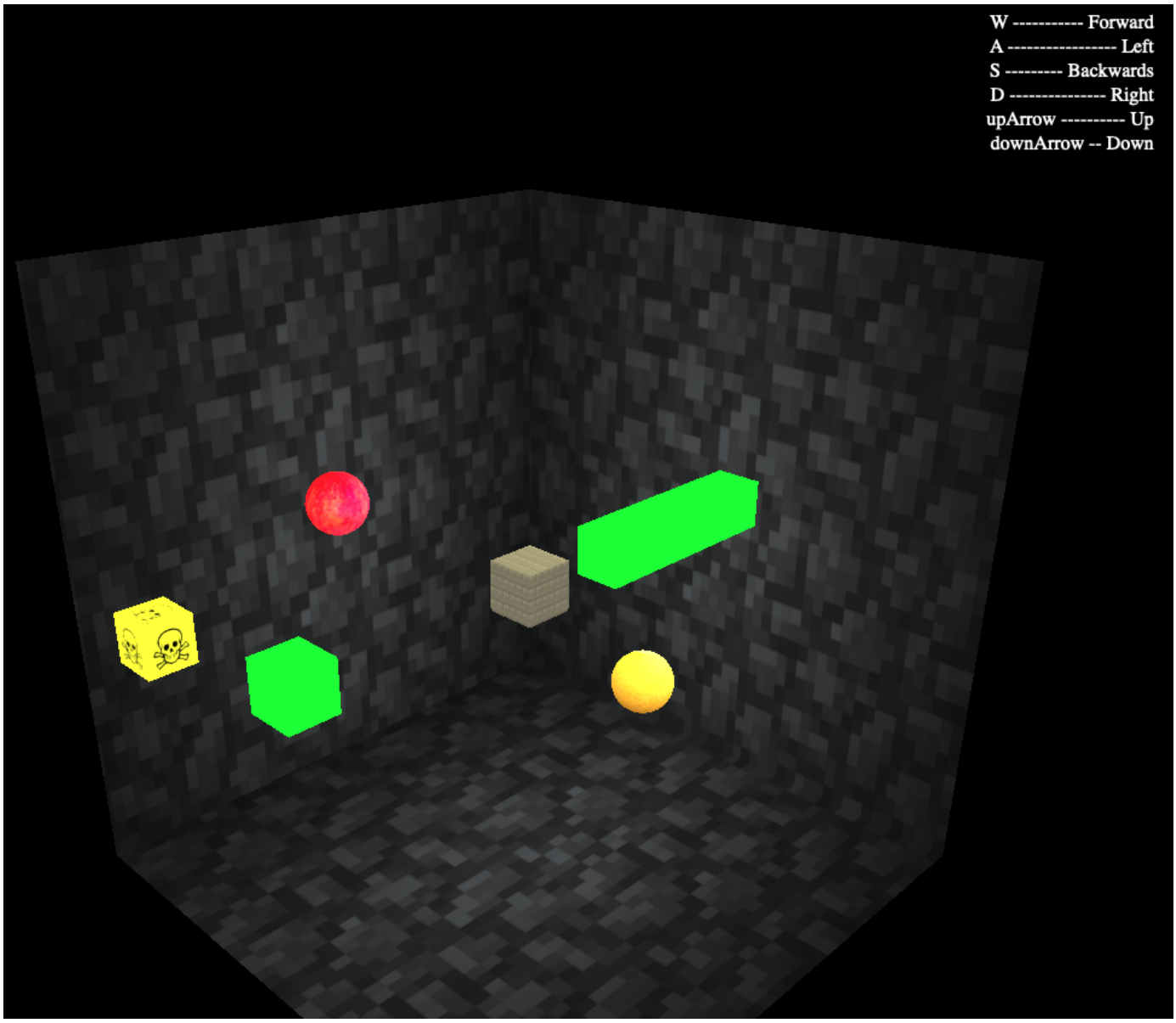

Figure 1: Above is a look at our 3D Snake Game. We have the green snake, the red apple, poison apple and the center cube to divert the snake upon collision.

ABSTRACT

Our project is a 3D WebGl representation of the 2D pixelated snake game in which the objective is to consume a target object to elongate the snake by 1 unit. Our goal was to build a fun game that represents the 3D capabilities of THREE.JS. While there are many retro 2D games out there that have been long forgotten, this project shows how old games can be revamped with new technologies.

KEYWORDS

WebGL, Visualization

ACM Reference Format:
Josh Glazer and Ryan Zeng. 2022. Snake Game with a Twist. In *CS460: Computer Graphics at UMass Boston, Fall 2022.* Boston, MA, USA, 2 pages. https://CS460.org

1 INTRODUCTION

The importance of this project is to demonstrate creative liberties of the team while exercising knowledge learnt from the class lectures hosted by Daniel Haehn on WebGL functionality. The project is divided into two halves. First, the one-to-one reiteration of the game is implemented with the added mechanics of an operational third person camera and a 3-dimensional representation of the snake constructed using an array of cubes. Additionally, the project was constructed and decided upon due to the team's ambition to create a fun game, and so it was decided upon to rehash an existing game permitting the team to prioritize creativity on representing already existing rules. What contribution the project serves, is that it's a presentation of how WebGl and THREE JS alone are being used to implement 3D graphics based web-games , without the use of the Unity Engine.

2 RELATED WORK

We began with a lot of youtube videos to inspire us with how to logically turn a 2d game into a 3d game. While there were some examples of a simple 3d game in three js they all had the common theme of using object oriented programming to build the scene objects. [7] We took inspiration from youtube videos which touched on the subject of building arrays of objects and how they would be placed inside a 3d space. [8]

Three.js has a Vector3 class that represents a 3D vector which is an ordered triplet of numbers (x, y and z). Three common things the vector can represent: a point in a 3D space; a direction and length in a 3D space, with the length being the Euclidean distance from (0, 0, 0) and (x, y, z); and any arbitrary ordered triplet of numbers. [1]

The YouTube tutorial covers the basic fundamentals about third person cameras in video games and how to implement them in Three.js. [4]

The following YouTube video is a tutorial on how to implement a third person perspective camera in Three.js from scratch. How a third person camera works is that it follows a playable character while viewing the environment in the same direction as the playable character, but ahead of the player. Also, there is some dampening with the camera as it follows the character, or bobbing effect to express smoother animation movement. [4]

3 METHOD

The process we took to complete this project included our research into three.js and how its vast amount of objects can be used to create a simple 3D game. Objects like THREE.Clock were found to be great for controlling the speed of the game since you could control how frequent the animations would occur. Javascript methods such as *distanceTo* were found to be excellent for determining collisions between objects. Another javascript method we used was the, *onKeyDown* event listener, which was used to control the snake using the keyboard controls. When we ran into problems we would try to debug using the console or research more about an issue to learn what others did when running into similar problems. This was helpful because the console would give more information about an object such as its attributes like *mesh.position, mesh.visible* and *clock.elapsedTime.*

3.1 Implementation

We started with the boilerplate code used for creating a scene, adding lighting, cameras and the animation loop. Then we dived into adding the cubes and spheres into our scene which would represent our snake and apple. While we had all the objects added to the scene we were then faced with the challenge of animating those objects so that the snake appeared to be moving. Once we had the snake moving we had to bound our snake inside the board game or else it would run off into the abyss. Next we had to let the user control the direction of the snake using the keyboard controls. And finally we had to implement a method of detecting collisions between not only the snake and apple but also the snake colliding with itself as well.

3.2 Milestones

3.2.1 Milestone 1: Representing the snake game in a three dimensional environment.

Fortunately, WebGl renders an HTML canvas with a coordinate system composed of x, y, and z coordinates. Added to the scene was a snake array of cubes, a red apple which extends the snake by 1 unit, as well as a golden apple which doubles the snake in length, and lastly, a poison apple which reduces the snake length by half. These creative liberties are meant to increase the game complexity as well as strengthen our knowledge using THREE.js objects. Looking back at completing assignment 2, creating pixel art composed of XTK cubes, all cubes and spheres were coordinated differently for x and y positions, but all on the same z coordinate. The result was a flat 2D pixel image. For this project we will need to modify not just the x and y coordinates but also the z coordinate for the snake to appear in 3D. In order to visualize the snake in 3D we included 3 sides to be textured walls. This allowed for the user to efficiently see the snake move in all three dimensions.

```
// ---create snake array---
for ( var i = 0; i < 8; i ++ ) {
    var snakeMaterial = new THREE.MeshPhongMaterial({ color: 0x00ff00 });
    snake.push( new createCube( new THREE.Vector3(
                                /* X */     0.5 + pad / 2,
                                /* Y */     (i + i * pad) - halfBoard + 0.5,
                                /* Z */     0.5 + pad / 2),
                            /* material */  snakeMaterial,
                              /* scene */   scene ));
}

// ---create apple object---
appleChoice = true;
var texture = new THREE.TextureLoader().load('apple.png');
var appleMaterial = new THREE.MeshStandardMaterial( {
      map: texture
});
apple = new createCube( createApple(), appleMaterial, scene );

// ---golden apple doubles snake length---
var texture = new THREE.TextureLoader().load('gold-apple.png');
var goldenAppleMaterial = new THREE.MeshStandardMaterial( {
      map: texture
});
goldenApple = new createCube( createApple(), goldenAppleMaterial, scene );
appleChoice = false;

// ---poison white apple removes half the snake length---
var texture = new THREE.TextureLoader().load('danger.png');
var poisonAppleMaterial = new THREE.MeshStandardMaterial( {
      map: texture
});
poisonApple = new createCube( createApple(), poisonAppleMaterial, scene );

// ---create center cube to redirect snake in different directions---
var texture = new THREE.TextureLoader().load('bg.jpg');
var material = new THREE.MeshStandardMaterial( {
      map: texture,
      color: 0xb8b09b
});
cube = new createCube( new THREE.Vector3(0.58, 0.58, 0.58), material, scene );
```

3.2.2 Milestone 2: Animating the snake to appear like it's moving.

While adding the snake to the scene wasn't too difficult, animating the snake will be a lot harder because not only are we animating one object, we are animating an array of objects. Luckily we don't have to animate every object in the snake array to make it appear moving. For this we just have to repurpose the tail of the snake as the head and it will appear as if the snake is moving. This allows for the snake to move one unit at a time. While the snake is now moving, there are no boundaries which contain the snake to just the board size, so we must implement a boundary which keeps the snake on the board at all times.

```
// ---redirects the snake from one wall to the opposite wall---
if ( head.mesh.position.x < -halfBoard) {
  head.mesh.position.x = halfBoard - 0.5;
} else if ( head.mesh.position.x > halfBoard) {
  head.mesh.position.x = -halfBoard + 0.5;
} else if ( head.mesh.position.y < -halfBoard) {
  head.mesh.position.y = halfBoard - 0.5;
} else if ( head.mesh.position.y > halfBoard) {
  head.mesh.position.y = -halfBoard + 0.5;
} else if ( head.mesh.position.z < -halfBoard) {
  head.mesh.position.z = halfBoard - 0.5;
} else if ( head.mesh.position.z > halfBoard) {
  head.mesh.position.z = -halfBoard + 0.5;
}
```

This objective required us to not only contain the snake but also redirect the snake when it reaches the edge of the board. For this we redirected the position of the snake to the opposite side of the board to appear as if the snake was running through the wall from one side to the other.

To control the animation speed we included a clock from three.js which we use to determine when to animate the snake instead of animating it every loop which would be too fast for anyone to play the game.

```
  // ---remove the center cube so more natural feel---
  if ( clock.elapsedTime > 5 ) {

      cube.setPosition( disappear );
  }
};
```

3.2.3 Milestone 3: Changing the direction of the snake

```
// ---event listener to collect user inputs through the keyboard---
document.addEventListener("keydown", function(e){
        if ( e.key == "ArrowUp" ){
          keys.push(new THREE.Vector3(0,1,0));
        } else if ( e.key == "ArrowDown" ){
          keys.push(new THREE.Vector3(0,-1,0));
        } else if ( e.key == "w" ) {
          keys.push(new THREE.Vector3(0,0,-1));
        } else if ( e.key == "
                                any
          keys.push(new THREE.Vector3(-1,0,0));
        } else if ( e.key == "s" ) {
          keys.push(new THREE.Vector3(0,0,1));
        } else if ( e.key == "d" ) {
          keys.push(new THREE.Vector3(1,0,0));
        }
});
```

Now the snake is moving forward but we have yet to make the snake move in different directions. For this we must devise a system for the user to use keyboard controls to navigate the snake through 3d space. Here is where we use the *onKeyDown* event listener to include user input to change direction of the snake. In total we need 6 keystrokes which are w,a,s,d for movement on a 2d plane and the keystroke upArrow and downArrow are used to translate up and down planes. To determine the direction of each keystroke we used *Vector3* which is a three.js object used to represent a point in 3d space.

```
function animate() {

  requestAnimationFrame( animate );

  // ---determines if snake has collided or not---
  if ( collided == false ) {

    // ---increment speed counter---
    movement_speed += clock.getDelta();

    // ---speed check determines snake movement---
    if ( movement_speed > delta ) {

      // ---remove and declare the first element in the array as the tail---
      var tail = snake.shift();

      // ---Declare head as the last element in the array---
      var head = snake[snake.length - 1];

      // ---Determine direction of the snake using keys array---
      direction = keys.length > 0 ? keys.pop(0) : direction;


      // ---Determine snake movement using head position, direction and padding---
      var movement = new THREE.Vector3(
                          /* X */    head.mesh.position.x + direction.x + Math.sign(direction.x) * pad,
                          /* Y */    head.mesh.position.y + direction.y + Math.sign(direction.y) * pad,
                          /* Z */    head.mesh.position.z + direction.z + Math.sign(direction.z) * pad
      )
```

For example, if the snake were moving to the right then our X =1, y=0 and z=0 to move right but if we want to move left it would look like x= -1, y=0 and z=0. When we reposition the snake tail as the new head we take in account the direction to determine where in 3d space to position the new head.

```
// ---set the new x, y and z position of the tail of the snake---
tail.setPosition( movement );

// ---push the new tail position on to the snake array---
snake.push( tail );

// ---replace the old head with the new one---
head = tail;

// ---detect collision between head of snake and apple---
// ---upon collision snake is lengthened by 1 unit
if ( head.mesh.position.distanceTo( apple.mesh.position ) < 2 ) {

  apple.setPosition( createApple() );

  snake.unshift( new createCube( new THREE.Vector3(
                                 snake[0].mesh.position.x,
                                 snake[0].mesh.position.y,
                                 snake[0].mesh.position.z), new THREE.MeshToonMaterial( { color: 0x00ff00 } ), scene));
};
```

3.2.4 Milestone 4: Detecting Collisions inside the game

```
// ---detect collision between head of snake and golden apple---
// ---upon collison snake is doubled in length
if ( head.mesh.position.distanceTo( goldenApple.mesh.position ) < 2 ) {

  goldenApple.setPosition( createApple() );

  for ( var i = 0; i < snake.length / 2; i++ ) {

    snake.unshift( new createCube( new THREE.Vector3(
                                   snake[0].mesh.position.x,
                                   snake[0].mesh.position.y,
                                   snake[0].mesh.position.z), new THREE.MeshToonMaterial( { color: 0x00ff00 } ), scene));
  }
};
```

While the snake was now moving and it appeared to be working, our snake and apple were yet to detect any collisions thus far. This means that the snake will travel right through the apple without even noticing. The objective of our project was to re-enact the snake game but in 3D so we must have a way to detect collisions in order to spawn a new apple as well as lengthen the snake by 1 unit. While this problem required some debugging, the console really helped with this problem because if I typed in " snake ", the snake array object, into the console it would return to me a bunch of attributes associated with this object. This was helpful because I was able to use these attributes to detect collisions using *mesh.position* to compare the head of the snake to the apple as well as the head of the snake to the body of the snake because the snake is not allowed to collide with itself.

```
// ---detect collision between head of snake and poison apple---
// ---upon collison snake is reduced by half its length---
if ( head.mesh.position.distanceTo( poisonApple.mesh.position ) < 2 ) {

  poisonApple.setPosition( createApple() );

  for ( var i = 0; i < snake.length / 2; i++ ) {

    snake.shift().setPosition( disappear );
  }
};

// ---detect for collisions within the snake array itself---
for (var i = snake.length - 2; i > -1; i--) {
  if ( head.mesh.position.distanceTo(snake[i].mesh.position) < 1) {

    collided = true;
    break;
  }
};
```

3.2.5 Milestone 5: Implementing the Third Person Camera

```
// ---change position of camera based on direction of snake to give---
// --- the game a more natural feel---
if ( direction.y == 0 ) {
  camera.position.set(10, 30, 10);
} else if ( Math.abs(direction.y) == 1 ) {
  camera.position.set( 20, 10, 30)
}
```

The team attempted to utilize a third person camera, since the snake game is being readapted into a three dimensional space. Unfortunately, this milestone became almost more than that could be chewed, because there are so many considerations of how the camera must interact in accordance to game elements and the limitations of having no game engine, or any real rotation of the cubes composing the snake. Thus we ended up using a simpler camera implementation which involved the use of *camera.position.set* to change the view of the camera giving the user a more natural feeling to the game since its in 3D now.

The third person camera acts like an independent driver that follows a constantly updating path which is the snake's body. This camera is always relative to the snake's tail by some adapting offset, thus allowing the camera to view the entirety of the snake and the environment ahead of it. There are two central ideas to consider when implementing a third person camera: first is the camera's position and its look at position [4]. Both these values of the camera must change relative to the snake's cube components' disposition every frame.

Since the third person camera is similar to a driver, every turn will represent a new path for the driver to follow, and the number of cubes on a path is the number of frames the driver takes to finish the path, and be able to turn to the next path. The driver is always looking ahead to chase the lead, so the lookahead should always be some spaces ahead of the snake - simple. The tricky parts involve positioning the camera.

Like a driver, the camera accelerates when approaching the end of the path, so the camera traverses more distance upon each subsequent frame, this is done by simply subtracting the new destination with the current destination of the camera and dividing that by the number of remaining cubes, or frames, on the path. The remainder must also be accounted for too. The result is an overall distance the camera moves upon the current frame. The result gets greater upon the next frame.

How and what exactly is the final destination for the current path? That is calculated by recording when a change in direction is detected, and the cube position where it happened which neighbors the head cube repositioning itself by this new direction. In short, the final destination is relative to the cube at snake[snake.length -2]. Of course that is not the final destination, but some offset relative to that position, so how is this offset determined?

The third person camera must be maintained so that it appears to be relative to the top left corner of a cube with the face closest to the viewer's screen. Imagine if some generic cube, and the trajectory of the camera relative to it, is another cube encapsulating it (See figure 3.2.1 below).

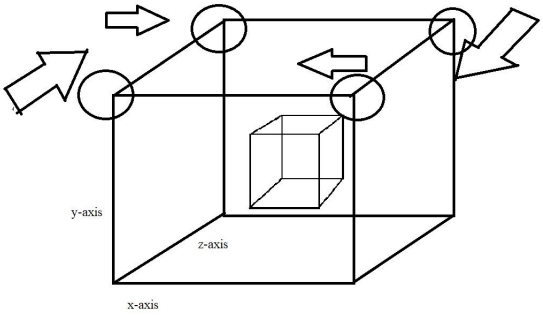

Figure 3.2.1: *The following figure above is a 3D perspective of the camera's trajectory and positioning in regards to a 3D cube. This is the 3D rendition of the previous figure. The circles represent the camera's position, the arrows the direction the snake shifts, and the encapsulated cube some generic cube, while the encapsulating cube is the camera's trajectory path.*

The camera in figure 3.2.1, even when the snake is moving in either of the x and z directions, always maintains its position, at least according to the player's eyes, but not so much in the actual coordinate system. This is also witnessed in Figure 3.2.2.

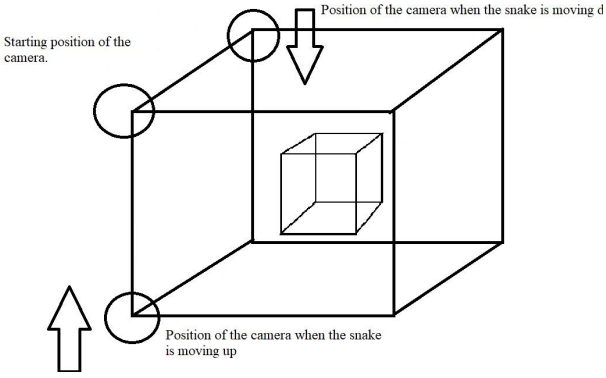

Figure 3.3.2 *The above image shows the camera's positioning in respect to a change in direction of the y-column, or up and down*

Figure 3.3.3 shows a top down view of the camera's positioning upon x and z direction inputs.

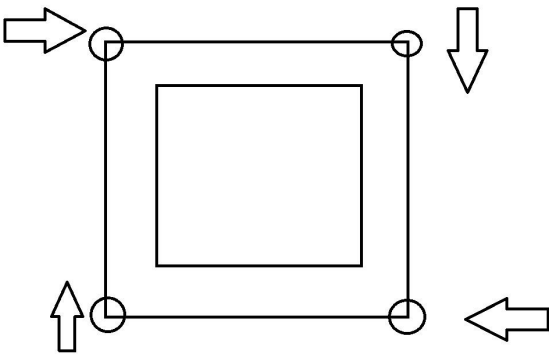

Figure 3.3.3 *The diagram shows a top down view how the camera must be positioned from a top down angle, considering the x and z axis only*

In summary, there are six positions that need to be considered for a camera to be positioned, in respect of a single cube; with each position their own (x, y, z) coordinate vector. Additionally, there are a varying finite number of frames to transition the camera's position, based on the number of cubes that precede a turn. There are six inputs each dictating where the camera will be positioned. Lastly, the camera will always look ahead at some number of cubes in front of the snake's head.

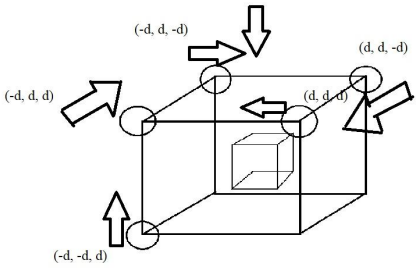

Figure 3.3.4 *The figure above shows the different vector offsets from a cube's position, in respect to the direction input by the player. Here is a list (d is the edge length of a cube and the vectors are the form of (x, y, z) for a coordinate system):*

User Input for Movement	Offset vector from a cube's position
w (forward)	(-d, d, d)
a (left)	(d, d, d)
s (backwards)	(d, d, -d)
d (right)	(-d, d, -d)
Arrow Up	(-d, -d, d)
Arrow Down	(-d, d, -d)

Finally, once the generic offset is acquired based on direction, and the position where a new direction is decided, then these values are summed.

Now the camera can drive along paths and make turns to new paths which are queued whenever new directions are inputted. Of course if there are no longer paths being queued, then the driver is just cruising on a straight path. But when the path is mirrored from an opposing wall after colliding with another, then there are even more complications. What happens when a path goes through a wall and makes a turn after collision is detected and half the

snake is adjacent to the opposing wall?

For that the path will be broken into two pieces to calculate the trajectory of the camera to a final destination, the camera will still rotate and increment in a single step for each frame, this time from its current position to the collision point, and the emergence point of the opposing wall to the final destination. The calculated rotation of the camera by some generic cube's center once collision point is encountered, is then carried to the emergence point where it is applied and incremented appropriately by the step at that frame of reallocation.

Lastly, what would happen when the snake elongates upon consuming apples. The rule to maintain is that the camera must always view the snake and the environment ahead, therefore, the camera shall traverse backwards by any number of frames relative to the number of cubes added to the snake's end, and this takes first priority.

That is the last of the third person camera's implementation that the team organized for days, and the result is a camera that perfectly follows the snake's tail while looking ahead of it. The camera is capable of accelerating and decelerating depending on the size of its current path, and this adds a lot of variety to the game, as the snake appears to move faster or slower even when its speed is constant. This highlights the importance of a camera and how it dynamically changes the feel of a scene.

Additionally, the original overview camera is kept, so that the player can view where the target apple is, since the lookahead of the third person camera is restricted to look ahead of the snake. Consequently, the thirdPersonCamera is limited in view, and finding any targets may be left to chance. Relying solely on the overview camera is also a burden, as it's hard to view exactly where the snake or apple is in the 3d plane. In short, both cameras are utilized, with the overview camera at the top right corner.

3.2.6 Implement a Game Over Sequence

```
// ---remove the snake starting from tail in descending order---
if (count % 5 == 0 ) {
  snake.shift().setPosition( disappear );
}

// ---remove all apple objects from scene---
apple.setPosition( disappear );
goldenApple.setPosition( disappear );
poisonApple.setPosition( disappear );

};
```

Thus far, the snake and scene objects remain in the scene even after a collision is detected. Our idea is to create an animation which is triggered by the detection of collision. For this we created a boolean variable called, *collided*, which we used to determine whether or not the snake had collided with itself. When this variable becomes true, our game over sequence is triggered inside the animation loop. After the snake has collided, the sequence begins with all apple objects disappearing from the scene as well as the snake descending into itself tail to head. WebGl is not capable of producing sound, but other parts of HTML are. Utilize the HTML 5 audio api. An image will be loaded that says game over when collision of the snake with itself is detected. This image will be loaded on the scene for the overview camera to see.

3.1 Challenges

Describe the challenges you faced.

- Challenge 1: Rendering the snake array to the scene with the correct position and spacing between objects.

```
snake.push( new createCube( new THREE.Vector3(
                                /* X */     0.5 + pad / 2,
                                /* Y */     (i + i * pad) - halfBoard + 0.5,
                                /* Z */     0.5 + pad / 2),
```

While rendering the snake object into the scene we quickly ran into the issue of where to place the snake array in relation to each cube. If they were too close then we would have overlapping cubes or too far and the snake would lose its form. Thus we must position the snake objects so there is some padding between them. With this padding, we can allow the snake cube objects to move in any direction and remain the same form.

- Challenge 2: Animating the snake array for movement across the board game.

```
// ---Determine snake movement using head position, direction and padding---
var movement = new THREE.Vector3(
                /* X */    head.mesh.position.x + direction.x + Math.sign(direction.x) * pad,
                /* Y */    head.mesh.position.y + direction.y + Math.sign(direction.y) * pad,
                /* Z */    head.mesh.position.z + direction.z + Math.sign(direction.z) * pad
)
```

While the snake array is now positioned correctly on the board, animating the snake to allow for movement is our next challenge. The task here is determining where to place the next cube at the new head of the snake after we remove the tail object of the snake. In order to solve this problem we need to

determine the direction which is inputted by the user. Once we have the desired direction along with the padding, we can determine the position of the new head of the snake. To appear as if the snake is moving we must remove the tail of the snake and reposition it onto the new head of the snake. Now with the desired position of the new head, we must push this cube back on the snake array.

- Challenge 3: Detecting Collisions between snake head and apple as well as snake head and snake body

```
if ( head.mesh.position.distanceTo( apple.mesh.position ) < 2 ) {
```

To detect collisions between the snake head and the apple we utilized the javascript function *distanceTo,* which we used in class for homework in order to detect robots walking into each other. Within this function we used the attribute *mesh.position* to determine the position of each the head of the snake, the apple as well as the body of the snake. The *distanceTo* function will return a numerical value representing the distance between each mesh. We used this value to determine if a collision occurred or not. It is possible to determine whether an object is close to a collision as well. This was helpful in the survival algorithm within the apple object because we want to make the apple sense if the snake head is close which will trigger the apple to avoid the snake in an attempt to make the game harder.

- Challenge 4: Implementing a camera which will follow the snake in 3d space

There were major issues when implementing a third person perspective camera that follows the snake. There are two instances to keep track of when following the snake, the camera's position, and the position the camera looks at. These positions are to update every frame according to some component of the snake. This was the first issue on deciding what part of the snake the camera should follow. At first, the team considered the snake simply following the head, but several problems arose from this decision.

Recall that the snake actually doesn't rotate when changing direction, rather the snake's components are repositioned. Also, since all faces of the cube are the same, rotation is easily imitated, although very snappy, as a cube repositions itself in a single frame. It is good practice for the camera to follow this rotation, otherwise, the camera will be at a constant vector (x, y, z). If the camera were set to look ahead some constant distance from the snake's head, then the result would be that the camera will sometimes appear at different angles from the user's perspective as the snake makes different turns.

Hence, for a more naturally feeling camera, the camera must change position every frame to complete some rotation. But if the camera's position were dependent on just a single cube, like the head, then the rotations would be extremely rigid and nauseating.

After a number of hours brainstorming, the team came to the realization that like the featured tutorial by SimonDev[4], the camera should rotate and change positions in reference to the entire snake. The team then offset the camera with some vector (x, y, z) from the tail. Unfortunately, this vector cannot be constant, as again the camera will be at different angles in reference to the snake when there are turns. This resulted in more hours of brainstorming about how the camera should be repositioned.

The resolution is to have multiple generic offsets to add to the tail's position, and each generic offset is chosen based on the direction of the snake. Then more complications arose, including where should the camera translate to, and how many frames must the camera take to change position? Recounting that animation occurs for every frame, and the framerate is corresponding to the repositioning of the cubes in the snake, the cubes are the answer. There are a number of cubes preceding a change in direction, these cubes are to be accounted for and that is synonymous with the number of frames occurring prior to a turn. So if a turn is detected, then the camera will begin changing positions and rotate to the cube prior to the snake's repositioned head, which the team evaluated to be the turning point, within x-amount of frames equivalent to the number of cubes on this straight path.

Realizing that rotations are being accounted for every new directed path, the team decided to break the camera's trajectory into intervals. We then concluded what should be accounted for in the interval: first a generic offset decided by direction; the number of cubes on the path; and a calculation of the new position the camera should arrive at. These values are what will allow us to update the camera to a new position based on the number of cubes, or frames, by incremental steps. This was easy to figure out in account of SimonDev's featured basics on camera fundamentals [4].

Unfortunately problems arose when considering what if the snake doesn't change direction. From that, the team decided upon queuing the intervals whenever there's a change in direction and popping these intervals once finished. If the queue became empty, then the team reasoned after debate that the snake is straight. This signifies that upon the next turn and even at the start of the game, when the next direction is inputted, the camera will have a snake.length amount of frames to change positions and rotate.

When there's no change in direction, simply create an interval of one frame that changes the camera's position to the next cube unit position in the latest recorded direction.

But even more problems arose when considering that the snake appears directly from a wall opposing a wall it collided with. This took considerable time to ponder as to how the camera should react to this situation. How should the camera follow the snake under these conditions? How can this transition appear as smooth as possible? Finally, the team concluded to edit how we use intervals in which it should account for wall interferences. Easier said than done, as this implementation required much thought.

When the snake's head collides with a wall, it is repositioned

to the opposing wall in a mirrored position. These two positions are integral as they're where the tail will eventually traverse. The two positions are then recorded for that reason, and the trajectory of the camera is concluded to be broken up into two paths under these conditions. The camera's current position to the cube position prior collision, and the cube position right after repositioning. Of course the camera is not a cube, but is some offset away, so the generic offset depending on the next direction is added to the two new account values.

There are no complete rotations within the first broken path from the camera's starting position to the position near the collision, but this incomplete rotation must be added to upon the next frame, once the camera jumps to the other end of the position after the wall. This was tricky to evaluate, but shortly resolved after some arithmetic calculations.

What if there are no turns through the walls, but the snake moving along a straight path? This took more considerable time to think about, until realizing that this is already accounted for as there be an instantaneous, but necessary, jump from place to place under a single frame.

There were a lot more technical difficulties and lots of planning and thought put into implementing this third person camera. Only the surface has been scratched under this list of challenges.

4 RESULTS

We started this class not knowing a lot about javascript and even less about three.js. In this semester we have gained a lot of knowledge not only from class but from researching for the homeworks as well as this project.

Not all our intended milestones were met when implementing the game, for example, an algorithm that focuses on the survival of an apple from the snake. There are apple varieties, including a golden apple that doubles the snake's current length, and a poison apple that kills the snake. Speaking of death, an animation of the snake collapsing into itself from tail to head has been implemented as well. Unfortunately, we weren't able to implement sound that plays automatically upon the death, nor is there a sufficient game over screen that follows. Further, there is no score board that's been implemented yet, as more efforts were focused on gameplay functionality.

We couldn't implement the third person camera, because when calling for methods or calling the instance fields of an external java script class under the animate function, the scene freezes. For some reason, accessing the instance field of another class, for example IntervalPosition inside the _interval.js file causes the scene to no longer request animation frames.

The intended implementation of the third person camera is to have a constructor class, called ThirdPersonCamera inside the _3dcamera file, dictate the positioning and look at position of a Three.js camera passed as parameters. This class, however, only focuses on calculating the given step the camera must shift per frame given a number of frames remaining to complete the transition and the destination to arrive at.

To determine the position to arrive at, and the number of frames to permit the transformation of the camera, all such calculations and bookkeeping would have been handled by the IntervalPosition class. For every new path there would be an interval that records the number of cubes on the path; if the path has received interference from a wall; the number of cubes that precede a wall collision; and the next position of the camera where a turn is made.

The third person camera was designed to follow the snake under all gameplay conditions thus far, including the fact that the snake can teleport from colliding with walls, and when it makes a lot of zig zag turns. The camera even accelerates its speed of traversal depending on the length of the path it's rotating itself on. There were so many complications with implementing code to consider all the intricate cases of motion, but this unfortunately froze the scene. Furthermore, we weren't able to implement the viewing of two cameras at once, based on code implementation featured in the Youtube tutorial [6]. It would have been impressive to have an overview camera at the same time.

To compensate for the lack of dynamic viewing, the current camera is capable of changing positions twice. Additionally, to add more flare to the scene, apple types were appropriately textured with images, including the poison apple being decorated as dangerous, and the apples even contrast in shape from the snake's components.

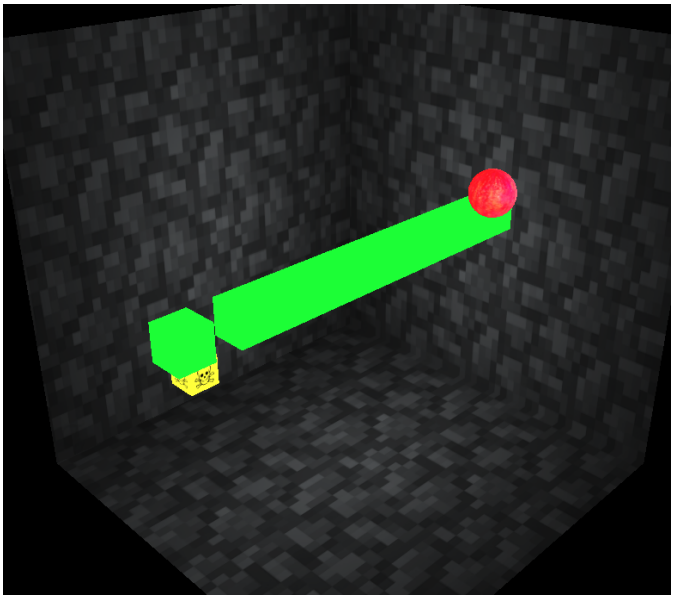

Figure 2: The above image is a long snake.

5 CONCLUSIONS

https://talkingeagle.github.io/project/index.html

All in all this was a fun project to work on because you could see the scene come to life before your eyes. We had some challenges and we learned a lot about coding in 3D graphics. While we didn't get to complete everything we set out to accomplish, we are glad we were able to work on this project because we learned a lot about coding games in javascript and especially three.js.

REFERENCES

[1] Ricardo Cabello et al. 2010. Three.js. *URL: https://github. com/mrdoob/three.js* (2010).

[2] Daniel Haehn, Nicolas Rannou, Banu Ahtam, P. Ellen Grant, and Rudolph Pienaar. 2012. Neuroimaging in the Browser using the X Toolkit. *Frontiers in Neuroinfor- matics* (2012).

[3] Mjurczyk & Maina. (2021, Aug). How do I make a camera following an object regardless of the x, y, and z rotation. Retrieved from: https://discourse.threejs.org/t/how-do-i-make-camera-follow-an-object-regardless-of-the-x-y-and-z-rotation-of-the-object/28974

[4] SimonDev. (2020, Nov 9). Three.js Cameras Explained | Tutorial for Beginners! (JavaScript). Retrieved from: https://www.youtube.com/watch?v=FwcXultcBl4

[5] SimonDev. (2020, Nov 16). Simple Third Person (using Three.js/JavaScript). Retrieved from: https://www.youtube.com/watch?v=UuNPHOJ_V5o

[6] flanniganable. (2022, Jan 10). 3e How to use 2 cameras three.js. Retrieved from: https://www.youtube.com/watch?v=_TtVdWAc9Sc

[7] suboptimal engineer, (2022, Feb). Coding a retro snake game with javascript https://www.youtube.com/watch?v=NE7_oFi57ic

[8]Gianmarco Picarella, (2018) coding a 3D snake game with three.js https://www.youtube.com/watch?v=44SpOZgB0lQ