**Title**: Space triangles

**Team members:** Amelia Chin and Zachary Shaw

**Introduction**

Our game is a 3D flying action game with amazing cinematics based in the Star Wars universe. You play as a Tie Fighter and you have reached the death star canal. The goal is to reach the end with the most possible points while dodging all the obstacles.

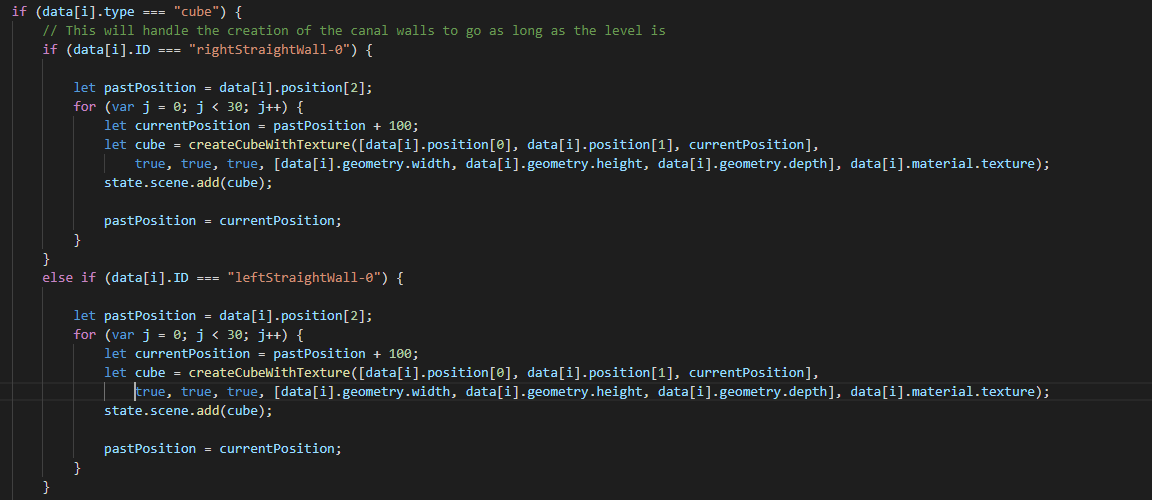
Some of the challenges that we have faced while creating this game were the collision detection, translating asteroids and the json file itself. Initially, the collision detection had a very small window for it to work properly because we currently have a collision box around our main player to detect what is overlapping with our main player. We had it so that it took in a list of objects that collided with the player and it would just collide with all the items in that list. But we ran into issues when we had more than 1 collidable objects right beside each other in close proximity (e.g. 2 asteroids and a powerup)… We later found out that we could read the first object that got detected and only have the player collide with that object. This made the pinpointing of collision much better. We also found that translating the asteroids left and right constantly with delta time was a challenge. With the player’s positioning and how we’ve set up the map, the coordinates were very awkward to deal with. To the left of the player, the coordinates went in the positive direction. To the right of the player, the coordinates went in a negative direction. With this in mind, we had to get the models to translate between position x value of 25 to -25. And finally, the json file was a challenge to handle because I had to keep a mental image of what the game map would be. Keeping in mind of our awkward coordinates to the left and right of the player, I had to keep track of each object respective to the player… and it was a lot of objects. But eventually we decided to use a hand drawn map to keep track of the positionings and future placements of objects.

**Methods**Parts 1-5 includes summary of functionality, implementation details.

**Note**: all functions (with the exception of the plane generation function which I will credit below) are written by us and it utilizes functions that are already built in into Three JS.

For implementation we used the library Three JS. They provided a lot of building blocks and we had to figure out a way to put all those pieces together.   
<https://threejs.org/>

* **Part 1 – Playing field / terrain**: Initially, we wanted to infinitely generate a flat plane so that we can auto spawn obstacles on the X positions. Then we scrapped that plan and decided to keep the small initial plane that we’ve got with the function “setupPlane” in our helperFunctions.   
  It uses the PlaneGeometry functionality already built into Three JS and all we did was hook up the geometry specs, shadows and coordinates for positioning to set up our plane.   
  To save computing power for rendering, we made it so that the plane moves with our main player by changing the positioning of the plane at the same rate as the moving player.  
  For reference we used (credit): <https://codepen.io/DonKarlssonSan/pen/deVYoZ>  
  But their way of implementation was not to our liking.   
    
  As for the walls, we made it so that it takes 2 reference models within the json and generate multiple instances using a for loop to increment the position for the next cube to fit the level design. It uses the createCubeWithTexture functions to load in the geometry and death star canal texture.

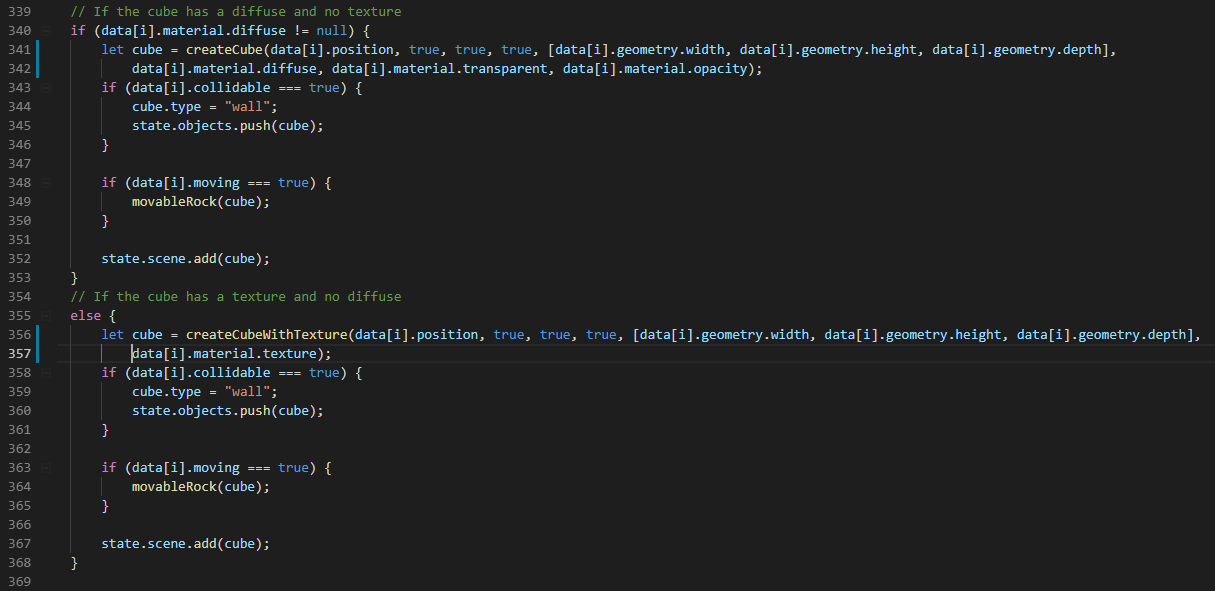


* **Part 2 – Main Player:** Initially, we made a space ship entirely out of triangles (which is where we got our game title!) But we found that we could use a library called Three JS which was super helpful because they have a bunch of built in functions that allowed for the implementation of models.   
  E.g. var mtlLoader = new THREE.MTLLoader(); - Which loads the material.  
  var objLoader = new THREE.OBJLoader(); - Which loads the OBJ file for use  
    
  The functions that we wrote to use these functionalities are “loadModel” and “loadModelNoMaterial” within the helperFunctions script. The difference between these 2 functions is that one takes a material and one does not.   
  To load in a model, we place the OBJ file within a folder and specify the path.   
  Then for models with materials, same process for mtl files. We then can set the positioning, shadows, scales, etc… specific to the models. Once the models are found and all parameters are specified, we add them to the scene using “state.scene.add(object)”.  
    
  To specify the main player, we have a boolean value passed into the loadModel function with the parameter “isPlayer”. This allows for a specification that we predetermined for the player which includes the initial position and attached camera functionalities.   
    
  In the specs, it stated that we had to move the player using keys. However, through testing trials, we found that it didn’t quite feel correct. So, we swapped over to using the mouse to maneuverer around the game. The tie fighter will follow the mouse cursor on the screen and you would use that to move around the map.  
    
  So, we went online to search for a model for our main player.   
  The model we found: <https://free3d.com/3d-model/tie-fighter-94029.html>
* **Part 3 – Add Objects and Interactions with main player**: The objects that we chose to use for interacting with the main player were obstacles and powerups. More specifically the asteroids, cubes and triangles for powerups. Some asteroids are stationary, and some moves left and right.   
    
  We used a json file to specify the type of object, moving boolean (if any), collision, position, geometry, color/textures and scale (if any).  
    
  To detect the objects, Zach built a collision detection which uses an array of all the objects we wish to detect as collidable and use rays that extend out from the player towards these objects. We then used a function called ray.intersectObjects(state.collidableObjects) which will return all the objects collided by the rays and we can then check their distance to see if the distance to the intersect is less than the player to that object’s geometry distance then we have a collision. We also used distance checking for the collidable array so that only objects in the scene that were close enough to “collide” would be appended to the list and any object which had been passed or left our detection range would be removed from the array thus limiting the amount of actual detection required for each frame.  
    
  The initialization of the interactable objects all occur within the function “createObjs” within the game script.  
  Since we have 3 different type of “obstacles” within the json file, we sort the objects by data[i].type which acts like a name for the type of obstacle. From then on, each type of object has a different implementation.  
    
  **Cubes**: Some are sorted by IDs, but the ones that we will care about in this are the ones without IDs.

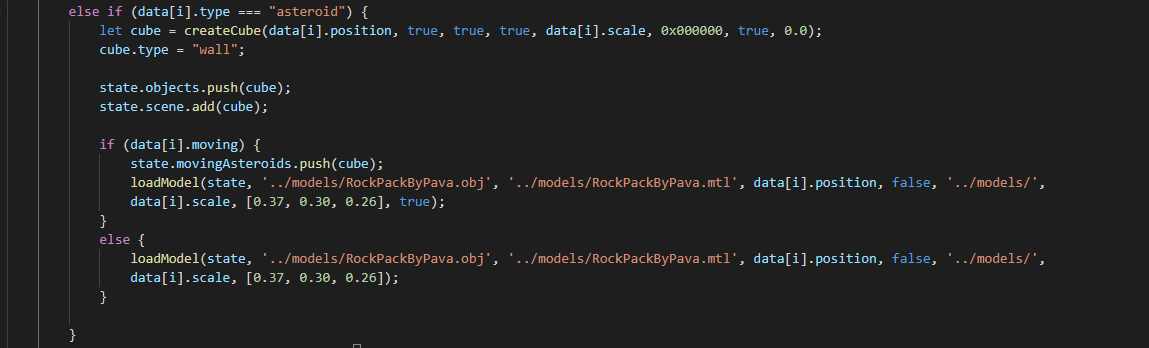
We sorted these by diffuse and texture values attached in the json file.

If the cube came with a diffuse and no texture, we use the “createCube” function within the healperFunction script to create the cube using cube geometry and specify other parameters like shadow, visibility, etc…

If the cube came with a texture and no diffuse, then we use the “createCubeWithTexture” function within the helperFunction script to create the cube using geometry and specific other parameters like the createCube. The only difference is that it will find the texture using a path and load up that texture onto the cube.

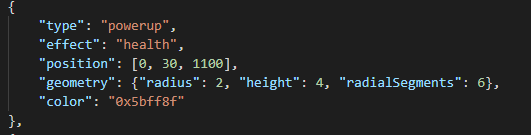
  
  
**Asteroids**: data[i].type === “asteroid”  
There are 2 things that we need with the asteroids.

1. The first is the model itself which we used the loadModel function to load in. We also separated it by the “moving” boolean which helps define the objects that we allow to be moved by translation near the end of the level.
2. The second is the cube that is attached to each model (using same position and scale). This will act as the collision box that is used for collision detection. By assigning the cube.type to equal “wall”, that allows us to check for the type during collision detection and allow for the proper reaction for the main player model. Proper reaction being that the main player will stop moving and take damage for colliding with the asteroids.

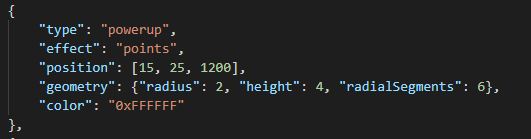


**Powerups**: These are objects that the player can “pick up” and activate a power.   
There are 3 different types of powerups that we’ve created.   
To create these, we pass the values obtained from the json file and feed them into the “createPyramid’ function located in the helperFunctions script.

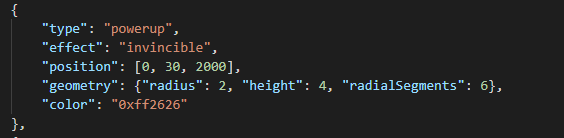
The first is the health type and it is a green triangular pyramid within the game. This allows the player to regain health. It also allows the player to obtain points so it is more incentive to go for these.



The second is the points type and it is a white triangular pyramid within the game. This allows the player to gain points for the score.

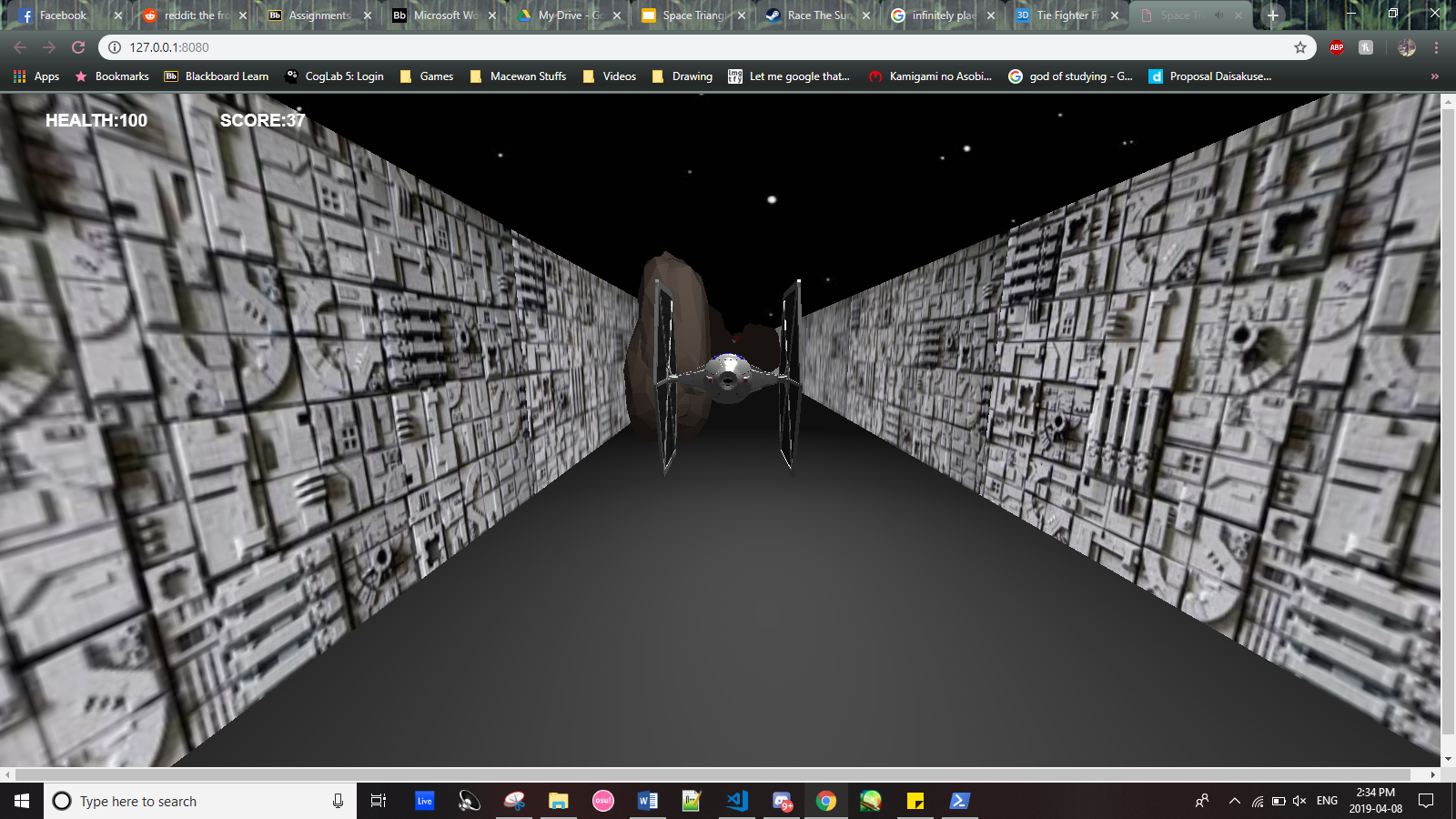


The third is the invincible type and it is a red triangular pyramid within the game. This allows the player to have 4 seconds of invincibility which means that they are free from any collidable objects and do not take damage during that time. For visual effects, we made the main player model a little more transparent to signify this invincibility. After 4 seconds, the model will return to its normal opaque visibility.



**Part 4 – Non-player character:** We chose to use moving obstacles to interact with the main player in a way to cause them to have a more difficult time in avoiding the obstacles.

In addition to the asteroid implementation explained above in part 3, if the “moving” boolean is set to true within the json file for each of the asteroids, we push that asteroid (within the loadModel) and its collision box into the state.movingAsteroids. This is used within the “moveAsteroids” function within the helperFunctions script. What this function does is move those asteroids left and right within the canal at delta time.



* **Part 5 – Change of view:** We made it so that when you press the key “c”, it will change the view from third person perspective to first person perspective.

**Link to theory**  
For modeling, we used three.js functions that helped a lot with creating simple geometry such as cubes, spheres and pyramids. We created custom functions of our own that could be used over and over with very little repetition of code. So instead of 10 lines to make a sphere I would create a function that would take in the important criteria such as the geometry, color, position etc and it would return the sphere. For the models in our game we used three.js’s obj file loader which helped a lot to load in the models and display them in a hierarchy that made sense, where the parent was a group and the children were all meshes.

For viewing, a single camera was created which follows the ship at the same constant speed as the player ship, we use a lookat function that takes in a point in 3D space and focuses the camera on it for looking at the player ship.

For shading and light models, the lighting we used in the game was done using three.js’s light api which allows you to designate exactly what type of light you’d like to use and its properties. We use 4 spotlights that move with the ship (front, left, right, top). We also used an ambient light to help with making the scene brighter and easier to see.

For texture, we used three.js’s .mtl file loader that loads in material files and can apply them to objects quite easily, but for some of our models/objects we use three.js’s simple texture mapping that allows the application of simple .png or .jpg files to objects in the space.

For visibility and transparency, we used transparency a lot for our collision detection which uses invisible boxes around the player and all collidable objects in the game which act as “box colliders”. This is done to reduce the number of rays that need to be cast for each vertex to detect collisions. Each asteroid in the scene has an invisible cube that handles its collision detection. Transparency is also used for a few panes of glass walls in the game as well as on the ship itself when a red powerup is flown over the ship turns transparent and can pass through walls/asteroids freely for 4 seconds.

**Analysis and discussion**:

During this project we made a 3D flying action game using the library Three JS. The objective of the game is to reach the end and earning as much points as you can while dodging the obstacles. We learnt to incorporate theories and utilize built in functions while building the game. Understanding the theories behind the functionalities that the library provided made learning the process a lot easier. If we had more time to work on this project, we could have definitely made it a lot more presentable and better looking. But we stuck to doing what we could do given the amount of time. There wasn’t anything that didn’t work as planned because we planned the things that we know we could incorporate at the beginning. But if we had more time, we would have planned for more functionalities (e.g. multiplayer, more interactive objects, shooting mechanics, more levels, etc…) and a better game play.