Final Programming Project – Max Silin

Three-sphere world simulation

A1 Initial Project Overview

Project Overview

Scientists have hypothesised the idea that our world is actually a 4D sphere. The idea behind this is that we move on the surface of the 4D sphere (a three sphere), thus moving in curved 3D space. This means that any direction you try going (up, down, left, right, forwards, backwards), if you keep on going, you'd end up in the same place eventually, much like on our planet Earth.

However, I want to see this 3sphere world in action, as it would look from our perspective, were it to be real. I will make a simulation of how such a world would look like as you move around it. This would be an interesting experiment and give intuition into curved space physics.

The simulation will be an interactive game – it would involve a rocket that would be able to travel through a space-like world with objects situated around it. The player would be able to control where the rocket moves.

In some scenarios, a finite world without a centre (such as a 3sphere) may be good for multiplayer games where players would always be close to other players wherever they go – producing a more unified gameplay as well as introducing a disorientation factor that would allow gamers to build a new space awareness.

Identifying the Stakeholders

The main category of stakeholders would be **people interested in physics and maths.** They would want to see a 3sphere world because it is fun to experience maths visually. My stake holders will be:

- Lily, a Y12 student.
- Marco, a Y12 student.

The second category of stakeholders would be **gamers** that would like to have fun playing the game. They would probably be interested in playing it as a multiplayer game with someone so that it is more fun. For this, my stakeholders would be:

- Anagh, a Y12 student
- Students from lower years interested in games

Why I need a computer to solve this problem

I want to show the result of mathematical computations visually. The solution is a moving, interactive simulation, which inherently requires fast processing speeds and graphics, making it unachievable without a computer.

To give a clearer idea of the processing involved:

- When a player/user presses the forward key, the rocket must move (change coordinates) through a specified path, gradually. The program will be able to update the rocket's coordinates once every few milliseconds.
- The program will also render a new image of the world (i.e. objects) the rocket sees every few milliseconds as well. A computer may manage this by using parallel processing on the graphics card, which is much better than calculating the colour of every pixel procedurally. By leveraging the power of parallel processing, I will be able to provide a fluid experience of motion, as well as one that is accurate mathematically.
- There may need to be collision effects, so that a player cannot go into the 'inside' of an object.

As well as this, the simulation would also be a game, to make it fun for the gamers to play. This would mean users will need to interact with the game / each other.

All the interaction going on makes my simulation a real time system – the next image of the world must pop up in time for the player to react to changes.

Therefore, the simulation is practically impossible to make by any other means other than a computer, because it needs to be interactive, react at very high speeds, and do thousands of calculations.

A2 Research of stakeholder needs and existing solutions

Interview Planning

I have quite a good idea of what the main task of the project is. I also know that I want to liven the simulation and make it more appealing to the less physically minded audience by making it into a game.

I still need to find out:

- what preferences people have about moving the rocket.
- What genre of game my stakeholders would enjoy most.
- Any new ideas that would play nicely with the project.
- What hardware people use and how they'd expect to play the game.

However, I will be contacting my stakeholders later on in the development stage, too, to decide more niche features, for example:

- Game specifics – would they like asteroids, planets, stars, or a mix?

Since the stakeholders are my age or below, I will use more informal language to make their answers truly reflect what they think.

Questions for physics stakeholders

"As you know, I am also a physics fan. (Explain 3 sphere concept). I haven't found any similar simulations yet, but I just want to understand what effects we will notice if our world was a 4D sphere. So I am making this."

1. What is most interesting (to you) about this idea?

(This is so that I can see what part of the simulation my stakeholders would be looking forward to most)

2. What other physics rules (like gravity) would you also like there to be?

(To get ideas, if anyone has any good ones.)

3. In terms of the simulation, would you like to use it for anything specific (would you like access to mathematical data) or would you play it for casual interest / fun?

(To tell me the intention of my Physics stakeholders. If they would actually need to analyse mathematical data from it then I would have to make an interface to extract that data.)

4. What device would you want to use to see the simulation?
5. How do you imagine moving around the world? Would you like to change the speed at which you move - for example fast and slow?

(To get an idea of what movement would be most intuitive.)

6. Would you like to download the simulation onto (their device) to play, or play it from a browser?

Other comments?

Questions for game stakeholders

(This would be in the form of an interview)

"I am making a space game – yes, you get a rocket. Imagine planets around you, massive stars and spacious expanses of nothingness. However, there is something different about my game – the space is curved. So if you go straight on far enough, you may end up in the same spot you started! That's the world. What I need from you is to tell me what sort of game you'd like this to be. Please answer some of the following questions so that I can tailor suit the game to what you'd enjoy most."

1. Pick your favourite game mode and rank the rest:
- Escaping gravity for as long as possible (i.e. don't splat yourself on a planet or burn in a star).
- Race your friend/ghost from one planet to another. (Remember though – going backwards may be as good as going forwards – space is curved and there may be massive planets in your way)
- Mess around with power ups.
- Space hide'n'seek - evade your friend by hiding behind planets.

(To understand what game genre most people like.)

2. What would you favourite power up be for your favourite game mode?

(To get ideas)

3. Also rank these modes:
- Play against your own ghost
- Play against your record.
- Play against someone else's record.
- Play against a robot
- Play against your friend (on the same keyboard)
4. What device(s) do you use to play (3D) games?
5. How do you imagine moving around the world? Would you like to change the speed at which you move - for example fast and slow?
6. Would you like to download the simulation onto (their device) to play, or play it from a browser?
7. Space for other comments or ideas.

Primary Interview

Interview 1 – Lily

Yr12 Physics Fan – doing Physics and Maths A level.

Max: "What would be most interesting to you about this idea?"

Lily: "I think the interface. Like what you'd see when moving. Because I can't visualise it yet."

Max: "What other physics rules (like gravity) would you also like there to be? (Collisions between objects or gravity or nothing)"

Lily: "Gravity would be nice. But perhaps with an on/off switch so that you can make use of both modes."

Max: "Collisions? Would you want to send asteroids flying away when you bump into them, or just stop?"

Lily: "Well, collisions would be quite complicated to do..."

Max: "Should I leave them out, then?"

Lily: "Yes. And they're not that exciting anyway, are they?"

(At this point Lily asked about the other part of the project – what game it would be)

Max: (Told her about game modes I am thinking of)

Lily: Prefers to race against her high score /record. Would be nice to save high scores to race against another day.

Max: "In terms of the simulation, would you like to use it for anything specific (would you like access to mathematical data) or would you play it for casual interest / fun? "

Lily: "Just casual, I guess... If you could put comments in the code so that I can understand the program that would be nice.

Max: "Ok. So just comments here and there to explain the maths?"

Lily: "Just general explanations should do, I think, yes."

Max: "What device would you want to use to see the simulation?"

Lily: "PC. It would be too small on an ipad or phone, wouldn't it?"

Lily: "As for controlling it, I'd use a keyboard. However, I guess you can also just use a mouse. "

Max: “Hmm, I didn’t think of that. How would that work?”

Lily: “You could click an area to move to it. Or scroll to go forward and back. I don’t mind keyboard though.”

Max: “How do you imagine moving around the world? Would you like to change the speed at which you move - for example fast and slow?”

Lily: “(About speed) Would be nice.”

Max: “And would you want to go at two (discrete) speeds – fast and slow – or choose to go at any speed you wish (continuous spectrum)?”

Lily: “Continuous would be nice. Though I don’t know how I’d control it?”

Max: “Hmm, I’ll think about that.”

Max: “Would you like to download the simulation onto your PC, or play it from a browser?”

Lily: “Browser”

Evaluation of Interview (her opinion):

- Gravity would be a nice extra – but to be turned on and off.
- PC is the preferred device for seeing the simulation.
- Liked the race idea. Proposed to race against your top score, and save it to race against another day. The timed scores can definitely be included as a feature (if a race is the game mode generally preferred). However, saving the scores may be difficult, especially from a browser. A backend server or use of cookies will then be required. Saving scores may be a nice extra feature that can be added if I have enough time.

Next steps:

- Made me realise that there can be different controls for moving the player – using a mouse, a keyboard, or both.
- Continuous speeds may be more natural for the simulation – but how you’d control the speed needs to be decided.

Research into other solutions

I have researched some other similar solutions. This is to understand common difficulties I may face and to understand whether the project will be feasible. It would also be useful to find out how characters move in these games/simulations.

Hyperbolica by CodeParade

Video of what world looks like: (https://store.steampowered.com/app/1256230/Hyperbolica/)

(Non-Euclidean Geometry Explained - Hyperbolica Devlog #1

) (Spherical Geometry Is Stranger Than Hyperbolic - Hyperbolica Devlog #2

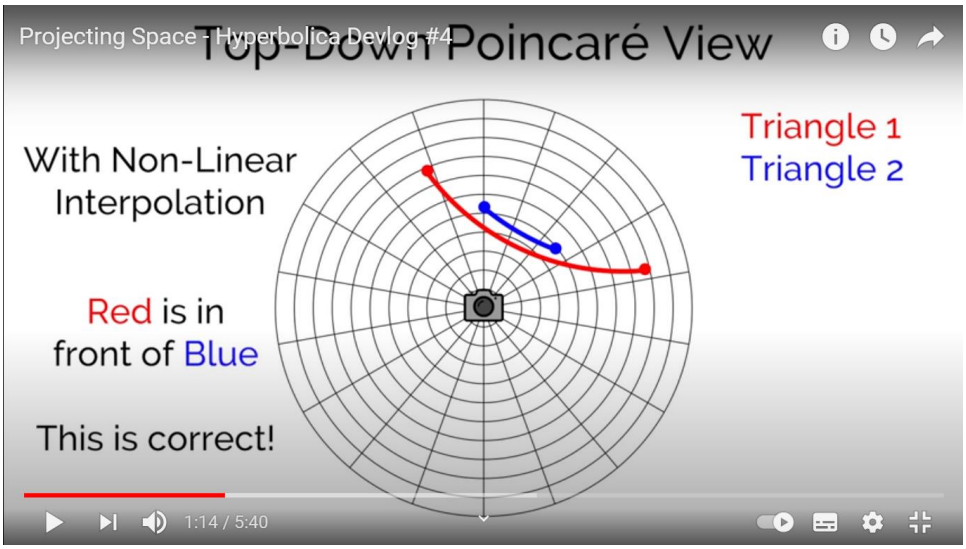
)

Summary of Hyperbolica:

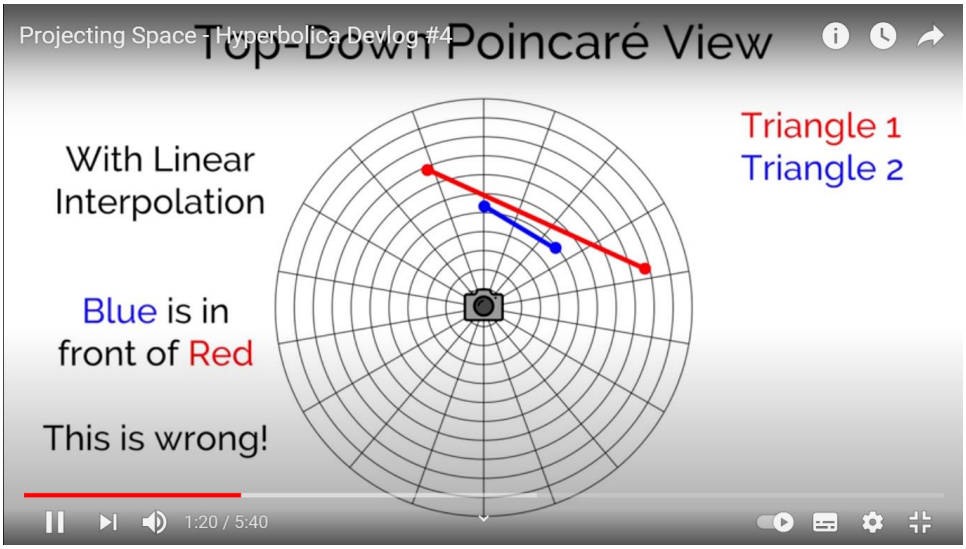
- implements spherical geometry.
- implement a 4D sphere to 3D world approach.
- Not much documentation
- nice mist that gives perception of distance– this gives the player a better sense of perspective. Perhaps in my world farther planets should appear darker rather than lighter, as they are in space.

In Hyperbolica Devlog #4, he highlights a common bug that can occur. He talks about how fragment shaders can switch which object is in front of which. This is because it interprets the edges as straight lines, when they would actually be curves (in some projection modes like Poincare).

How lines/surfaces should appear if user is the camera:



How fragment shader would make them appear:



The result is that the order of objects is changed (blacked out windows):

To fix this, he instead uses a projection which “preserves linearity”, (I.e. straight lines turn into straight lines – no curves), called the Beltrami Klein model.

The analogy to this Beltrami Klein model (used for hyperbolic spaces) is the gnomic projection (used for spherical spaces). If I ever get a bug like that, I may need to use that projection.

Hyper Rogue

(Sources: https://www.roguetemple.com/z/hyper/ , https://www.roguetemple.com/z/hyper/dev.php)

Summary of HyperRogue:

- game that implements 2D spherical and hyperbolic projections.
- implements stereographic projection and uses tiles instead of a continuous space.
- player clicks to go to a tile. This was quite annoying because the I have a laptop with a touchpad, not a mouse, so it was difficult to move my pointer to the correct tile and click every time.
- My world will be continuous, so I do not need to store tessellations of tiles, like they have.

However, to implement continuous movement and mechanics (like shooting a bullet)...

For continuous game mechanics (i.e., the shmup mode and the racing mode in HyperRogue, movement animations, etc.), every object remembers the discrete cell it is on, and a transformation matrix (`at`) to map from that **cell's internal coordinates** to object's model coordinates. This two-layer relative approach **allows us to do precise calculations on the region of hyperbolic plane** around the player, and at the same time do not lose precision for the monsters which are very far from the location of the player (such monsters do not act, and their `at` will simply be still valid when they get back into the sight range).

They make a separate coordinate for every object to say where exactly it is located in the tile. These coordinates can be manipulated mathematically. Once it reaches the end of a tile, it is 'teleported' to the adjacent one.

It is common to use the Poincaré disk model to explain hyperbolic geometry; this is also the projection used by default in HyperRogue. However, for computational purposes, Minkowski hyperboloid model is usually better and more natural.

The continuous movements along tiles use Minkowsky or Poincare geometry.

What I gathered from this research:

- Some players will not have a mouse (on laptop), so moving a pointer and clicking may be difficult for them
- These two continuous geometries (Poincare and Minkowsky) can probably be adapted into the fourth dimension to produce the results I am looking for with my 3D world.
- It would also be interesting to see why the Minkowski geometry is better for computational projects so that I can avoid design mistakes in my own program.
- hyperbolic coordinates will make the precision of the position of the object deteriorate the further away it goes from the centre.
- This is not the case with spherical geometry, which I am using.
- However, I may need to analyse whether to store 4D coordinates of objects, or 3D spherical coordinates.

HEXGL– A track high record game

https://hexgl.bkcore.com/ - Home

https://hexgl.bkcore.com/play/ - Play the game

https://github.com/BKcore/HexGL - Github code repository

HexGL allows you to

- race against a timer
- on a winding track, suspended high in the air.

It implements a **curved track** – it may be interesting to explore the code behind it to find out how the movement is implemented.

What I also liked about it:

- Uses Left and Right arrows for sharp turning.
- Uses A and D for more fine-tuned turning.
- Uses the **forward key to accelerate** – this is a great solution to the problem of having a continuous spectrum of speeds, discussed in the interview with Lily.
- Automatically decelerates.

- Intuitive start screen giving necessary controls:

- It also determines when the spaceship **collides with a border, and stops it**. The implementation behind this may be interesting to explore.
- The game can be played from a browser – it uses WebGL and Three.js as frameworks to render complex graphics. I observed no lag which shows that **running games from a browser does not hinder the performance too much** compared to running it as an installed program.

Secondary Interviews

I now have ideas from the games I've researched and am intending to also ask stakeholders about how they feel about them.

Interview with Anagh

Max: "Pick your favourite game mode."

Anagh: "I think a race. Yeah. I feel that would be good as it would allow you to experience the whole world. If you also involve power ups, you can be smart with how you use them. "

Max: "Do you have any specific power ups in mind?"

Anagh: "Maybe going high speed, or maybe putting asteroids in front of the other player. I guess weapons would also be interesting to use."

Max: "Yes. Imagine firing a rocket or laser. It would follow the curvature of space, too, highlighting this to you."

Max: "What game mode would you prefer: racing against (A) your own ghost (B) a robot (C) your friend (on the same keyboard) or (D) Single-player (against a timer) "

Anagh: "Playing with someone. "

Max: "Even, if it's on the same keyboard?"

Anagh: "Yes, that should be fine. If not, or if I don't have someone to play with, I guess playing against a timer would be good enough."

Max: "Lily proposed the idea to race against a timer. I think it would be nice to have different race courses. For example, from planet A to B and planet C to D."

Anagh: "Yes, I like that idea, actually. Then people can discuss tactics of winning different races."

Max: "And would you like these courses to be randomly generated or have pre-set courses?"

Anagh: "I think pre-set courses would be better – so that people can discuss tactics..."

Anagh: "What do you think the obstacles will be like?"

Max: "I'm open to suggestions."

Anagh: "I think it would be nice to have asteroids. With the power ups, you can get a shield to protect against them, but if you're unprotected, you'll damage your ship. Also, black holes, so that you can get sucked in if you approach them too close. Or same idea with a dense neutron star."

Max: "Ok, or maybe you burn if you approach a star too close..."

Anagh: “Yes, something like that.”

Max: “What device would you use to play this game?”

Anagh: “Computer. Or maybe a phone.”

Max: “How do you imagine moving around the world? Would you like to change the speed at which you move - for example fast and slow?”

Anagh: “I normally use WASD to move with a keyboard. You can also get like an accelerating bar... So, if you press a key, it accelerates you.”

Max: Because that reminds Max of HexGL, Max tells Anagh about it: about using a key to accelerate, keys to turn at different speeds. Asks Anagh’s opinion.

Anagh: “Yes, I think turning with different keys is also a good idea...”

Max: “Would you like to download the simulation to play, or play it from a browser?”

Anagh: “A browser would be preferrable. But both options will be fine.”

Anagh seemed keen on the idea of having different obstacles like black holes and neutron stars that would suck you in.

He also expressed an interest in having a range of power ups to improve your ship/rocket’s defence and speed.

He liked the idea of racing from planet to planet.

Interview with Marco

Said it sounded similar to the game No Man’s Sky.

Max: “In terms of the simulation, would you like to use it for anything specific (would you like access to mathematical data) or would you play it for casual interest / fun?”

Marco: “Just for fun.”

Max: “And would you like to understand the Maths behind it a bit?”

Marco: “Yes, I think so.”

Max: “Would you like a guide or something different?”

Marco: “Yes, I think a guide would cover everything. But if you wanted to, you could maybe make in game explanations... At the start of the game, the people would choose whether they want detailed explanations or just some general knowledge.

For example, if they picked the easy option, the explanations may tell you that black holes suck objects in. Maybe relative weights of stars.

However, if they picked more detailed explanations, they would be told about how the maths behind the simulation works.“

Max: “Yes. I probably won’t be able to implement explanations integrated into the game, but I can make separate guides.”

Marco: “Okay. These are just ideas, so I’m fine with that.”

Marco: “I thought also, what would the objects in the world be?”

Max: “I thought planets but I’m open to ideas.”

Marco: “I just thought, you could also have an atomic world, where the planets become atoms, and gravity could become the strong force and so on...”

Max: “That’s a cool idea! Maybe I can make different textures to theme to world.”

Marco: “And then the user can switch between the two…”

Max: “Okay, maybe.”

Max: “What device would you want to use to see the simulation?”

Marco: “PC.” (Personal Computer)

Max: “How do you imagine moving around the world? Would you like to change the speed at which you move - for example fast and slow?”

Marco: “Mouse for looking around. However, so that I know where I’m going, I want the viewpoint to point forwards once I start moving. Or left if I turn left, I guess. (about changing speeds) What, like a shift key?”

Max: Tells him about the option of an accelerating bar.

Marco: “Yes, that would probably be better.”

Max: “Would you like to download the simulation to play, or play it from a browser?”

Marco: “Download it.”

Interview Analysis

Marco had a lot of ideas about making the game more scientific in other areas. For example, he proposed viewing the atomic world vs the galactic world. He also proposed to make the simulation explain different scientific knowledge, such as electrons orbiting the atom.

Since I think the scope of the project should be the curvature of the world, I think I will refrain from implementing lots of scientific explanations.

Anagh had similar ideas, instead emphasising his desire for different obstacles – black holes and neutron stars. I think it should not be too difficult to implement this and the excitement of trying to avoid dangerous obstacles would add to the game. It may be a nice extra feature that I can implement.

All the people I asked would play the simulation for casual interest.

However, Marco and Lily both wanted explanation into the maths behind the simulation. I will produce a brief guide to describe the workings of the physics of the game as part of my documentation.

<u>My opinion</u>

My stakeholders have given me a range of ideas. However, because I need to finish the project within a year, and because I am also an interested stakeholder, I will single out my opinions as to what the game should look like.

1. I think the 3sphere concept should be the focus of the simulation – so people need to understand t more through seeing. Therefore, I will include an ability to shoot beams of light – which will highlight how light travels in a 3sphere world.
2. I understand that just exploring the world without an objective may be boring. So I will make a series of race courses – planet to planet – that you can choose to play. These will be timed for single player. There will also be the option of adding a second player to race against. The second player will have alternate controls on the same keyboard.
3. My stakeholders and I prefer freedom of movement. Therefore, I will include an acceleration bar, as in HexGL.
4. Because Marco said he’d be interested in seeing how the game would look if planets were swapped for atoms, I may try to incorporate freedom as to the textures of objects. I will keep this in mind when designing the simulation, to try to allow the program to scale well to include different themes.
5. I will program the game to run on a browser, instead of a desktop application. I will explain my reasoning behind this later.
6. Scores will not be stored on the back end. If a player stores their scores by logging in, this will act as a barrier to playing the game. It will also need a lot of implementation and backend resources to store the scores, all for a feature that is not too necessary. The alternative to user accounts is users simply entering their nickname if they got a high-score – and this will be shown to all users. However, this would introduce legal and ethical issues as the players could enter

their own names or personal information accidentally instead of a random nickname – and this will be available for all to see. Scores may be stored temporarily on the client side, during a playing session, though, if the functionality is wanted.

7. Gravity will not be implemented. I think people will find the curvature of my world weird enough – gravity will just confuse them more. There is also the caveat that gravity may work differently on a 3-sphere so I don't want to spread wrong intuition to people.
8. The game will not show how the maths behind it works. Instead, I will make comprehensive guide that physics stakeholders can access to better understand the workings. This is because I feel like comments in the code would not be read very easily by people who may have little or no experience programming. However, I will still include them for programmers wishing to understand how my solution works.

A3 The Proposed Solution

The solution would be a game playable from a browser.

I decided the game would be playable from a browser because:

- most people I spoke to prefer to try out the game on a browser rather than waiting to download and install it.
- It would be more reliable for newcomers to try out my game if they don't have to download it – downloading a game from a random website requires trust because some such games are intended as a trojan horse and may infect a user's computer with malware or spyware. By playing the game from a browser, the user reduces the risk of a malware infection.
- I have more experience with browser scripting (I know JavaScript but not C++), so I can make start screens and menus much more easily than on desktop.
- The downside is that the game will run more slowly due to browser security checks. However, seeing as HexGL managed to program their game with WebGL, and my graphics will be less complex than theirs, I do not think performance will be a big issue.

Technologies used:

- I will use Web Graphics Library (WebGL) to communicate with the GPU. This would allow me transform 4D coordinates to coordinates on the screen, for each vertex, simultaneously. This would mean the frames would be updated fast enough to give the effect of smooth motion.
- I will use HTML and CSS for the start screens and menus of the game/simulation.
- Javascript will be used to provide the logic of the game, and to configure different settings. For example, I will use javascript to monitor key presses and pause the game, for example.
- I may use secondary frameworks such as three.js to help with rendering but this should not be necessary.

<u>Hardware and Software Requirements</u>

To play the simulation, you would need:

- A PC/laptop with a keyboard.
- A graphics (GPU) card inside – for example, an Intel HD Graphics 4000.
- An internet connection, and access to a DNS server to access the simulation online. After the webpage loads, internet connection will not be required.
- An operating system, such as Windows 10.
- A browser that supports WebGL and Javascript. Most major browsers (Chrome, Internet Explorer, Safari, Mozilla and Opera) do support it.

Features

I have compiled three tables of features for my project.

1. These features will definitely be included...

Base Features
Player will be able to move through a 3-sphere world.
There will be objects like planets around – these may vary in design or size.
A player will not be able to go through objects.
Firstly, a player will be able to explore the world at ease.
Secondly, the player can choose to play a race. In a race, the player will race against a timer to get from one start planet to another.
Alternatively, another player may join the first. The two players can race against each other using the same keyboard.
The game may be paused.

2. I will pick which of these features I or my stakeholders would like most, once I finish the base functionality. I think allowing the stakeholders to play with a basic product first would highlight what exactly the simulation is missing.

Features that may be included
Ability to shoot light beams to gain understanding of how the space is curved.
Obstacles that may damage your ship – stars and black holes – may be included.
Multiple different racecourses to choose from.
Going from race AB straight to race BC without stopping.
Power ups to use in a race.

Storing scores on the client side.

3. Features that will be too complex or unnecessary to include.

Features that will not be included
Gravity.
Mathematics and explanations shown as the game is played.
Storing scores on the back end.

Success Criteria

Criterion	**Evidence**
The forward key moves the rocket forwards.	As a video
The left and right keys (WASD or arrows) turn the rocket and viewpoint left and right.	As a video
Forward motion and turning can be combined.	As a video
Objects/obstacles would appear on the screen, as they would look in a 3sphere. As you move towards an object from the other side of the 4sphere, it should appear to shrink, then grow.	As a video with evidence of the object being on the other side of the sphere to start with.
Objects closer should appear in front of objects further away,	When in line with two objects, the first would block the second.
If a player arrives at an object's boundary, they should be stopped.	Video.
The player can pause the game by pressing escape or equivalent at any stage.	Screenshot of pause screen in race mode and in exploration mode.

Criterion	**Evidence**
The player can choose a race from a menu. There is a button to open the menu.	Screenshot of button and menu.
There is an option to join another player when racing. Both options should have buttons.	Screenshot.
If single-player, the race will start when a timer starts counting. The timer will be visible all through the race.	Screenshot of timer.
The destination planet will be highlighted.	Screenshot.
Once a player gets to the destination planet, the race finishes.	Video.
Once a race is finished, the time for the (winning) player is shown.	Screenshot.

If two players are playing, the screen splits into two windows. Each player can then control their own rocket independently.	Video of two people playing.
Once a race finishes, the player returns to exploration mode, once a button is pressed.	Button screenshot and exploration mode following screenshot.
If two players were playing a race, the screen returns to being one player, once race mode is exited.	Series of screenshots.

Design Stage

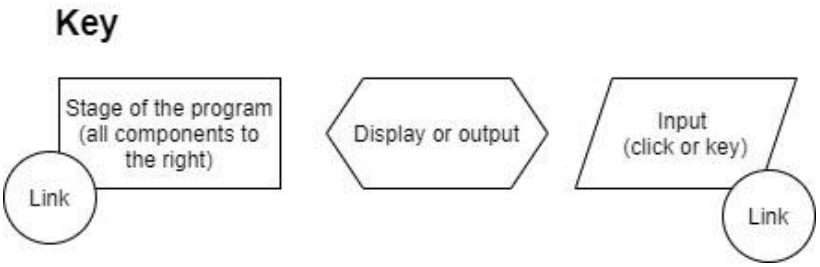
D1. Screen Stages User Journey – display, processing, input summary.

_____

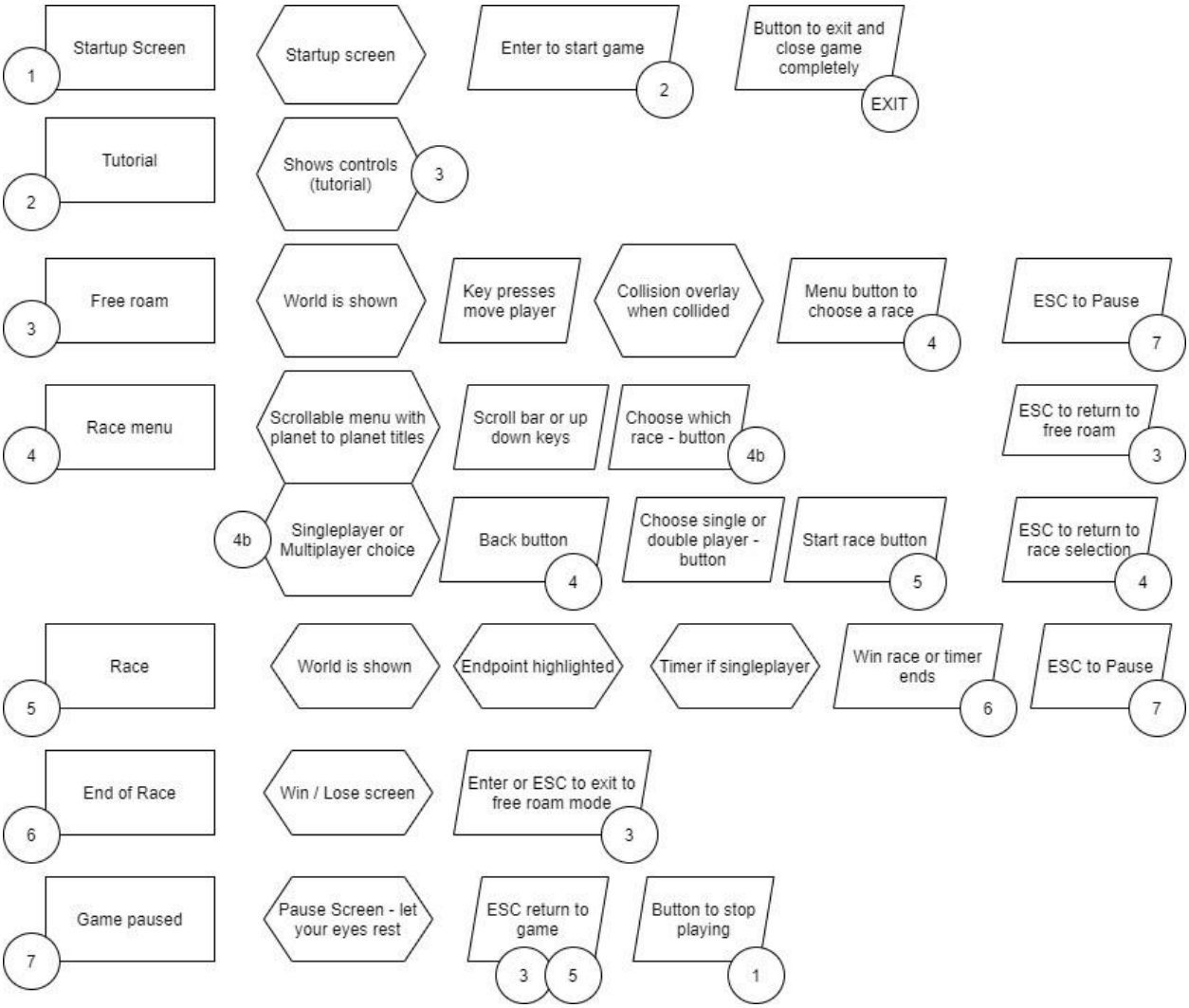
Understanding the diagram:

- Sharp rectangles are the screen / stage of the game.
- All the things found at that stage are to the right of it.
- Hexagon is a display. Parallelogram is an input (click, or key press).
- Links to the right of an input lead to stages with the links on the left.
- The link numbers correlate to screen designs shown underneath.

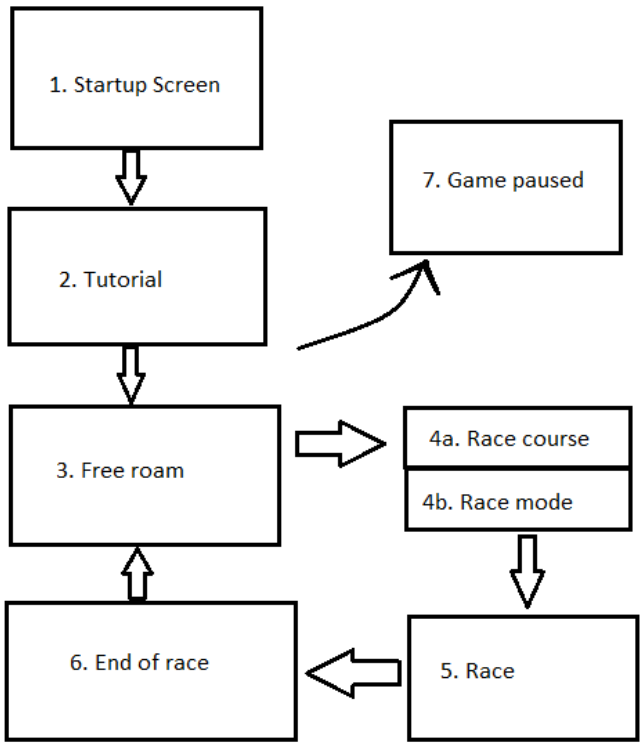
_____



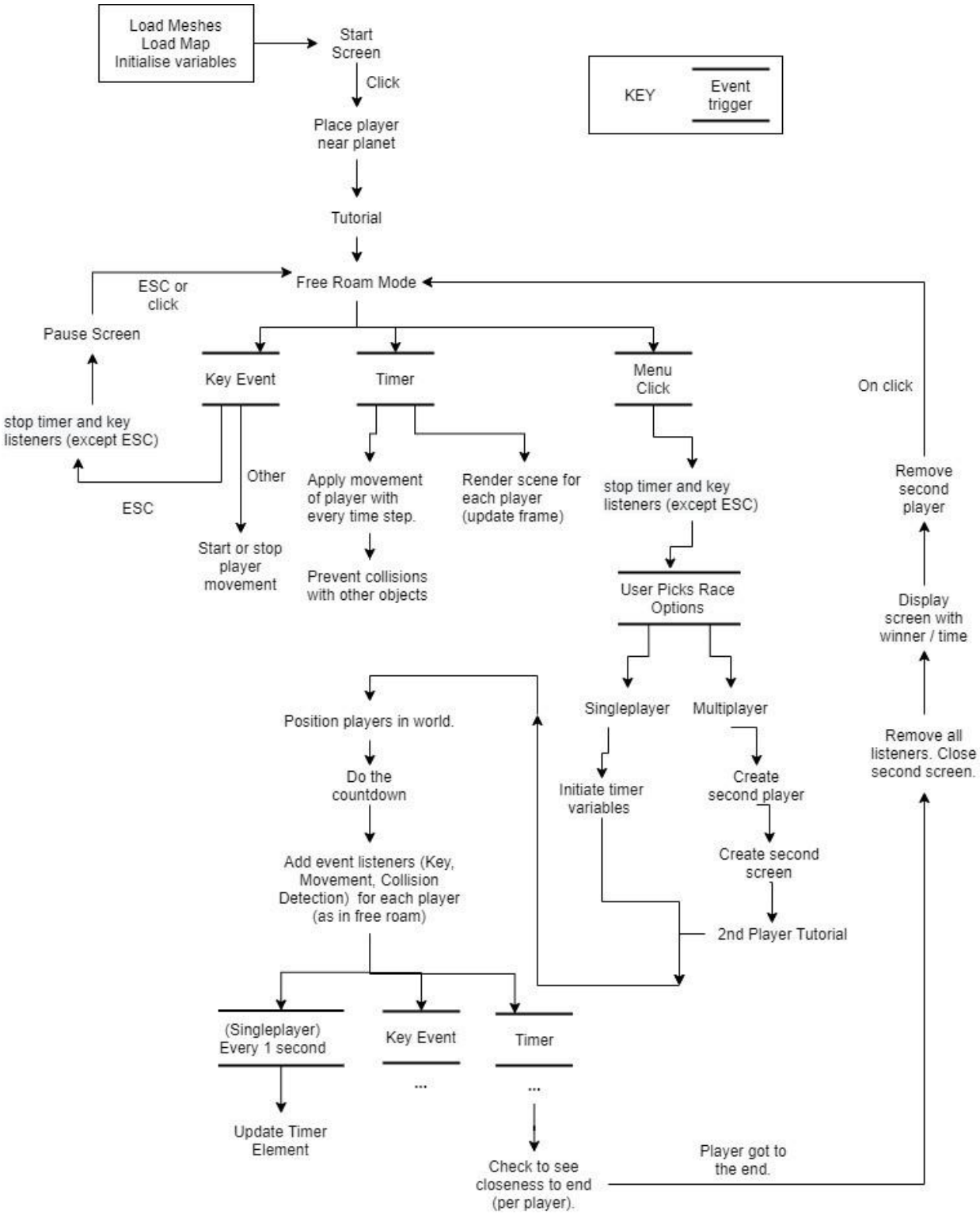
_____



A simplified flow diagram:



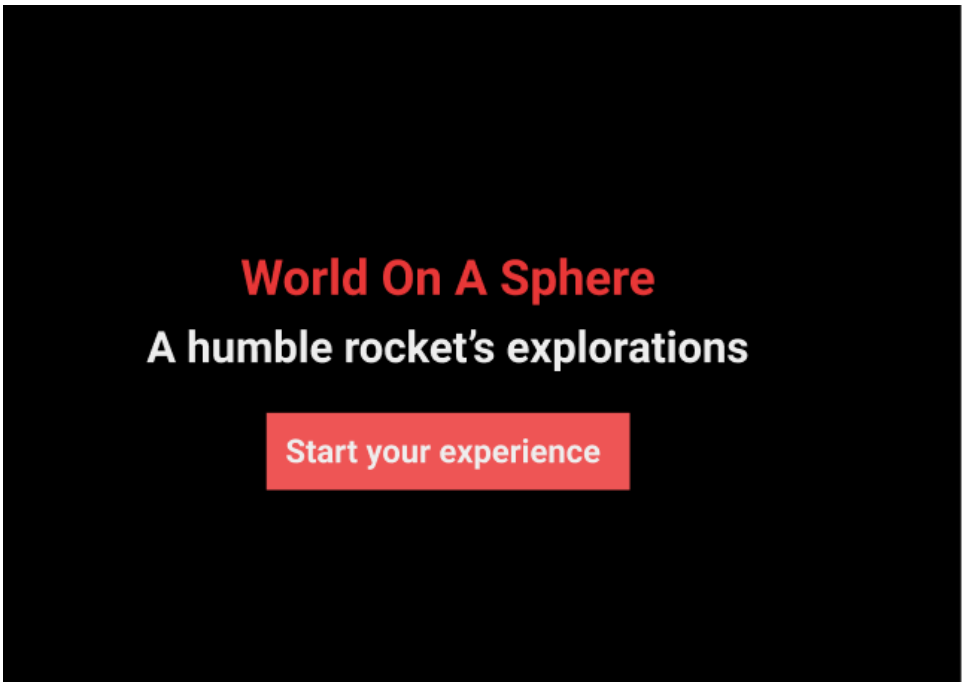
A more detailed flow diagram:


Load Meshes
Load Map
Initialise variables
Start Screen
Click
Place player near planet
Tutorial
Free Roam Mode
ESC or click
Pause Screen
stop timer and key listeners (except ESC)
ESC
Key Event
Other
Start or stop player movement
Timer
Apply movement of player with every time step.
Prevent collisions with other objects
Render scene for each player (update frame)
Menu Click
stop timer and key listeners (except ESC)
User Picks Race Options
Singleplayer
Multiplayer
Initiate timer variables
Create second player
Create second screen
2nd Player Tutorial
Position players in world.
Do the countdown
Add event listeners (Key, Movement, Collision Detection) for each player (as in free roam)
(Singleplayer) Every 1 second
Update Timer Element
Key Event
...
Timer
...
Check to see closeness to end (per player).
Player got to the end.
Remove all listeners. Close second screen.
Display screen with winner / time
Remove second player
On click
KEY
Event trigger

D2. Concept of game and Screen Designs

1. Startup Screen

This is the screen shown when the user starts the website. Clicking the button sends you to the tutorial.

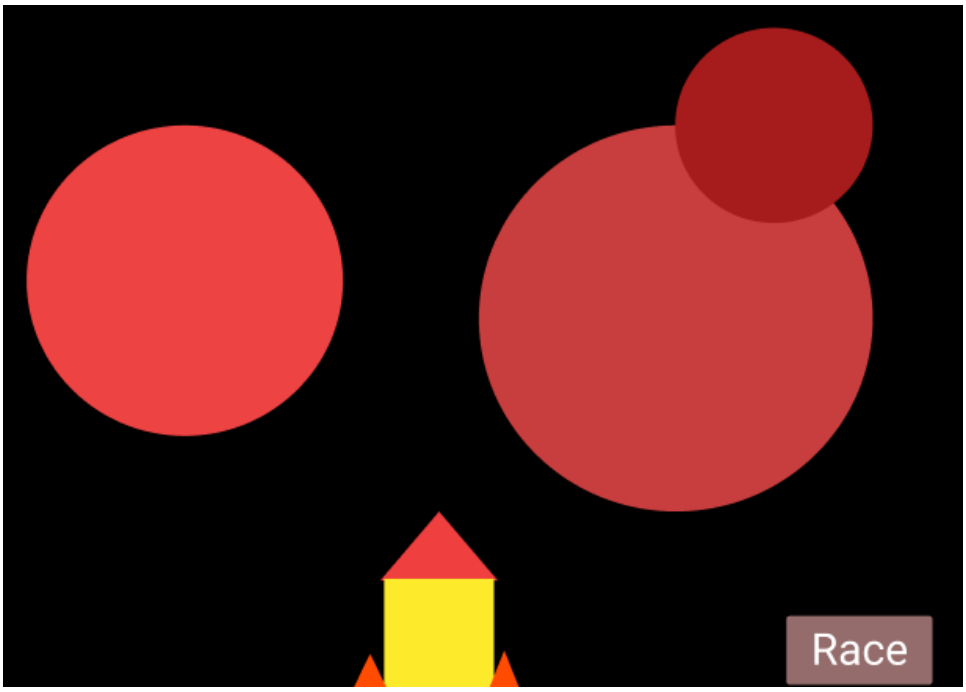


2. Tutorial

Control keys are shown one by one. The player moves and sees what each key does. A prompt is shown to help user locate keys. Pressing ESC skips the tutorial to go to Free Roam mode. The tutorial automatically moves to free roam mode.

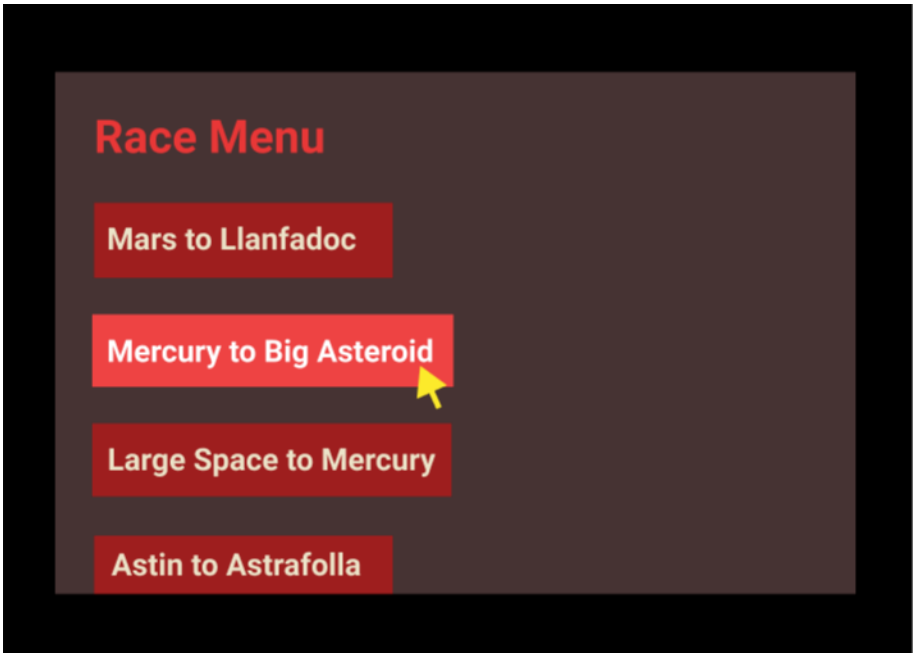
3. Free roam

The user can explore the weird 3 sphere effects by moving around the world. There is only one button – leads to race menu. Rocket is in the centre – intuitive that you control it. Pressing ESC gets you to Pause screen.

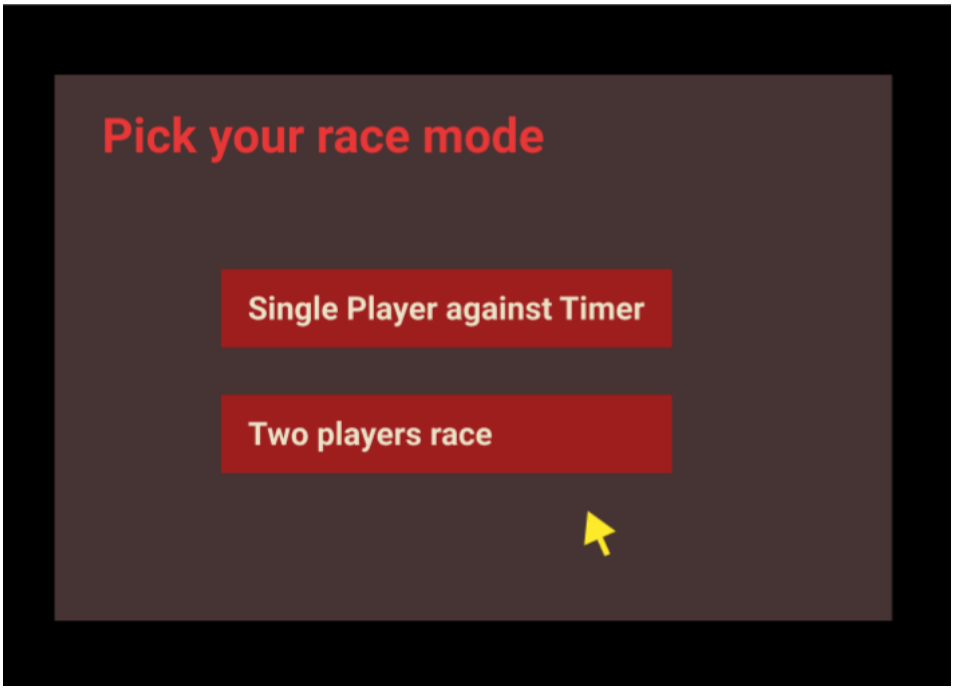


4a. Race Menu

The player can choose the location of race. Races will only have names (no screenshots) so this is the most compact design. There will also be a scroll bar to indicate that you can scroll, if need be.

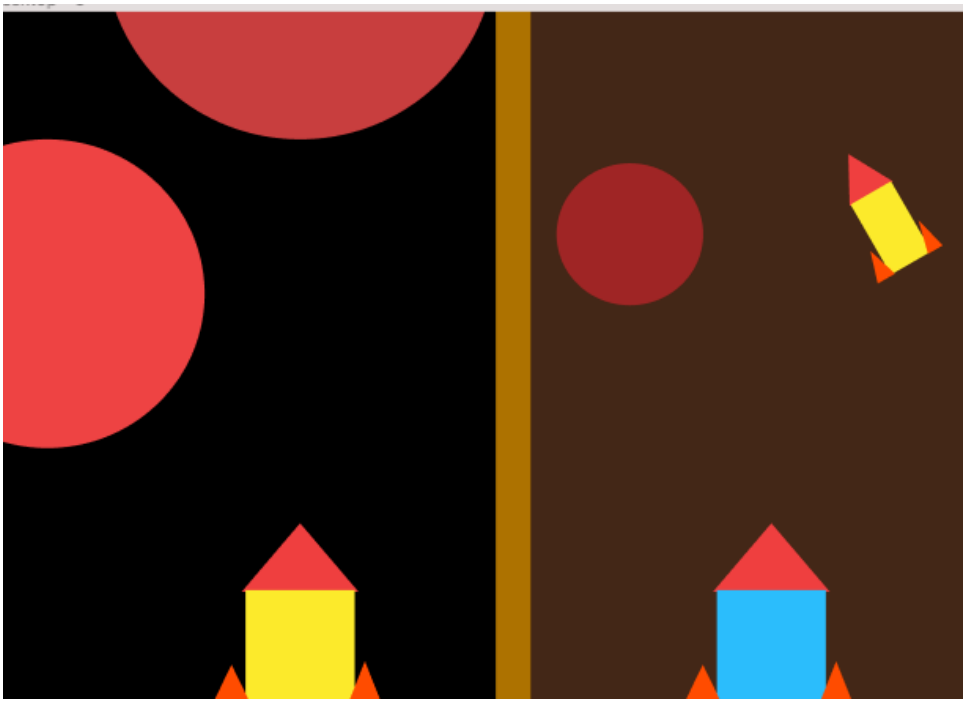


4b. Single Player or Multiplayer choice.



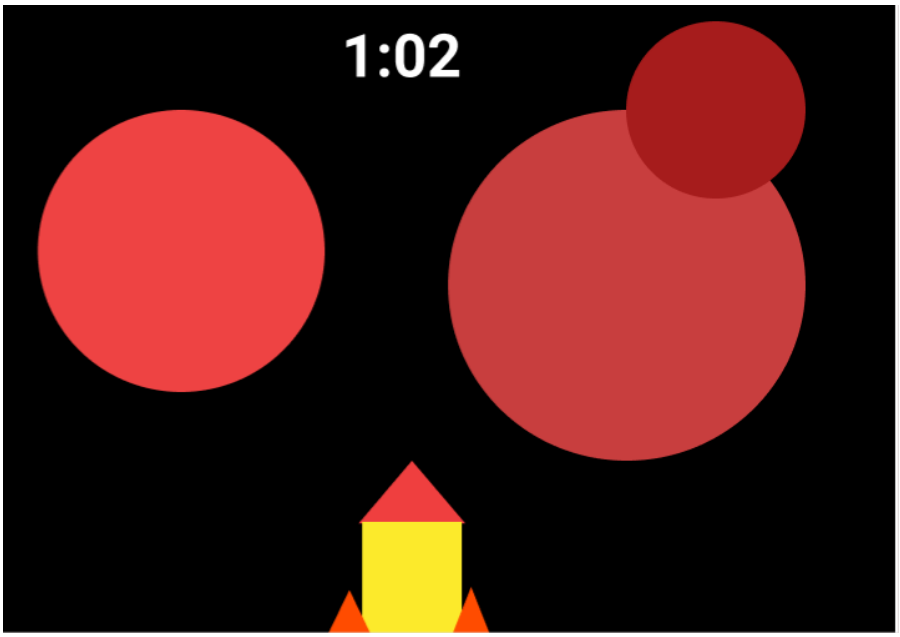
5a. Race mode - Multiplayer

Players race. Barrier and different tint to separate players. Players can see each other.

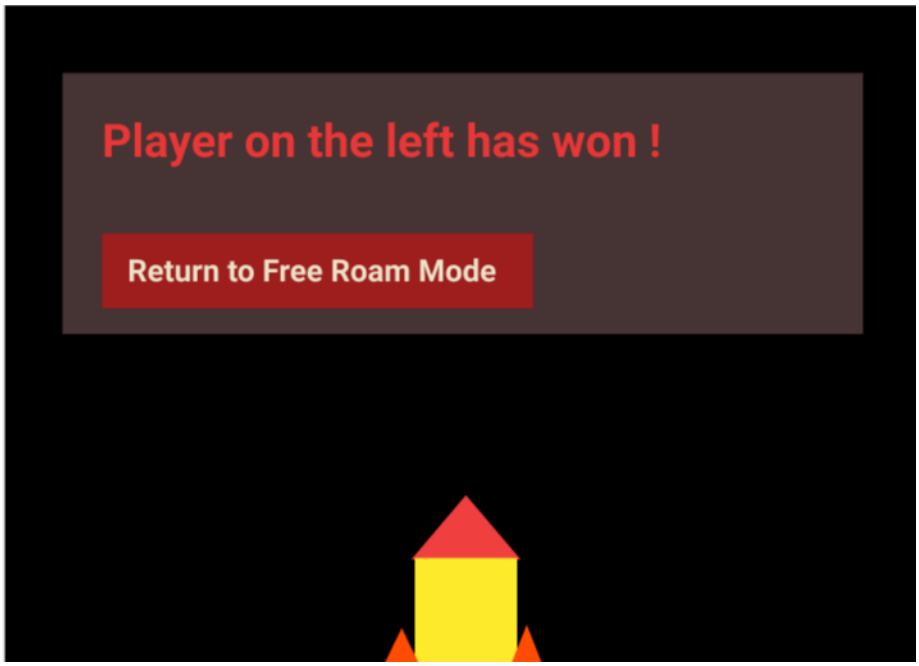


5b. Race mode - single player

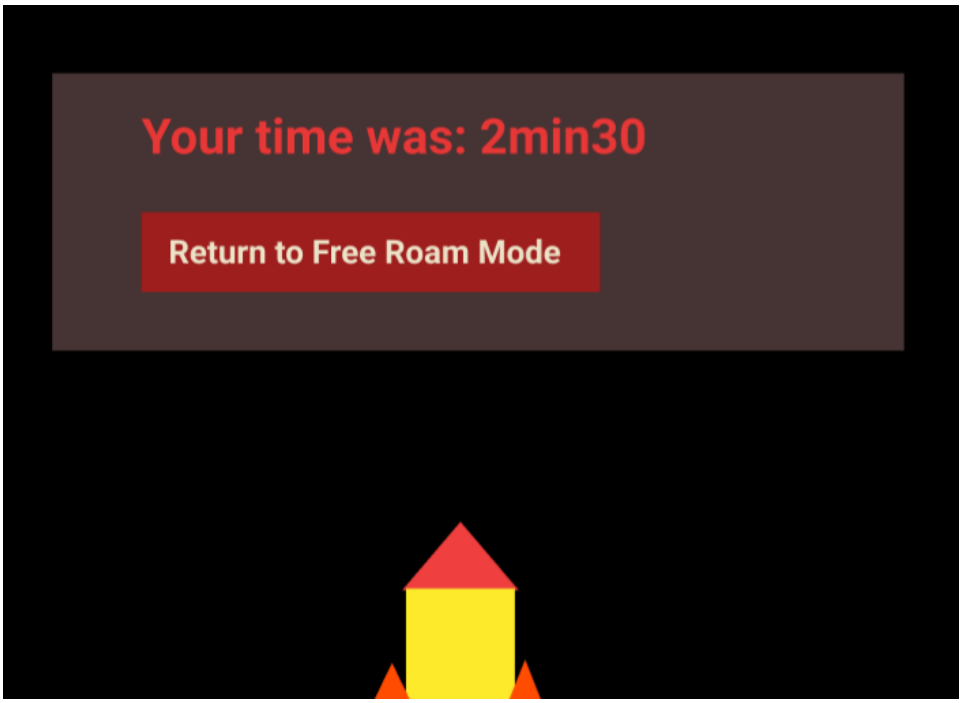
A timer is shown to show how much time has passed.



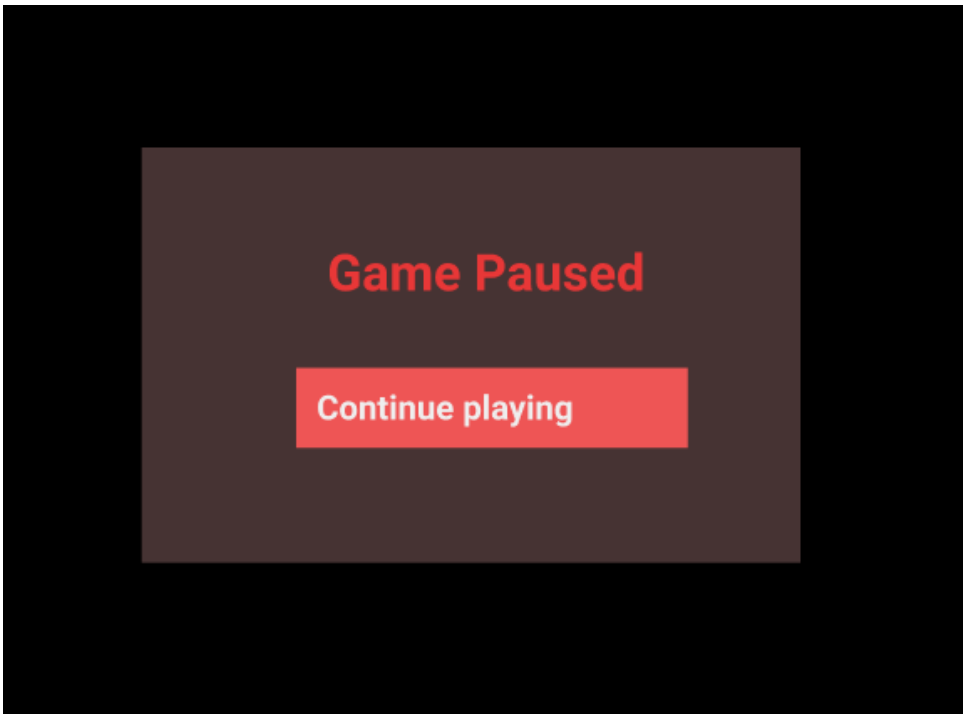
6a. End of race screen - multi player



6b. End of race screen - single player



7. Pause screen



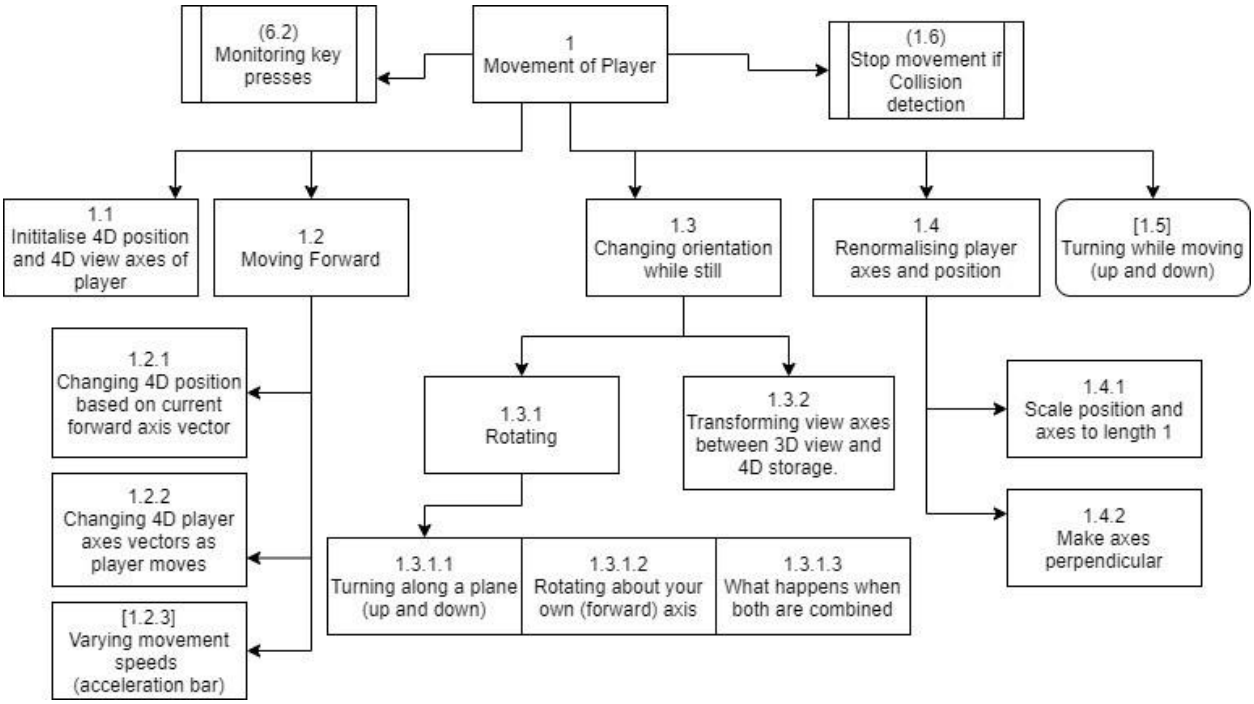
D3. Solution Planning

I will find what algorithms need to be implemented by splitting the problem into logical parts, and splitting these further still.

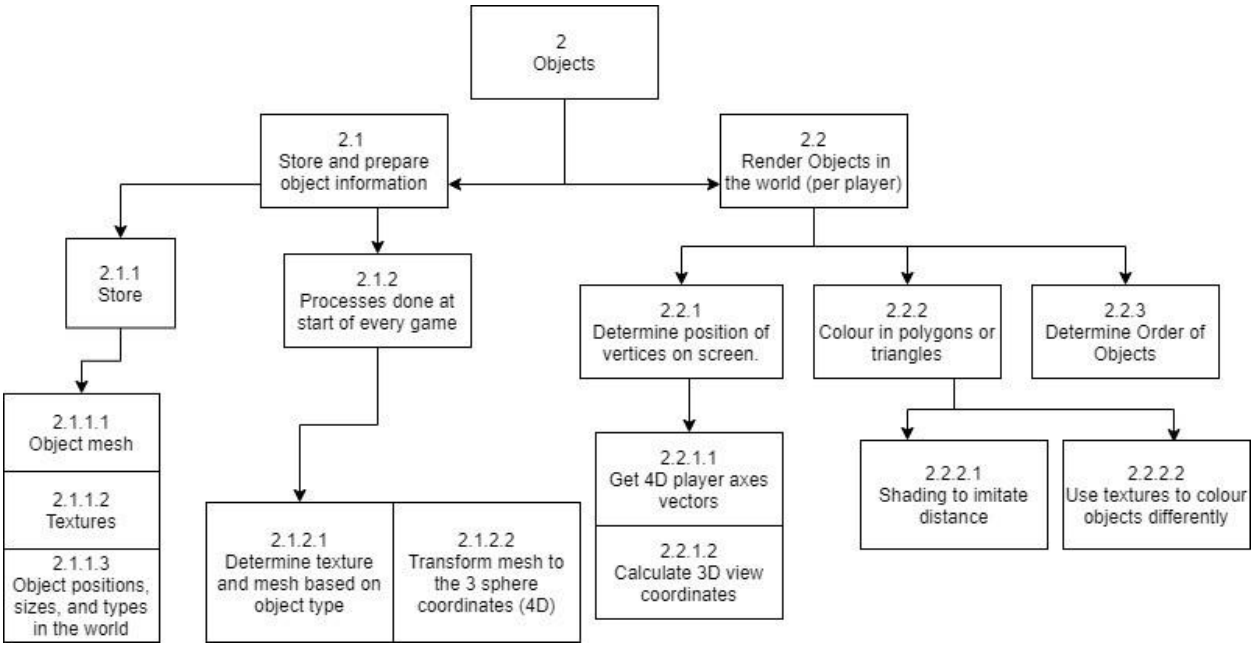
The main aspects of this program are:

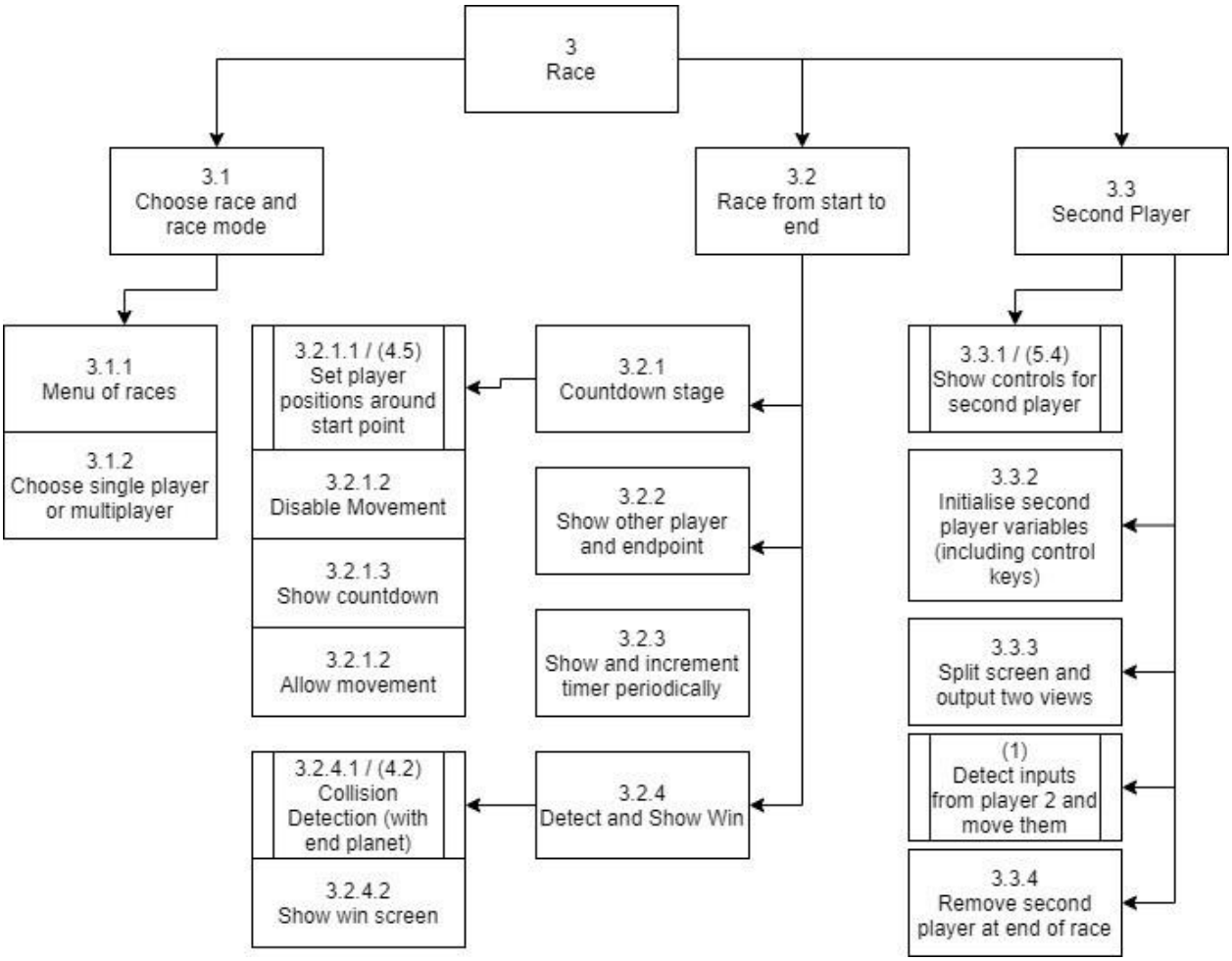
1. Moving the player
2. Rendering objects
3. Race game logic
4. Scanning for and preventing collisions (this will require a separate data structure so will be dealt with separately.
5. Output – as HTML forms and text

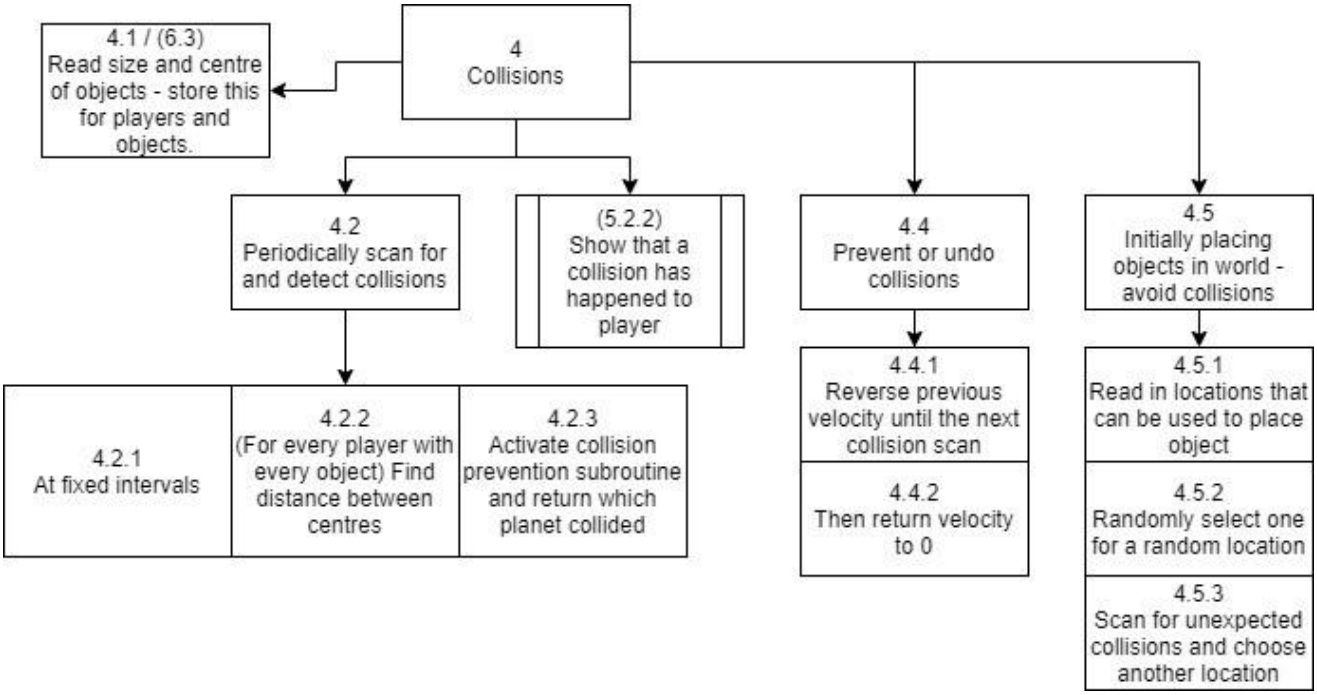
I have split these into further subtasks below:

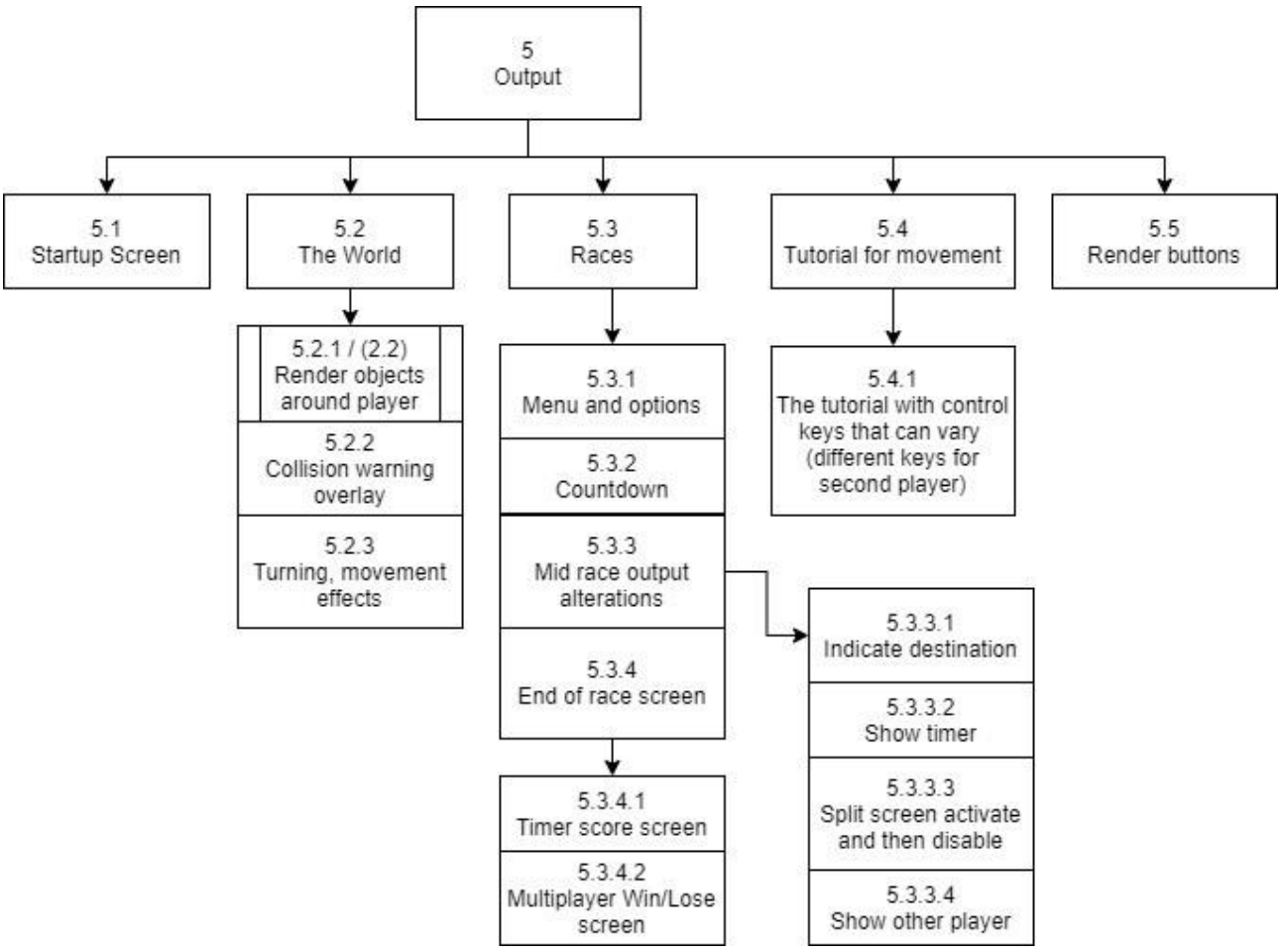


- Note about 1.4: As a player moves, the precision of the axes and coordinates is lost. To avoid weird effects, like gradual stretching of part of the screen, the axes and coordinates need to be kept perpendicular to each other. They also need to be kept magnitude 1. Section 1.4 will attempt to achieve this.


2
Objects
2.1
Store and prepare object information
2.2
Render Objects in the world (per player)
2.1.1
Store
2.1.2
Processes done at start of every game
2.2.1
Determine position of vertices on screen.
2.2.2
Colour in polygons or triangles
2.2.3
Determine Order of Objects
2.1.1.1
Object mesh
2.1.1.2
Textures
2.1.1.3
Object positions, sizes, and types in the world
2.1.2.1
Determine texture and mesh based on object type
2.1.2.2
Transform mesh to the 3 sphere coordinates (4D)
2.2.1.1
Get 4D player axes vectors
2.2.1.2
Calculate 3D view coordinates
2.2.2.1
Shading to imitate distance
2.2.2.2
Use textures to colour objects differently


3
Race
3.1
Choose race and race mode
3.2
Race from start to end
3.3
Second Player
3.1.1
Menu of races
3.1.2
Choose single player or multiplayer
3.2.1.1 / (4.5)
Set player positions around start point
3.2.1.2
Disable Movement
3.2.1.3
Show countdown
3.2.1.2
Allow movement
3.2.1
Countdown stage
3.2.2
Show other player and endpoint
3.2.3
Show and increment timer periodically
3.2.4.1 / (4.2)
Collision Detection (with end planet)
3.2.4.2
Show win screen
3.2.4
Detect and Show Win
3.3.1 / (5.4)
Show controls for second player
3.3.2
Initialise second player variables (including control keys)
3.3.3
Split screen and output two views
(1)
Detect inputs from player 2 and move them
3.3.4
Remove second player at end of race

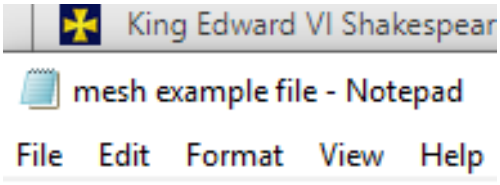

4
Collisions
4.1 / (6.3)
Read size and centre of objects - store this for players and objects.
4.2
Periodically scan for and detect collisions
(5.2.2)
Show that a collision has happened to player
4.4
Prevent or undo collisions
4.5
Initially placing objects in world - avoid collisions
4.2.1
At fixed intervals
4.2.2
(For every player with every object) Find distance between centres
4.2.3
Activate collision prevention subroutine and return which planet collided
4.4.1
Reverse previous velocity until the next collision scan
4.4.2
Then return velocity to 0
4.5.1
Read in locations that can be used to place object
4.5.2
Randomly select one for a random location
4.5.3
Scan for unexpected collisions and choose another location


5
Output
5.1
Startup Screen
5.2
The World
5.3
Races
5.4
Tutorial for movement
5.5
Render buttons
5.2.1 / (2.2)
Render objects around player
5.2.2
Collision warning overlay
5.2.3
Turning, movement effects
5.3.1
Menu and options
5.3.2
Countdown
5.3.3
Mid race output alterations
5.3.4
End of race screen
5.3.4.1
Timer score screen
5.3.4.2
Multiplayer Win/Lose screen
5.3.3.1
Indicate destination
5.3.3.2
Show timer
5.3.3.3
Split screen activate and then disable
5.3.3.4
Show other player
5.4.1
The tutorial with control keys that can vary (different keys for second player)

D4. Stored game data

File(s) or data stored	Usage	Format
Object mesh (one for every type of object)	Tells the program what the object looks like, using 3D coordinates and vertex indices. Stores the 3D model of an object. May also include the colours of vertices to colour objects.	Stored in a JSON or JS file.
Map file	Tells the program - the positions of objects in the world. - what mesh to use for every object - the size of every object	Stored in a JSON or JS file.
Race file	Tells the program - Where in the world players can be placed to avoid collisions with planets. - The different names and starting positions of races that can be chosen.	Stored in a JSON or JS file.
Starting player position and orientation	Used to place the player at an initial location on the map.	Hardcoded as a prototype. This may be put as part of the map file if needed, to avoid planet collisions.

Below: An example of a mesh file, with vertex positions, vertex colours, and the vertices used in triangles to make a solid shape.




```
planetMesh = {

 vertexPositions: [
   1.0,  1.0, 0.5,
   2.0,  1.2, 0.3,
   2.0, -0.4, 0.5,
   3.0, -0.2, 0.25,
  -1.0,  2.5, 5.9,
   0.5,  0.5, 0.9
 ],

 vertexColours: [
  1.0, 1.0, 0.0,
  0.5, 0.3, 0.0,
  0.1, 0.2, 0.0,
  0.5, 0.6, 0.7,
  0.1, 1.0, 1.0,
  0.8, 0.15, 0.0,
 ]

 trianglesIndexed: [
  0, 1, 2,
  0, 1, 4,
  0, 2, 4,
  0, 5, 1,
  2, 5, 1,
  3, 4, 5,
 ],

}
```

Below: The map file will use meshes by index, to prevent needing to write the filename of a mesh for every object that uses it. The center coordinates of planets are stored as 4D coordinates on a sphere.

```
map = {
 meshes: {
  0: 'planetMesh',
  1: 'rocketMesh',
  2: 'weirdPlanetMesh'
 }
 textures: {
  0: 'planetTexture',
  1: 'player1',
  2: 'player2',
 }
 objects: [
  {
   mesh: 0,
   texture: 0,
   center_coords: [0.05, 0.65, -0.2, 0.39],
   size: 0.5
  },
  {
   mesh: 0,
   texture: 0,
   center_coords: [0.5, 0.6, 0.7, -0.56],
   size: 0.4
  }
   ...
 ]
```

Below: the race file will have races with names, start planets (from map file), end planets, and possible player start positions and orientations, in p (position), v (forward axis), u (right axis), w (up axis) coordinates.

race f - Notepad

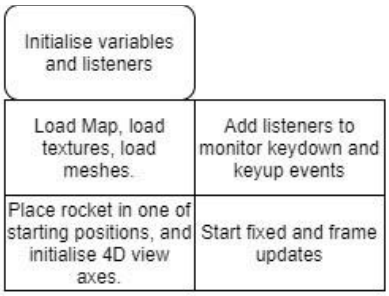
File Edit Format View Help

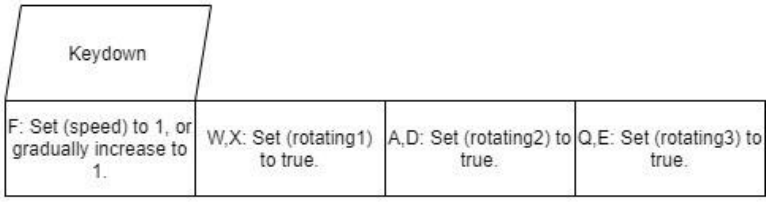
```
races = [
 {

  name: "The amazing mars race",

  start_planet_index: 0,

  end_planet_index: 1,

  start_positions_orientations: [
   [ 1.0,   0.0,  0.0,    0.0,
     0.0,   1.0,  0.0,    0.0,
     1.0,   0.0,  0.707,  0.707,
     0.0,   0.0, -0.707,  0.707
   ],
   [
     p ...
     v ...
     u ...
     w ...
   ]
  ]
 },

 {
  name: "Beyond the darkness",
  ...
 }

]
```

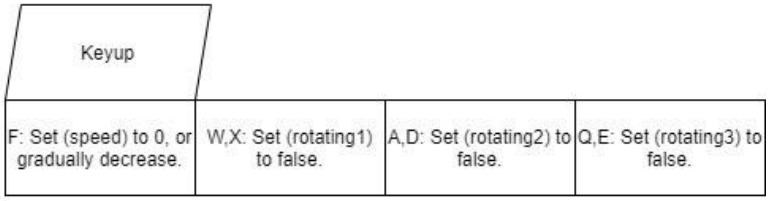
D5 Structure of algorithms

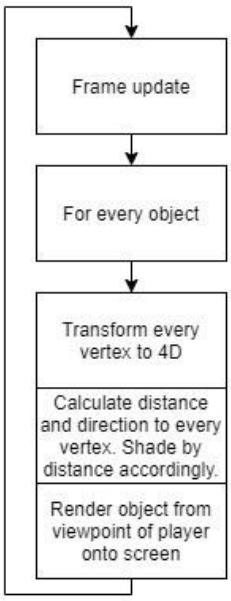
The following diagram (top left) shows all the tasks that need to be done at the start of the algorithm.

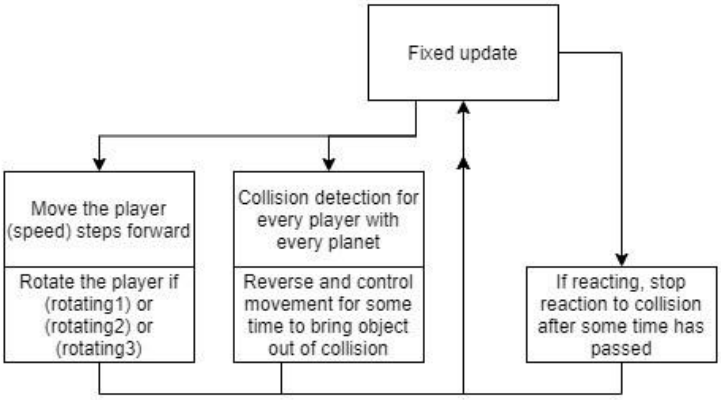
After this is done, looping and input processes take on the running of the program (all other diagrams describe these processes).





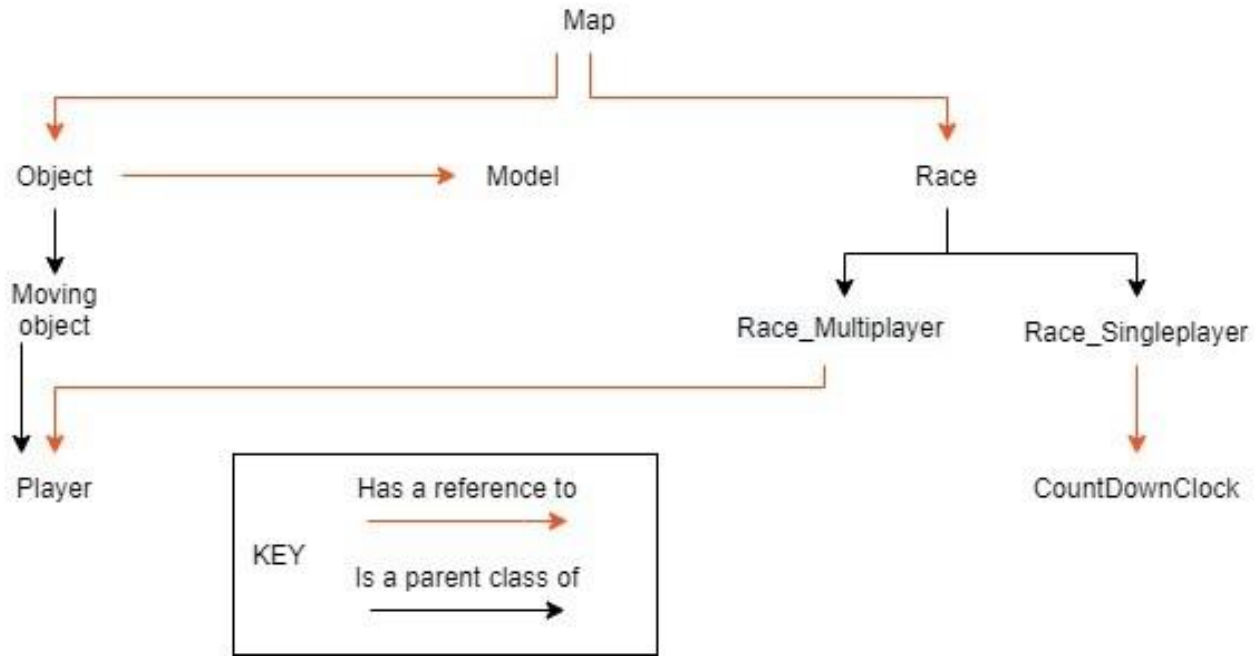






D6 Planned Classes

How the classes relate to each other:



Map

The Map class will be the program's main interface to controlling the game.

```
class Map:

  objects: [list of Object refs]

  races: [list of Race refs]
```

Model

The Model class is responsible for storing the loaded mesh data.

```
class Model:

  vertices: [list of vertices (4 coords per vertex) (all stored
contiguously)]

  triangles_indexed: [list of triangles (3 indexes to vertices list and 1
index to colours list)]

  colours: [list of colours (3 0-255 integers per colour)(all stored
contiguously)]
```

Object

A Player is a type of Object, as is a planet. The coordinates are stored in p. The axes are stored in v, u, w. The renderer uses the Object model_3D to render it.

```
class Object:

  ID: int

  p: [4 floats]

  v: [4 floats]

  u: [4 floats]

  w: [4 floats]

  size: 0.6

  model_3D: ref to Model
```

Moving Object

An object that can rotate or move. The rotation algorithm will be explained later.

```
class MovingObject inherits Object

  rotation_Rotating: false

  rotation_Angle: 0.0

  rotation_AngularSpeed: 1.0

  rotation_Mode: [0, 1]

  rotation_TimeUpdateFunction: (dt, self) => (update self)

  rotation_KeyChangeFunction: (event { type:'keyup'||'keydown', key: 'KEY
LETTER'}, self) => (update self)

  findActualCoordsAfterRotation: method => [[p], [v], [u], [w]]
```

Player

The Player class includes a keys list, which defines the controls that a player can use. This would allow two players to use the same keyboard.

```
class Player inherits MovingObject

  // Keys correspond to different movement types

  keys: ['F','W','S','A','D','Q','E']

// [Forward, Turn up, Turn down, Turn left, Turn right, Twist left, Twist
right ]
```

```
  // For multiplayer mode
  screen: {xPc: float, yPc: float, heightPc: float, widthPc: float},

  // The lower, the more world you can see.
  // May be useful to decrease for multiplayer mode
  pixelDistanceToScreen: float
```

Race

```
class Race
  name: String
  endObject: int ID of non moving object
  finishCloseness: float (represents distance to end planet to win)

  // Possible player start locations, to avoid collisions when placing
player.
  startCoords: [
[[p],[v],[u],[w]],
[[p],[v],[u],[w]],
[[p],[v],[u],[w]]
  ]
```

Race_Multiplayer

```
class Race_Multiplayer inherits Race
  players: [
    {
      object: ref to Player instance,
      won: undefined || false || true,
    },
    {
      object: ...
      won: ...
    }
```

```
  ]
```

Race_Singleplayer

```
class Race_Singleplayer inherits Race
  time: float
  tickTime: float (in ms)
  timeoutListener: ref to listener
  displayElementHTML: ref to HTML element
  CountDownClock: ref to CountDownClock
```

CountDownClock

Used to display 3 2 1 Go before the race starts

```
class CountDownClock:
  startAt: int
  endPhrase: "Go"
  displayElementHTML: ref to HTML element
```

D7 Main Algorithms

Some algorithms will be designed later. I have included a section at the end to describe these.

D7.1 Placing players

Used to place player(s) inititally or at the start of a race.

```
IN -> possibleStartCoords

IN -> players : [{object: Player instance, ...}, {object: Player instance,
...}]


if (possibleStartCoords.length  < players.length):
  ERROR (too few start places) -> This shouldn't happen
  STOP
endif


availableCoordIndexes = range(possibleStartCoords) //[0,1,2] for list of 3


for (pIndex from 0 to players.length):
  i = RandomInteger(0, availableCoordIndexes.length)
  chosenCoord = possibleStartCoords[availableCoordIndexes[i]]
  availableCoordIndexes.remove( index i )


  players[pIndex].object.p = chosenCoord[0]
  players[pIndex].object.v = chosenCoord[1]
  players[pIndex].object.u = chosenCoord[2]
  players[pIndex].object.w = chosenCoord[3]
Endfor
```

D7.2 GL Matrix Library

I will use the glMatrix library at https://glmatrix.net/ to help with matrix multiplication ad finding inverses of matrices. Using a library will:

- Save me programming time
- The algorithms from the library are optimised for time. Since I will use matrix multiplication for every vertex the program renders, time is critical to produce a smooth simulation experience.

The inverse of a matrix is (using the library):

```
inverse(mat3) = glMatrix.adjoint(mat3) / glMatrix.determinant(mat3)
-OR-
invert(mat3)
```

D7.3 Multiplying a vector (v) by a matrix (m) to get a vector (u)

```
Function multiplyMatVec(m, v):
u = new Vec3;
      u.x = m[0][0] * v.x + m[1][0] * v.y + m[2][0] * v.z;
      u.y = m[0][1] * v.x + m[1][1] * v.y + m[2][1] * v.z;
      u.z = m[0][2] * v.x + m[1][2] * v.y + m[2][2] * v.z;
      Return u;
End function.
```

D7.4 A useful function to rotate 2 of 4 axes (4D)

```
vector_rotation_4D = function (mode, angle, vectors4x4):
  cos = Math.cos(angle)
  sin = Math.sin(angle)
  // rotate axis A to axis B
  A = mode[0]
  B = mode[1]
  vectors4x4_new = deepCopy ( vectors4x4 )

  // ( Using vector linear multiplication and addition )
  vectors4x4_new[A] = vectors4x4[A]*cos  + vectors4x4[B]*sin
  vectors4x4_new[B] = -vectors4x4[A]*sin + vectors4x4[B]*cos
  // -- OR -- //
  // ( A more low level form )
  vectors4x4_new[A][0] = vectors4x4[A][0]*cos  + vectors4x4[B][0]*sin
  vectors4x4_new[A][1] = vectors4x4[A][1]*cos  + vectors4x4[B][1]*sin
  vectors4x4_new[A][2] = vectors4x4[A][2]*cos  + vectors4x4[B][2]*sin
```

```
  vectors4x4_new[A][3] = vectors4x4[A][3]*cos  + vectors4x4[B][3]*sin
  vectors4x4_new[B][0] = -vectors4x4[A][0]*sin + vectors4x4[B][0]*cos
  vectors4x4_new[B][1] = -vectors4x4[A][1]*sin + vectors4x4[B][1]*cos
  vectors4x4_new[B][2] = -vectors4x4[A][2]*sin + vectors4x4[B][2]*cos
  vectors4x4_new[B][3] = -vectors4x4[A][3]*sin + vectors4x4[B][3]*cos

  RETURN vectors4x4_new
end of method
```

D7.5 Normalising vectors

To keep the position and player view axes length 1.

Otherwise, as you move, the world will start to stretch, and the positioning accuracy may deteriorate. This should be done after every movement has been completed.

```
Function normalise_vec_4(v):
   Size = v.x*v.x + v.y*v.y + v.z*v.z + v.w*v.w;
   V.x = v.x / Size
   V.y = v.y / Size
   V.z = v.z / Size
   V.z = v.z / Size
   Return v;
End function
```

D7.6 Loading Algorithms

1. Load and interpret map and meshes into JavaScript.
2. Make array with all planet vertices. Each vertex will have 4 coordinates and (a pre-set colour OR coordinates corresponding to the texture image pixel coordinates)

Start program

Via HTML, load in models like this:

```
<script type='text/javascript' src='planetMesh.js'></script>
```

Read the json

```
Import planetMeshData from 'planetMesh.js'
Import rocketMeshData from 'rocketMesh.js'
```

… // other meshes

D7.7 Rendering Algorithm

It seems to me that mathematical notation may be easier to understand here. I have made a LATEX guide to explain the rendering algorithm. Here, I show how the maths would translate to pseudo-code.

Part 1. Putting 3D models into the 4D sphere coordinates.

Each planet will have 4 vectors - a center position vector, and 3 axes vectors, to determine its orientation.

$$c_{4D} = center$$

$$v_{4D},\ u_{4D},\ w_{4D} = orientation\ axes$$

The model/mesh has vertices.

Each vertex will have a 3D mesh position $m = \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix}$.

The 4D axes (v, u, w) correspond to the 3D axes (x, y, z) of the mesh model.

To determine the distance from the origin to the vertex:

$$D = \sqrt{m_x^2 + m_y^2 + m_z^2}$$

To determine the direction to go (d_{4D}) from the center position (c_{4D}), we do a matrix multiplication:

$$d = \begin{pmatrix} d_x \\ d_y \\ d_z \\ d_w \end{pmatrix} = \begin{pmatrix} v_x & u_x & w_x \\ v_y & u_y & w_y \\ v_z & u_z & w_z \\ v_w & u_w & w_w \end{pmatrix} \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix}$$

We then make sure the direction is a unit vector.

$$d_{4D} = unit(d)$$

To get the 4D position P_{4D} of each vertex, we simply do:

$$(P_{4D}) = (c_{4D})(\cos D) + (d_{4D})(\sin D)$$

To scale up a model, D will be multiplied by a scale factor (size).

Part 2. Getting the distance and direction between player and vertex

---

To render every vertex in the viewpoint, we have the following info:

$$v_{4D}, u_{4D}, w_{4D} = view\ axes\ of\ player$$

$$P1_{4D} = position\ of\ player$$

$$P2_{4D} = position\ of\ vertex$$

```
V = PlayerPosition[1]

U = PlayerPosition[2]

W = PlayerPosition[3]

P1 = PlayerPosition[0] // If the matrix format is 2D array

P2 = VertexPosition
```

We would need to get the distance between player and vertex (D), using the dot product:

$$cos(D) = (P1).(P2) = P1_1P2_1 + P1_2P2_2 + P1_3P2_3 + P1_4P2_4$$

```
cosD = P1[0]*P2[0] + P1[1]*P2[1] + …
```

… and calculate the 4D direction vector to go in to get to the vertex (d):

$$P2 = P1\ cos(D) + d\ sin(D)$$

$$d = \frac{P2 - P1\ cos(D)}{sin(D)} = \frac{P2 - P1\ cosD}{\sqrt{1 - cos(D)^2}}$$

```
d[0] = (P2[0] - P1[0]*cosD)/(sqrt(1-cosD*cosD))
d[1] = (P2[1] - P1[1]*cosD)/(sqrt(1-cosD*cosD))
….
d[3] = (….)/(...)
```

Part 3. Displaying the vertex on screen

To then display the vertex from the player's perspective, we need to use the player's view axes.

Here's how it works:

(d) (the direction in 4D) and ...

(v, u, w) (player's orientation axes in 4D) translate to...

(a, b, c) (the 3D coordinates to the point)

To get the 3D direction from the player, $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$, we have the equation:

$$\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix} = \begin{pmatrix} v_1 & u_1 & w_1 \\ v_2 & u_2 & w_2 \\ v_3 & u_3 & w_3 \\ v_4 & u_4 & w_4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

More simply...

$$\begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} v_1 & u_1 & w_1 \\ v_2 & u_2 & w_2 \\ v_3 & u_3 & w_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

So...

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} v_1 & u_1 & w_1 \\ v_2 & u_2 & w_2 \\ v_3 & u_3 & w_3 \end{pmatrix}^{-1} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

abc = inverse(pvuw.yzw)*d

To vary the field of view, each player would have a dist (pixelDistanceToScreen) value.

After we have (a,b,c) and D, we can position the point.

$$screen\ x = b\ \frac{dist}{a}$$

$$screen\ y = c\ \frac{dist}{a}$$

$$screen\ z\ (depth) = D = arcos(P1.P2)$$

screen z is used to order objects in the graphics pipeline **(2.2.3 on the decomposition diagram)**

Then, (screenx, screeny, screenz) are passed onto the fragment shader.

The complete algorithm

```
model
-> p, v, u, w x4
-> size
-> vertex
   -> m x3
```

player

-> p, v, u, w x4

-> pixelDistanceToScreen

|m| = sqrt(m.x^2 + m.y^2 + m.z^2)

D = Math.Pi * |m| * size

(d) = m.x*(v) + m.y*(u) + m.z*(w) = (v u w)*(m)

unit(d) = (d) / |d|

(P) x4 = (p)cos(D) + (d)sin(D)

cos(D) = (player.p).(mesh.p) = player.p[0]*mesh.p[0] + player.p[1]*mesh.p[1] + …

D = Math.arcos(cos(D)) from 0 to Pi.

d = (mesh.p - (player.p)(cos(D))) / (sin(D))

glMatrix invert (v u w) cropped by 4th (w) coordinate.

(Use adjoint and determinant or the invert functionality)

Multiply by distance cropped by the 4th (w) coordinate. (using glMatrix)

To get (a, b, c) view direction.

Check that a(v.w) + b(u.w) + c(w.w) = (d.w) to a good degree of accuracy.

screen.x = b/a * pixelDistanceToScreen

screen.y = c/a * pixelDistanceToScreen

screen.z = D

D7.8 Limiting Screen size for multiplayer

(to be designed)

D7.9 Colouring the triangles

(To be designed)

The fragment shader will colour all vertices white to start with, on a black background.

I will design and implement the shading and colours later.

D7.10 Moving the rocket

At key down events, the key pressed will be analysed, the key character saved, and the movement applied:

- Q or E – twist left or right – **1.3**
- A or D – turn left or right – **1.3**
- W or X/S – turn up or down - **1.3**
- F – go forward – **1.2**

Each sort of movement will be separate – there will be no combinations.

Two methods exist for rotating an object on a sphere.

A. Update position using a rotation matrix at every fixed update.
B. Update angle of rotation at fixed updates. Use angle in rotation matrix to render.

Here I will go for **method B** because:

- It is easier to add to the angle rather than to apply a matrix every update, and it is more accurate.
- Matrix multiplication is more optimised to do in the GPU, which can be done with method B, but not method A. This is because GPU cannot return the new position to the program – it is just used for rendering.

So here it is:

1. When a new movement begins, the initial coordinates and axes of the player will be recorded (**playerPosition** matrix). The type of movement will determine the rotation matrix used.
2. A rotation matrix based on the degree of rotation / translation will be applied to the saved coordinates at every render.
3. However, the saved coordinates and axes of the player will remain the same till the movement ends, at which point the coordinates will update and save.

The full algorithm:

```
(Player)(MovingObject) method rotation_KeyChangeFunction(event {
type:'keyup'||'keydown', key: 'KEY LETTER'})


  key = event.key
  player = self // To make the pseudo code clear

  // Translate key letter -> mode index -> mode
  // [Forward, Turn up, Turn down, Turn left, Turn right, Twist left, Twist right ]
  modes = [[0,1], [1,3], [3,1], [2,1], [1,2], [2,3], [3,2]]

  if key in player.keys:
    // Get index
    modeIndex = player.keys.index(key)

    // Get mode
    mode = modes[modeIndex]

    if event.type == 'keydown':
      // Save mode for player
      player.rotation_Mode = mode
      player.rotation_Rotating = true
      player.rotationAngularSpeed = 1.0

    else if event.type == 'keyup':
      // Stop player rotating if they were
      if player.rotation_Mode == mode:
        player.rotation_Rotating = false
        player.rotation_AngularSpeed = 0.0
        // Set new initial position and orientation
        player.coords = player.findActualCoordsAfterRotation()
player.rotation_Angle = 0.0
      endif

    endif
```

```
    RETURN true  // The key was used to move the player
  endif
  RETURN false   // The key may be used elsewhere (like ESC)
end of method
```

```
// vector_rotation_4D has been defined earlier
(Player)(MovingObject) method findActualCoordsAfterRotation:
  player = self

  newCoords = vector_rotation_4D( player.rotation_Mode, player.rotation_Angle,
player.coords )
  newNormalisedCoords = normalise(newCoords)
  RETURN newNormalisedCoords;

end of method
```

```
// At every fixed update, the movement will be applied by increasing linear
position/orientation angle.
(Player)(MovingObject) method rotation_TimeUpdateFunction (dt)
  player = self

  if player.rotation_Rotating == true:
    player.rotation_Angle += dt * player.rotation_AngularSpeed

    collision = CollisionDetection.checkCollision(player.ID)
    if collision == true:
      // rebound effect by reversing speed
      player.rotation_AngularSpeed = -player.rotation_AngularSpeed
      // maybe some infinite collision prevention later
    endif
  end if
end of method
```

D7.11 Collision Detection

If distance between player.p and object.p is more than player.size and object.size, a collision has occurred.

The way to calculate distance between 2 4D coordinates on a 3sphere has bee described earlier, in Rendering.

Algorithms to think about after first development stage

As the problem is still not well understood, and since some issues may or may not arise, I will program he base functionality first, and see if anything needs to be taken care of.

This repetitive improvement of prototypes **(RAD)** will allow me to move forward with the program and may raise issues with the design early on.

The things I may need to program may be...

A. An algorithm to output axes and coordinates of the player, so that I can use these to place planets where I would like. Doing this from inside the world may be easier than calculating the coordinates needed by hand. I plan to only use this in development.
B. Making axes perpendicular.
C. Tutorial for any player
D. Splitting screens
E. Showing the countdown
F. Race timer

Black Box Functionality Testing

Test	Which part	Purpose
If you start the program, there is a starting screen. You can press a button to start the simulation.	Start screen	
Once in the simulation, press ESC. A) The pause screen should appear B) Once you re-enter the simulation, the rocket is no longer moving.	Pause functionality	B) Prevent any potential bugs by resetting the movement to original variables.
In two player mode, if ESC is pressed, both players' games are paused and the start screen appears.	Pause functionality	So that the players can't get an advantage over one another by pressing pause on the others' screen.
Pressing x moves player y. X, y - F, forwards - W, turn up - S, turn down - A, turn left - D, turn right - Q, spin ACW - E, spin CW This can be seen by planets moving the other way.	Movement	Player can move.
Pressing, then releasing x for all of the above stops the motion.	Movement	So that the player can stop.
Press two buttons from the above. Only the first pressed motion should result. The planets must move around the same way as if only the first key was pressed. - A then W - Q then F - F then Q - S then A then F - A then D	Movement	Check that no movements can act simultaneously.

Press A, release A. Press F, release F. This should turn the rocket left, then move it forward.	Movement	Check that doing one movement does not stop consequent movements.
Point camera at a planet far away and press F to move to it. This should either shrink, then enlarge the planet, or just enlarge the planet.	Rendering Three Sphere Effect	
Point camera at an empty piece of space. Remember the starting image. Press and hold F to move forwards. You should reach your starting point after some time.	Three sphere effect	Check that movement logic works correctly
Planets should appear similar to their 3D models.	Rendering	Just check whether this is the case.
There should be a rocket bottom center of screen to represent player.	Rendering	Player knows where they are.
Approach planet head on. You should stop before the planet and not see inside it. You should move back unless you release or press a key.	Rendering	
Approach a planet from the side. The rocket should reverse movement when it reaches the planet.	Collision Detection	
After collision, quickly turn left till you face away from the planet. Does the user seem to be seeing the inside of the planet?	Collision Detection	Find out if user can look inside planet.
Place a planet close to the start position of the player. Is the player placed inside the planet? This should not happen.	Collision Detection Start of Free Roam	Make sure a player can't be spawned inside a planet.

White box Mathematical Testing

To be designed.

Bit by bit, and line by line

The sphere emerges...

Max Silin

Development Plan

I will follow an iterative approach to development, programming, testing and evaluating modules one by one, before going to program the next modules. I will try to do this in the following order of stages:

P1. Storage and Retrieval of Objects (2.1) and WebGL preparation.

P2. Rendering objects in 4D sphere mode (2.2)

P3. Moving the player through the world (1)

P4. Detecting and preventing collisions (4)

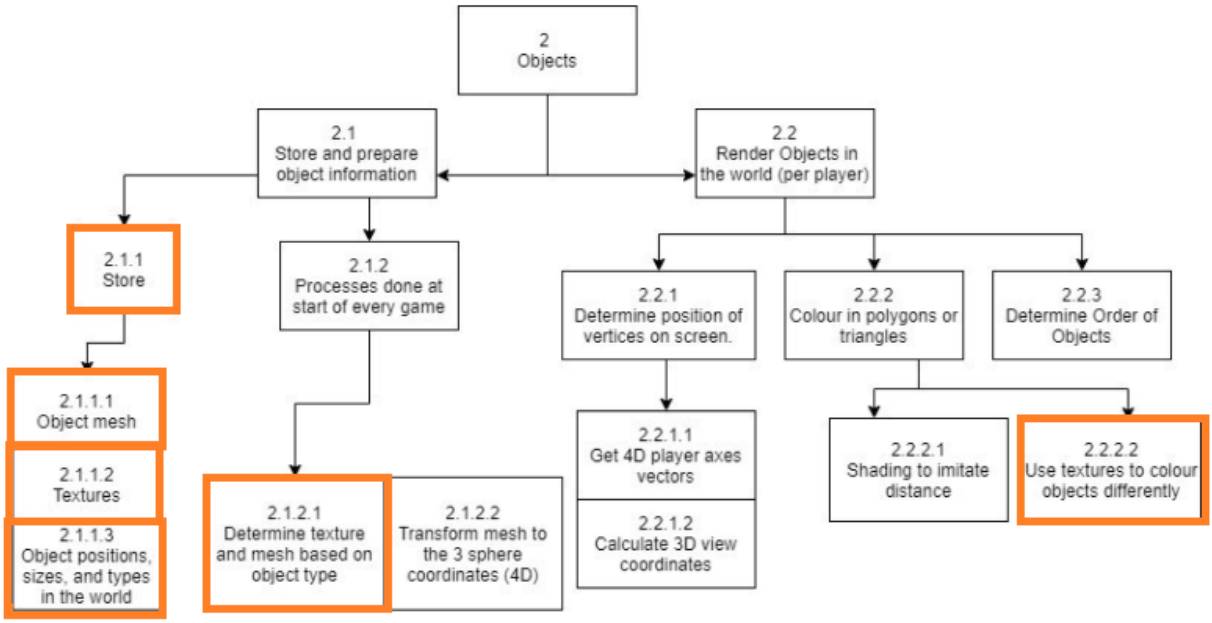
I will program the following if I have time:

P5. Race against timer (3.2 and 5.3.3.2 and 5.3.4.1) and race selection (3.1 or 5.5)

P6. User Experience like Tutorial (5.4) and Startup Screen (5.1)

P1. Iteration 1 - Storage and Retrieval of Objects and WebGL preparation.

Key: **To program now** **Changed**

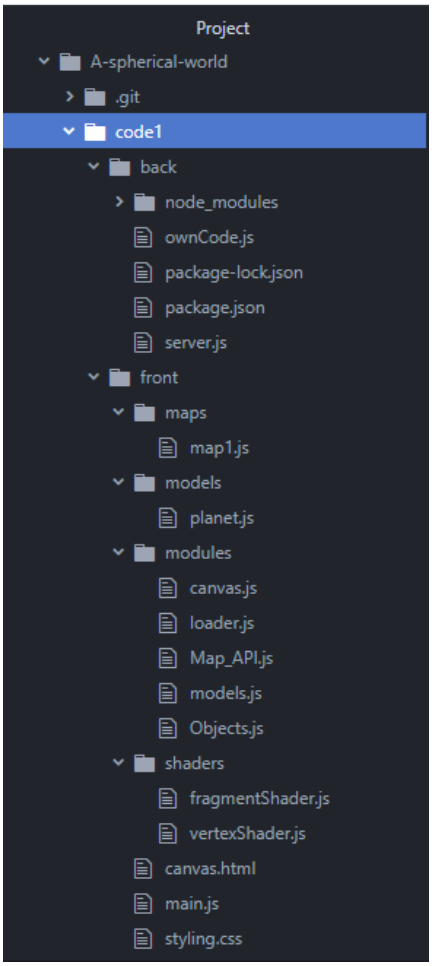


This section aims to tackle 2.1 - Store and prepare object information.

This section also aims to render objects on the screen in flat perspective.

P1.1 Starting Structure

P1.1.1 Folder layout



The “back” folder

- Stores node.js server to serve all the content to the webpage.

The “front” folder

- “maps” stores each map of each world in a separate file.
- “modules” stores all my front end javascript code to manage the application
- “shaders” stores the WebGL shaders I may need to use
- “canvas.html” is the webpage that has my WebGL window. HTML may be used for inputs such as buttons and outputs such as error boxes.
- “main.js” is an entry script that calls other modules
- “styling.css” stores CSS needed for document layout and input and output styling.

P1.1.2 Back End

Putting JavaScript code into different files means you need a backend server to serve these files.

I am using npm (Node Package Manager) to install the “express” and “nodemon” packages (run “npm install” on a new machine with this file “package.json” in place).

package.json

```
{
  "name": "back",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "game": "nodemon server.js"
  },
  "author": "Max S",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
    "nodemon": "^2.0.13"
  }
}
```

- Nodemon reloads the server script every time there's a change in the code, so I do not have to restart the server manually every time I update the code.

- "npm run start" runs the server as production ready.
- "npm run game" runs the server in development (refresh) mode.

```
C:\Users\Maxicl\code\nice\A-spherical-world\code1\back>npm run game

> back@1.0.0 game C:\Users\Maxicl\code\nice\A-spherical-world\code1\back
> nodemon server.js

[nodemon] 2.0.13
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Listening on port 5000.
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Listening on port 5000.
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Listening on port 5000.
```

```
Terminate batch job (Y/N)? y

C:\Users\Maxicl\code\nice\A-spherical-world\code1\back>npm run start

> back@1.0.0 start C:\Users\Maxicl\code\nice\A-spherical-world\code1\back
> node server.js

Listening on port 5000.
```

- Express allows you to create a back-end server quickly. In “server.js”, I am only serving scripts and static assets from the “front” folder.

server.js

```
1  const express = require('express')
2  const app = express()
3  app.use(express.json())
4
5  app.use(express.static('../front'))
6
7  const PORT = 5000;
8
9  app.listen(PORT, ()=>{
10   console.log(`Listening on port ${PORT}.`)
11 })
12
```

The application can now be accessed on localhost:5000/canvas.html

P1.1.3 Front End

“canvas.html" imports all javascript modules.

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>

    <title> My Spherical World </title>
    <meta charset="utf-8">
    <link rel='stylesheet' href='styling.css'/>

    <script src='./modules/canvas.js' type="module"></script>
    <script src='./modules/loader.js' type="module"></script>
    <script src='./modules/Map_API.js' type="module"></script>
    <script src='./modules/models.js' type="module"></script>
    <script src='./modules/Objects.js' type="module"></script>
    <script src='./models/planet.js' type="module"></script>
    <script src='./maps/map1.js' type="module"></script>
    <script src='./shaders/fragmentShader.js' type="module"></script>
    <script src='./shaders/vertexShader.js' type="module"></script>

    <script src='./main.js' type='module'></script>

  </head>
  <body>
    <div id='WebGL_Fallback' hidden='true' class='info_yellow'>
      Web GL is not supported by you or your browser. Please give it some support. A kind word or two will make him feel happier.
    </div>
    <canvas id='canvas'></canvas>
  </body>
</html>

```


styling.css

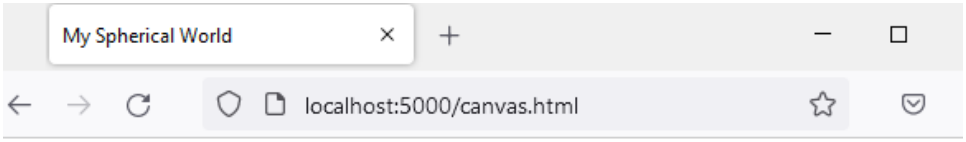
```css

canvas#canvas{
  height: 80%;
  width: 80%;
  margin: 5% 10%;
}

.info_yellow{
  width: 80%;
  margin: auto 8%;
  padding: 2%;
  background: #eeee77;
  border: solid 1px #dddd66;
  border-radius: 5px;
}

```

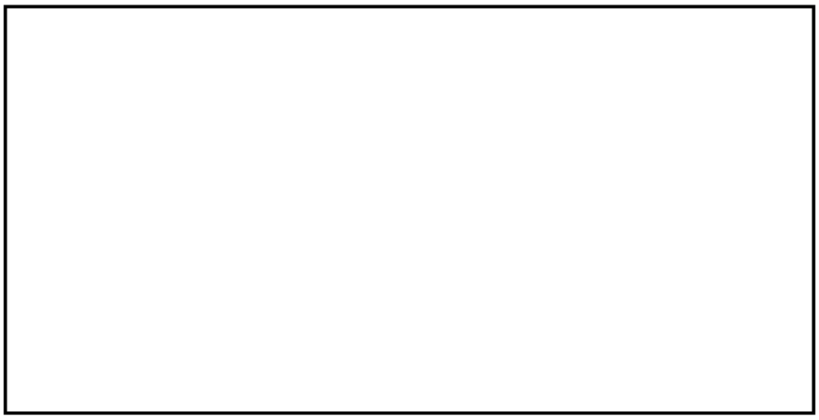
main.js

```
import { Canvas } from './modules/canvas.js';
import loader from './modules/loader.js';

let myCanvas;

window.addEventListener("load", function onLoad (evt) {
  "use strict"

  // Cleaning after ourselves. The event handler removes
  // itself, because it only needs to run once.
  window.removeEventListener(evt.type, onLoad, false);
  myCanvas = new Canvas('canvas');
  if (!myCanvas.WebGLAvailable){
    // Show fallback message
    let fallbackElement = document.getElementById('WebGL_Fallback');
    fallbackElement.hidden = false;
  }

  const runningGame = loader(myCanvas);
});
```

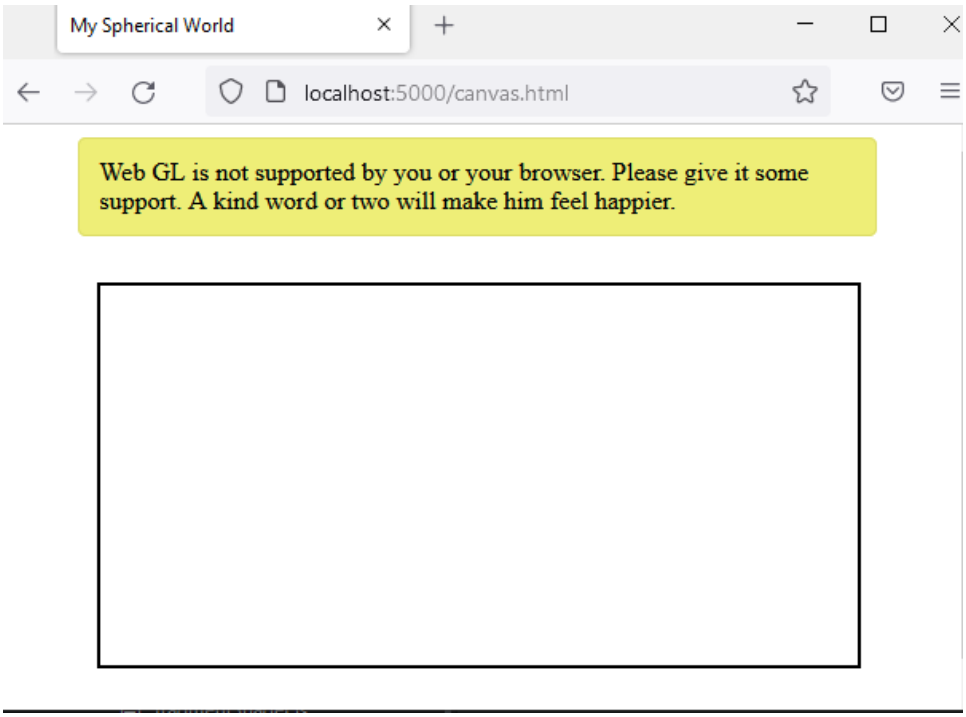
It sees if WebGL is available and reports an info box if the user's browser does not use WebGL.

Available.





Not available.



P1.2 Stored data

This part aims to tackle 2.1.1.1 (storing object meshes) and 2.1.1.3 (storing world maps), but not 2.1.1.2 (storing textures) as this is a UI enhancement and does not contribute to the functionality of the program, which should be a priority.

P1.2.1 Starting data

map1.js

Stores the positions of objects on the map, and the meshes associated with them.

canvas.html | canvas.js | *map1.js*

```
1  import Model_planet from '../models/planet.js';
2  const map = {
3    meshes: {
4      0: Model_planet,
5    },
6    objects: [
7      {
8        meshIndex: 0,
9        pvuw: [
10         [0.0, 1.0, 0.0, 0.0],
11         [1.0, 0.0, 0.0, 0.0],
12         [0.0, 0.0, 0.0, 1.0],
13         [0.0, 0.0, 1.0, 0.0]
14       ],
15       size: 0.5,
16     }
17   ]
18 }
19
20 export default map;
21
```

planet.js.

Stores planet mesh.

Vertices: The 3D position of every vertex

trianglesIndexed: The array index of each vertex in a triangle is stored in this array (3 integers per triangle)

Contains some random data as a prototype right now.

planet.js

```
const planet_vertices = [
  0.0, 0.1, 0.2, 0.5,
  0.1, 0.2, 0.1, 0.6,
  0.3, 0.0, -0.2, 0.7,
  0.2, -0.2, -0.3, 0.8,
  0.1, 0.1, 0.1, 0.9,
]

const planet_trianglesIndexed = [
  1, 2, 3,
  2, 3, 4,
  1, 2, 4,
  4, 3, 1
]

const Model_planet = { vertices: planet_vertices, trianglesIndexed: planet_trianglesIndexed };

export default Model_planet;
```

Map_API.js

A set of useful functions for handling map objects.

Map_API.js

```
function get_Object_mesh_index(objectIndex, map){
  return map.objects[objectIndex].meshIndex;
}
function get_Object_size(objectIndex, map){
  return map.objects[objectIndex].size;
}
function get_Object_pvuw(objectIndex, map){
  return map.objects[objectIndex].pvuw;
}

function get_Object(objectIndex, map){
  return map.objects[objectIndex];
}

const Map_API = {get_Object, get_Object_mesh_index, get_Object_pvuw, get_Object_size};

export default Map_API;
```

P1.3 Attaching WebGL

This part attempts to put the data associated with each object on the map (see P1.2.1) into WebGL buffers and render it with its 3D coordinates as a prototype.

P1.3.1 Canvas.js methods

Canvas class is used to handle WebGL communication jobs, like adding vertex data to GPU buffers, adding uniforms (data invariant for each object), adding rendering programs (shaders), and drawing the objects.

constructor() and getContext()

- Check if WebGL is available. If not, return false.
- Inititalise class attributes
- Makes screen background
- Uses shaders from /shaders files to generate a WebGL program

canvas.js

```
10    constructor(canvas_HTML_ID){
11      this.canvas = document.getElementById(canvas_HTML_ID);
12      this.WebGLAvailable = this.getContext();
13      if (!this.WebGLAvailable){
14        return;
15      }
16      // Stores references to meshes that were input into the GPU
17      // form of list item: {meshID: __, vertexBufferRef: ___ , indexBufferRef: ____ }
18      this.meshBuffers = [];
19      // Ref to shader program (to be made)
20      this.program;
21      // Locations of uniforms and attributes stored here.
22      this.attributeReferences = {};
23      this.uniformReferences = {};
24
25      // Colour screen black
26      this.clearColour();
27      // Attach shaders and make the WebGL program
28      this.createProgram();
29    }
30    getContext(){
31      // Sets up this.gl or returns false if unable to.
32      // Try to access WEB GL
33      this.gl = canvas.getContext("webgl");
34      // Report back if you can't get webgl
35      if (!this.gl) {
36        return false;
37      }
38      // Set window dimensions
39      this.gl.viewport(0, 0, this.gl.drawingBufferWidth, this.gl.drawingBufferHeight);
40      return true;
41    }
42    clearColour(){
43      this.gl.clearColor(0.0,0.0,0.0,0.0);
44      this.gl.clear(this.gl.COLOR_BUFFER_BIT)
45    }
```

compileShader() and createProgram()

- Tries to compile GLSL code provided in /shaders
- Links into a WebGL program and saves this program reference in this.program

```
import {vertexShader as vertexShaderSource} from '../shaders/vertexShader1.js';
import {fragmentShader as fragmentShaderSource} from '../shaders/fragmentShader1.js';
```

```
60    // shaderSource is a string of GLSL code
61    // shaderType is gl.VERTEX_SHADER or gl.FRAGMENT_SHADER
62    compileShader(shaderSource, shaderType) {
63      // Get rendering context
64      let gl = this.gl;
65      // Create the shader object
66      var shader = gl.createShader(shaderType);
67      // Set the shader source code.
68      gl.shaderSource(shader, shaderSource);
69      // Compile the shader
70      gl.compileShader(shader);
71      // Check if it compiled
72      var success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
73      if (!success) {
74        // Something went wrong during compilation; get the error
75        console.log( "could not compile shader:" + gl.getShaderInfoLog(shader));
76        return false;
77      }
78      return shader;
79    }
```

```
90    createProgram() {
91      // create a program.
92      var program = this.gl.createProgram();
93      // compile and attach the shaders.
94      // (SOURCES IMPORTED FROM /shaders)
95      // VERTEX S.
96      var vertexShader = this.compileShader(vertexShaderSource, this.gl.VERTEX_SHADER);
97      if (vertexShader === false){
98        throw 'Vertex shader not compiled.';
99      }
100     this.gl.attachShader(program, vertexShader);
101     // FRAGMENT S.
102     var fragmentShader = this.compileShader(fragmentShaderSource, this.gl.FRAGMENT_SHADER);
103     if (fragmentShader === false){
104       throw 'Vertex shader not compiled.';
105     }
106     this.gl.attachShader(program, fragmentShader);
107     // link the program.
108     this.gl.linkProgram(program);
109     // Check if it linked.
110     var success = this.gl.getProgramParameter(program, this.gl.LINK_STATUS);
111     if (!success) {
112         // something went wrong with the link
113         throw ("program failed to link:" + this.gl.getProgramInfoLog (program));
114     }
115     this.program = program;
116   };
117   useProgram(){
118     this.gl.useProgram(this.program)
```

Add_Indexed_Mesh() and findMesh()

- Saves mesh data (vertices and indices) in WebGL buffers.
- Saves these buffer references in this.meshBuffers

```
158     add_Indexed_Mesh(ID, vertices, trianglesIndexed){
159       let gl = this.gl;
160       // Create and bind buffer, then add vertex coordinates.
161       const vertexBuffer = gl.createBuffer();
162       gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
163       gl.bufferData(
164         gl.ARRAY_BUFFER,
165         new Float32Array(vertices),
166         gl.STATIC_DRAW
167       )
168       // Create and bind index buffer.
169       const indexBuffer = gl.createBuffer();
170       gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
171       // Fill the current element array buffer with data.
172       gl.bufferData(
173           gl.ELEMENT_ARRAY_BUFFER,
174           new Uint16Array(trianglesIndexed),
175           gl.STATIC_DRAW
176       );
177
178       this.meshBuffers.push({
179         meshID: ID,
180         vertexBufferRef: vertexBuffer ,
181         indexBufferRef: indexBuffer ,
182         metadata: {
183           numPoints: trianglesIndexed.length
184         }
185       });
186     }
```

```
200     findMesh(meshID){
201       // Find buffers to use
202       // using linear search
203       let found = false;
204       let meshIndex = 0;
205       let the_Mesh = {};
206       while ((found !== true) && (meshIndex < this.meshBuffers.length)){
207         the_Mesh = this.meshBuffers[meshIndex];
208         if (the_Mesh.meshID === meshID.toString()){
209           found = true
210           return the_Mesh;
211         }
212         meshIndex = meshIndex + 1;
213       }
214       if (found !== true){
215         console.log("Mesh was not found. Hey, programmer! Seems like there's a little mistake!")
216         return undefined;
217       }
218     }
```

add_AttributeReference() and add_UniformReference()

- Gets the locations of uniforms and attributes in the WebGL program, for the JS program to use.
- See how this is used in loader.js (further)

```
126     add_AttributeReference(KeyToVarDict){
127       // in { key : Shader Variable Name String } dictionary
128       // out { key : Attribute Location Reference } dictionary
129       // saves in this.attributeReferences
130
131       // For every attribute
132       let keys = Object.keys(KeyToVarDict);
133       for (let i=0; i < keys.length; i++){
134         let key = keys[i];
135         let ShaderAttributeName = KeyToVarDict[key];
136         // Add the location of the attribute
137         this.attributeReferences[key] = this.gl.getAttribLocation(this.program, ShaderAttributeName);
138       }
139     }
140     add_UniformReference(KeyToVarDict){
141       // in { key : Shader Variable Name String } dictionary
142       // out { key : Uniform Location Reference } dictionary
143       // saves in this.uniformReferences
144
145       // For every uniform
146       let keys = Object.keys(KeyToVarDict);
147       for (let i=0; i < keys.length; i++){
148         let key = keys[i];
149         let ShaderUniformName = KeyToVarDict[key];
150         // Add the location of the uniform
151         this.uniformReferences[key] = this.gl.getUniformLocation(this.program, ShaderUniformName);
152       }
```

DrawObject()

- Gives all the previously stored references, as well as the WebGL instance to a function "beforeDraw". This function can then use all this accumulated data to set up WebGL to draw the desired model with the desired uniforms.
- Then it tells WebGL to draw the requested object.

```
223     drawObject(beforeDraw, numElementsToDraw){
224
225       // Call the requested function to link correct buffers
226       // And to pass in uniform data
227       beforeDraw(this);
228
229       // Render the object to the screen
230       let primitiveType = this.gl.TRIANGLES;
231       let offset = 0;
232       let indexType = this.gl.UNSIGNED_SHORT;
233       this.gl.drawElements(primitiveType, numElementsToDraw, indexType, offset);
234
235       return true;
236     }
```

Now, you can see how these methods are used in action in loader.js .

P1.3.2 Rendering Objects from Map

I have decided to first develop a prototype to test if all my algorithms developed so far work.

Therefore, I will show the planet meshes as if they are just in 2D space, for now.

This requires me to add these properties to each object:

- xy: the xy coordinates of the object in the world. These will be replaced by pvuw 4D coordinates.
- colour: the colour of each object, in rgba to distinguish objects while testing.

The resulting map1.js looks like this for two objects:

```
  meshes: {
    0: Model_planet,
  },
  objects: [
    {
      meshIndex: 0,
      pvuw: [
        0.707, -0.707, 0.0, 0.0,
        0.707, 0.707, 0.0, 0.0,
        0.0, 0.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 0.0
      ],
      xy: [0.0, -0.5],
      size: 0.8,
      colour: [1.0, 0.0, 0.0, 1.0]
    },
    {
      meshIndex: 0,
      pvuw: [
        -0.84, 0.5, 0.0, 0.0,
        0.0, 0.0, 0.0, 1.0,
        -0.5, -0.84, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0
      ],
      xy: [0.0, 0.5],
      size: 0.56,
      colour: [1.0, 1.0, 0.0, 1.0]
    }
  ]
}
```

Size and xy are be passed to the vertex shader to determine positions:

```
2  export const vertexShader = `
3
4    attribute vec3 coordinates;
5    uniform float object_size;
6    uniform vec2 object_xy;
7
8    void main(){
9      vec3 coords = coordinates * object_size;
10     gl_Position = vec4(
11       coords.x + object_xy.x,
12       coords.y + object_xy.y,
13       coords.z,
       1.0);
     }
16
17   `
```

Colour is passed to the fragment shader.

Lines 3-7 tell the browser running the simulation which precision to use when handling floats. If high precision is available, the browser will use this.

loader.js | *fragmentShader1.js*

```
1
2  export const fragmentShader = `
3    #ifdef GL_FRAGMENT_PRECISION_HIGH
4      precision highp float;
5    #else
6      precision mediump float;
7    #endif
8
9    uniform vec4 object_colour;
10
11   void main(){
12     gl_FragColor = object_colour;
13   }
```

To get the vertex data and uniforms for each object into the shaders, a series of steps must be done. This is the job of loader.js .

“loader.js”

```
import Map_API from '../modules/Map_API.js';
import {Object4D} from './Objects.js';

import map from '../maps/map1.js';

function loader(Canvas){

  // Save meshes to renderer Canvas object.
  // For every mesh in the mesh dictionary
  let meshIndexes = Object.keys(map.meshes);
  for (let i=0; i < meshIndexes.length; i++){
    let index = meshIndexes[i];
    let mesh = map.meshes[index];
    Canvas.add_Indexed_Mesh(index, mesh.vertices, mesh.trianglesIndexed);
  }
  // Set up attributes and uniforms references
  Canvas.add_AttributeReference({
    'coordinates': 'coordinates'
  })
  Canvas.add_UniformReference({
    'object_xy': 'object_xy',
    'object_size':'object_size',
    'object_colour': 'object_colour',
  })
  let X_displacement = 0.2
```

```
// Turn map objects into list of those ready for processing
  let amount_of_objects = Map_API.get_num_Objects(map);
  let processed_objects = []
  for (let i=0; i<amount_of_objects; i++){
    // Find mesh for each object
    let meshIndex = Map_API.get_Object_mesh_index(i, map)
    let meshBufferLocations = Canvas.findMesh(meshIndex);
    if (meshBufferLocations){
      // Only add if mesh was found
      // map_object_index to access map properties like position
      // meshBufferLocations to access gl vertex rendering function
      processed_objects.push({
        map_object_index: i,
        meshBufferLocations: meshBufferLocations
      })
    }
  }
```

Explanation of the following functions (lines 55 to 94)

At_draw functions will be passed to Canvas.drawObject() which will pass itself (Canvas object) to the functions as a parameter.

At_Draw_1__generalSetup (canvasThis)

- Sets up general WebGL settings.

At_Draw_2__setAttribs(canvasThis, {vertexBufferRef, indexBufferRef})

- Tells WebGL which array of points and which array of triangles to use to draw each object.

At_Draw_3a__setUniforms_General(canvasThis, {timeFromStart})

- Sets the things invariant for all points of all objects. For example, time passed, or screen width and height.

At_Draw_3b__setUniforms_Player (canvasThis, {Player_pvuw})

- Sets the things invariant to do with the player/camera. This can be called once every loop, since the camera position will be the same irrespective of the object being drawn. **This is an area to optimise.**

At_Draw_3c__setUniforms_Object (canvasThis, {Object_pvuw})

- Sets the things invariant to do with each object.

These are combined using Generate_At_Draw (line 88) for every object every draw call. All the parameters for the At_Draw_x functions are passed into this. It returns a function At_Draw to which the canvas object can be passed.

```
55    // Set up the program here
56    function At_Draw_1__generalSetup(canvasThis){
57        let gl = canvasThis.gl;
58        gl.useProgram(canvasThis.program);
59        gl.enable(gl.DEPTH_TEST);
60    }
61    // Attributes are lists of vertices to render
62    function At_Draw_2__setAttribs(canvasThis, {vertexBufferRef, indexBufferRef}){
63      let gl = canvasThis.gl;
64      // vertex buffer
65      gl.bindBuffer(gl.ARRAY_BUFFER, vertexBufferRef);
66      gl.enableVertexAttribArray(canvasThis.attributeReferences['coordinates']);
67      gl.vertexAttribPointer(canvasThis.attributeReferences['coordinates'], 3, gl.FLOAT, false, 0, 0);
68      // Index buffer
69      gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBufferRef);
70    }
71    // Uniforms vary so must be recalculated every loop
72    // Time based Uniforms
73    function At_Draw_3a__setUniforms_General(canvasThis){
74        let gl = canvasThis.gl;
75    }
76    // Player position
77    function At_Draw_3b__setUniforms_Player(canvasThis){
78      let gl = canvasThis.gl;
79    }
```

```
80    // Object position
81    function At_Draw_3c__setUniforms_Object(canvasThis, {Object_size, Object_xy, Object_colour}){
82      let gl = canvasThis.gl;
83      gl.uniform1f(canvasThis.uniformReferences['size'], Object_size)
84      gl.uniform2fv(canvasThis.uniformReferences['object_xy'], Object_xy)
85      gl.uniform4fv(canvasThis.uniformReferences['object_colour'], Object_colour)
86    }
87    // Generates the function to call when drawing every frame
88    const Generate_At_Draw = ({vertexBufferRef, indexBufferRef, Object_size, Object_xy, Object_colour}) => (canvasThis) => {
89      At_Draw_1__generalSetup(canvasThis)
90      At_Draw_2__setAttribs(canvasThis, {vertexBufferRef, indexBufferRef})
91      At_Draw_3a__setUniforms_General(canvasThis)
92      At_Draw_3b__setUniforms_Player(canvasThis)
93      At_Draw_3c__setUniforms_Object(canvasThis, {Object_size, Object_xy, Object_colour})
94    }
```

Lines 100 – 123 run every few milliseconds.

However, lines 105-120 run at an interval of no less than msPerFrame.

Lines 109-117 run for every object in processed_objects (a.k.a every object on the map).

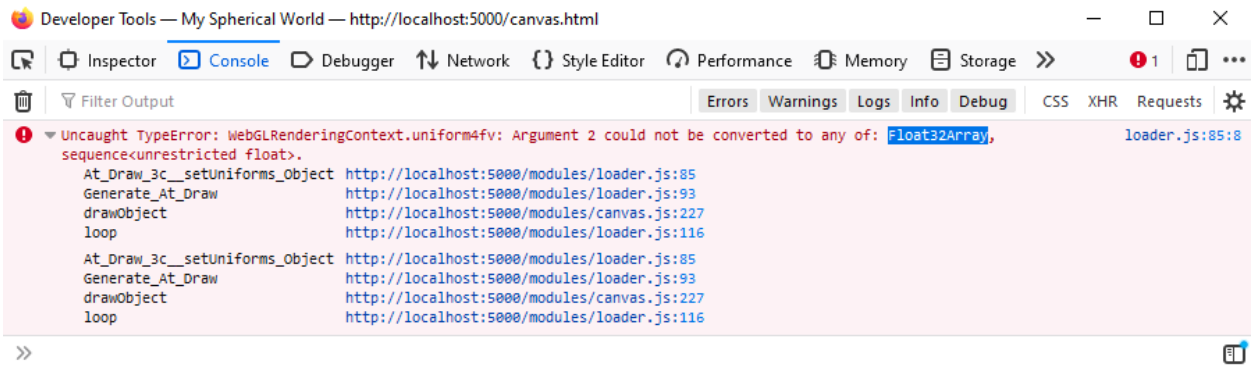
```
// Define the rendering loop function
let msPerFrame = 500;
let previousTime = Date.now()
function loop(){
  let timeNow = Date.now();
  let dt = timeNow - previousTime;
  // Render again if enough time has passed
  if (dt >= msPerFrame){
    Canvas.clearColour()
    let At_Draw;
    // Render every object
    for (let i=0; i< processed_objects.length; i++){
      let object_for_drawing = processed_objects[i];
      At_Draw = Generate_At_Draw({
        vertexBufferRef: object_for_drawing.meshBufferLocations.vertexBufferRef,
        indexBufferRef: object_for_drawing.meshBufferLocations.indexBufferRef,
        Object_size: Map_API.get_Object_size(object_for_drawing.map_object_index, map),
        Object_colour: Map_API.get_Object(object_for_drawing.map_object_index, map).colour,
        Object_xy: Map_API.get_Object(object_for_drawing.map_object_index, map).xy,
      })
      Canvas.drawObject(At_Draw, object_for_drawing.meshBufferLocations.metadata.numPoints);
    }
    previousTime = timeNow;
  }
  // loop after some time
  requestAnimationFrame(loop)
}
// Call loop first time to start rendering
requestAnimationFrame(loop);
```

With that, the objects should be rendered with their respective sizes and colours and coordinates.

P1.4 Tests and Alterations

I will test how all the sections programmed so far work together.

Running npm run game in /back folder, starts the server. Accessing it on localhost:5000/canvas.html **produces a blank canvas and an error in the console.**



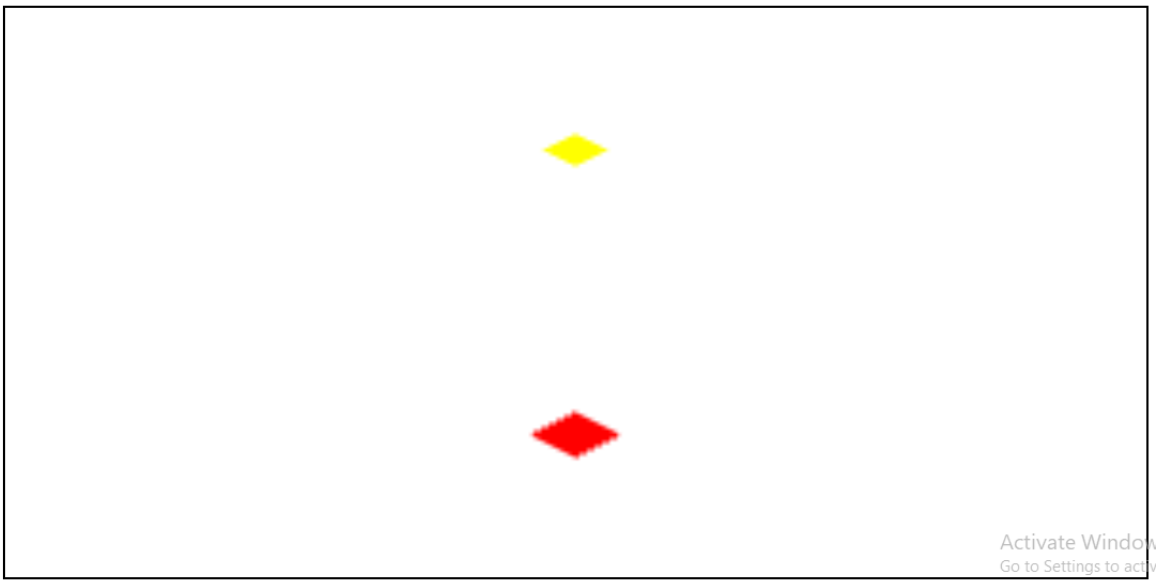
The error seemed to be here:

```
80    // Object position
81    function At_Draw_3c__setUniforms_Object(canvasThis, {Object_size, Object_xy, Object_colour}){
82      let gl = canvasThis.gl;
83      gl.uniform1f(canvasThis.uniformReferences['size'], Object_size)
84      gl.uniform2fv(canvasThis.uniformReferences['object_xy'], Object_xy)
85      gl.uniform4fv(canvasThis.uniformReferences['object_colour'], Object_colour)
86    }
```

And was caused by the fact that WebGL only accepts a certain structure as a list, not just a Javascript list. As a result, the corrected code looks like this.

```
80    // Object position
81    function At_Draw_3c__setUniforms_Object(canvasThis, {Object_size, Object_xy, Object_colour}){
82      let gl = canvasThis.gl;
83      gl.uniform1f(canvasThis.uniformReferences['object_size'], Object_size)
84      gl.uniform2fv(canvasThis.uniformReferences['object_xy'], new Float32Array(Object_xy))
85      gl.uniform4fv(canvasThis.uniformReferences['object_colour'], new Float32Array(Object_colour))
86    }
```

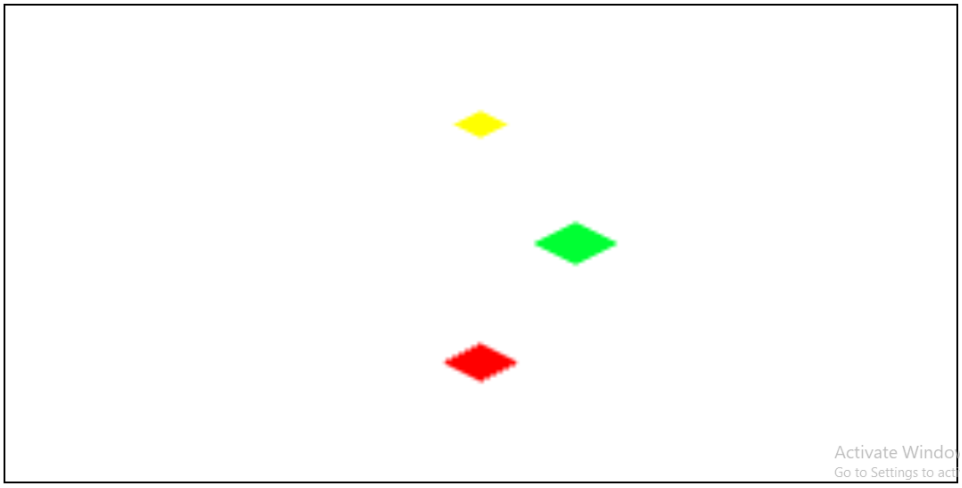
After the correction, here is the result (with the map from 1.3.2). (red has coordinates (0, -0.5) and yellow has coordinates (0, 0.5))



Adding the following entry to the map:

```
      {
        meshIndex: 0,
        pvuw: [
          0.0, 0.5, -0.84, 0.0,
          0.0, 0.0, 0.0, 1.0,
          0.0, 0.84, 0.5, 0.0,
          1.0, 0.0, 0.0, 0.0
        ],
        xy: [0.2, 0.0],
        size: 0.9,
        colour: [0.0, 1.0, 0.2, 1.0]
      }
    ]
}

export default map;
```

Produces this result:



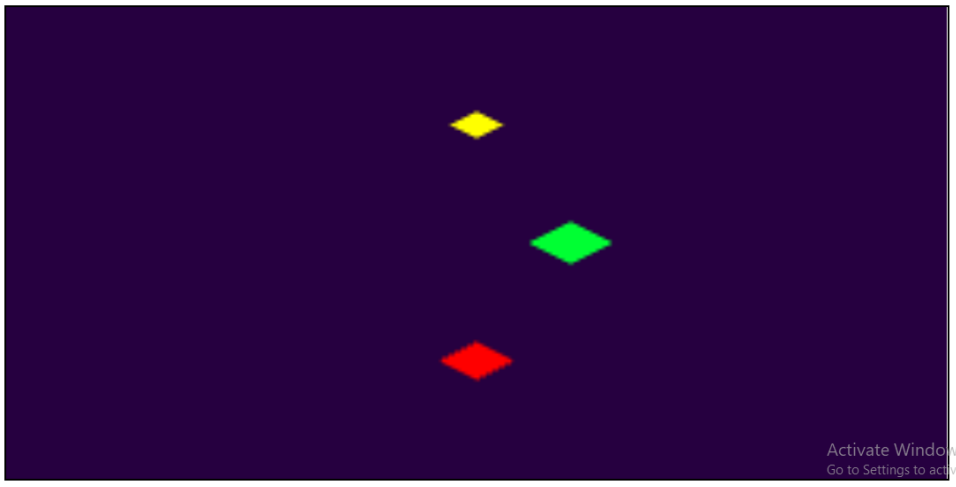
As you can see, the x-coordinate moved the object to the right of the center. It is also the right colour ([0, 1, 0.2, 1.0] in 0 to 1 rgba) and the biggest object on the map, with size 0.9 (others were 0.5 and 0.56).

The **unexpected result** seems to be that the canvas, being more horizontally stretched, stretches the objects as well. gl_Position seems to accept values between –1.0 and 1.0, 1.0 being right or top, and –1.0 being left or bottom of the canvas. This unexpected behaviour should be handled in the next iteration. **This is an area to correct in the next iteration.**

To test that the clearColour function does change the background colour, I have changed the background colour to dark blue.

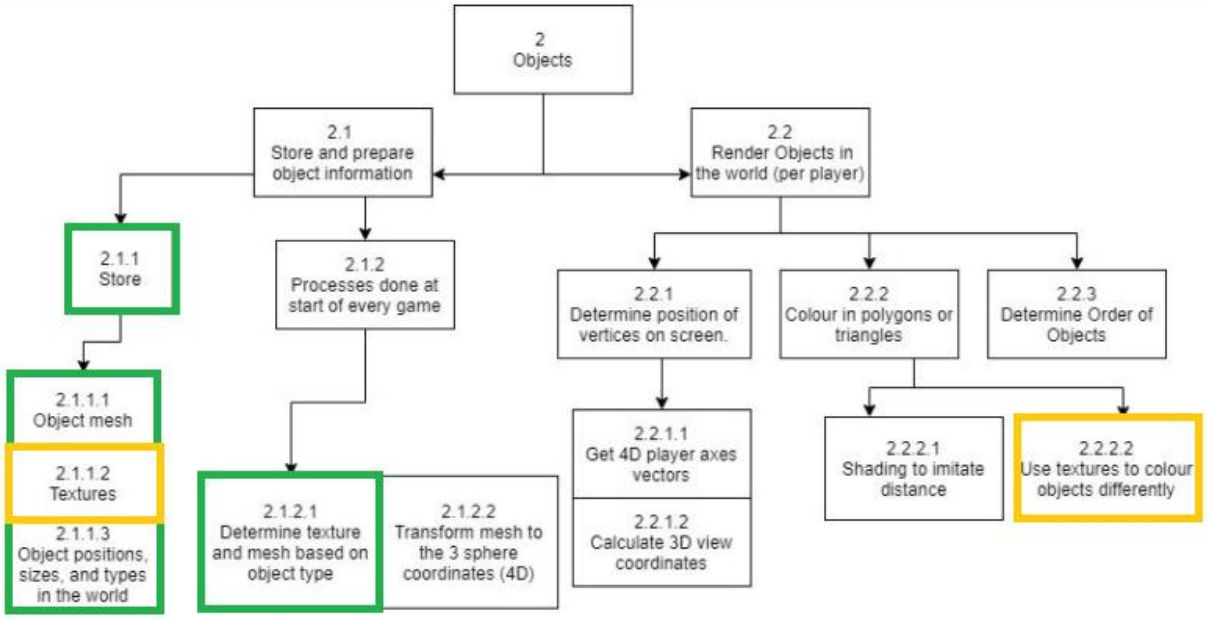
```
42    clearColour(){
43      this.gl.clearColor(0.15,0.0,0.25,1.0);
44      this.gl.clear(this.gl.COLOR_BUFFER_BIT)
45    }
```

This has reflected in the canvas correctly.



P1.5 Iteration summary and decisions

Key: **Completed** **Changed**



Changes:

- Textures will now be in the form of colours, but more complex designs may be introduced at a later stage.

Areas to work on in the next iteration:

- Shape stretching bug.

- Make planets more spherical

- Introduce shading by distance

P2. Rendering objects in 4D sphere mode.

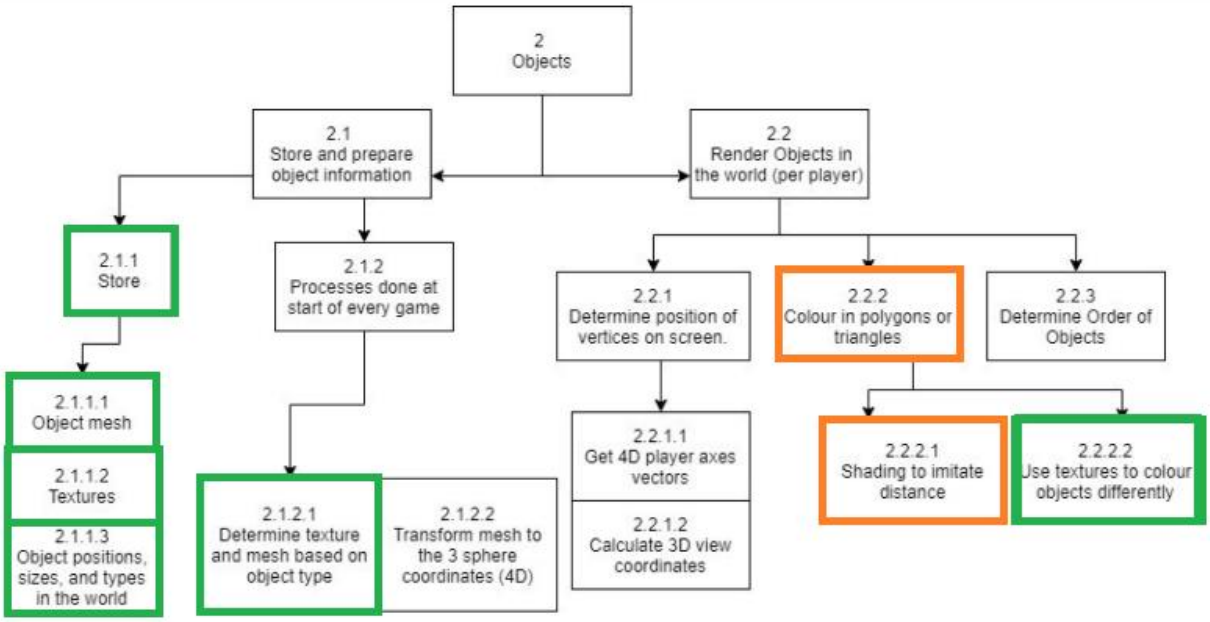
The development plan for this iteration is as follows:

P2.1 Correct the stretching effect.

P2.2 Make planets more spherical looking.

P2.3 Shade objects based on the z-coordinate (depth) to have a more accurate 3D picture. (2.2.2.1)

Key: **To program** **Changed** **Completed**



P2.1 Correcting the stretching effect.

P2.1.1 Planning

This section aims to solve the problem of the canvas stretching objects (see P1.4 if needed).

Technically, there are two parts to this:

1. Preserving object dimensions and preventing stretch.

2. Allowing the field of view to be controlled (what is the maximum angle the user can see into the world through the screen?)

(1) can be solved by dividing x by canvas_width and dividing y by canvas_height. When WebGL then scales these new coordinates by canvas_width and canvas_height, the desired x and y coordinates will be produced.

(2) can be solved by introducing a scale factor canvas_scale to increase or decrease the general size and coordinates of object from the origin.

This means 3 new uniforms need to be introduced: canvas_width, canvas_height, canvas_scale.

P2.1.2 Implementation

“loader.js” new uniforms

```
17    // Set up attributes and uniforms references
18    Canvas.add_AttributeReference({
19      'coordinates': 'coordinates'
20    })
21    Canvas.add_UniformReference({
22      'object_xy': 'object_xy',
23      'object_size':'object_size',
24      'object_colour': 'object_colour',
25      'canvas_height': 'canvas_height',
26      'canvas_width': 'canvas_width',
27      'canvas_scale': 'canvas_scale',
28    })
29    let canvas_scale_factor = 200;
```

```
74    // Uniforms vary so must be recalculated every loop
75    // Time based Uniforms
76    function At_Draw_3a__setUniforms_General(canvasThis){
77        let gl = canvasThis.gl;
78        gl.uniform1f(canvasThis.uniformReferences['canvas_height'], canvasThis.canvas_height)
79        gl.uniform1f(canvasThis.uniformReferences['canvas_width'], canvasThis.canvas_width)
80        gl.uniform1f(canvasThis.uniformReferences['canvas_scale'], canvas_scale_factor)
81    }
```

Vertex shader ”vertexShader1.js”

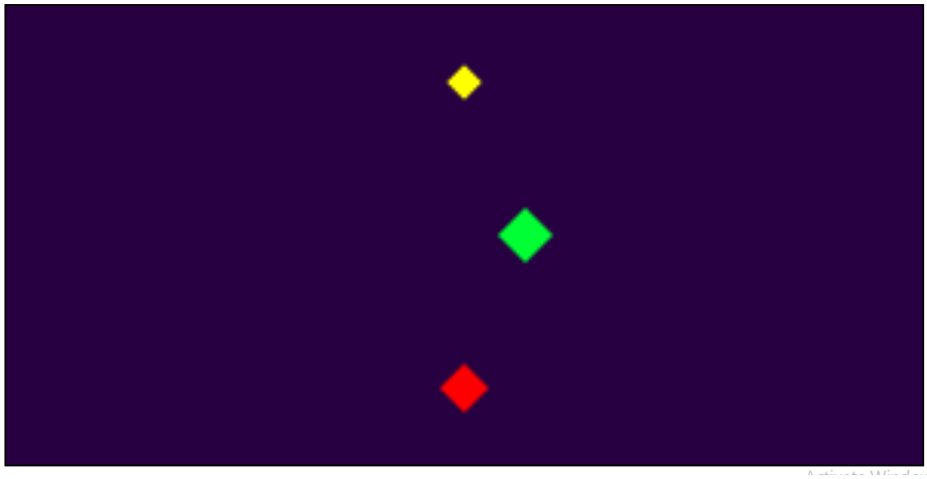
```
4     attribute vec3 coordinates;
5     uniform float object_size;
6     uniform vec2 object_xy;
7
8     uniform float canvas_height;
9     uniform float canvas_width;
10    uniform float canvas_scale;
11
12    vec4 scaleCoordsToCanvas(vec4 coords){
13      vec4 newCoords = vec4(
14        canvas_scale * coords.x / canvas_width,
15        canvas_scale * coords.y / canvas_height,
16        coords.z,
17        coords.w
18      );
19      return newCoords;
20    }
21
22    void main(){
23      vec3 coords = coordinates * object_size;
24      vec4 coords2 = vec4(
25        coords.x + object_xy.x,
26        coords.y + object_xy.y,
27        coords.z,
28      1.0);
29      gl_Position = scaleCoordsToCanvas(coords2);
30    }
```

Edited Canvas constructor method to get canvas height and width.

```
9     // Set up canvas and make it black.
10    constructor(canvas_HTML_ID){
11      this.canvas = document.getElementById(canvas_HTML_ID);
12      this.WebGLAvailable = this.getContext();
13      if (!this.WebGLAvailable){
14        return;
15      }
16      this.canvas_height = this.gl.drawingBufferHeight;
17      this.canvas_width = this.gl.drawingBufferWidth;
18      // Stores references to meshes that were input into the
```

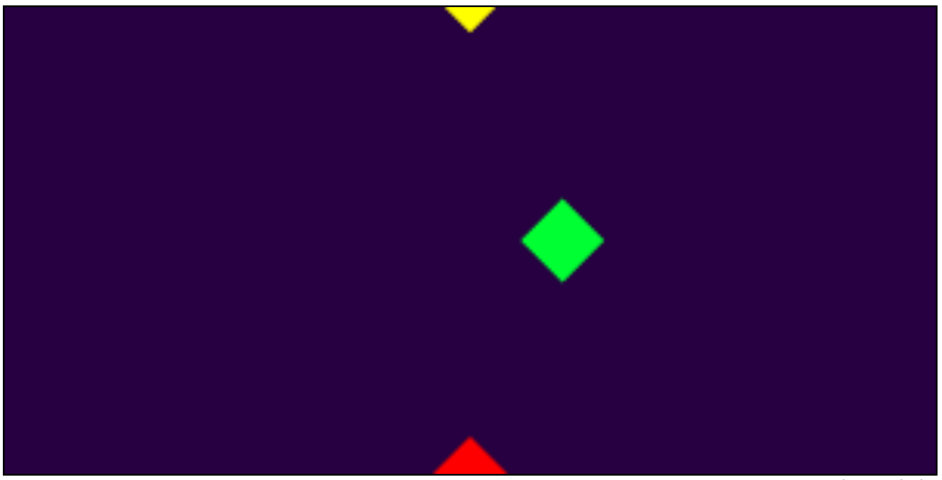
P2.1.3 Testing stretching correction

Here is the result. The shapes do not look stretched.



Increasing the scale factor from 200 to 300. This also works.

```
28      })
29      let canvas_scale_factor = 300;
30
```



P2.2 Program to make planets more spherical

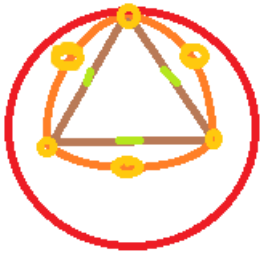
I made a program to generate a more spherical looking object of specified radius and smoothness.

It uses this algorithm...

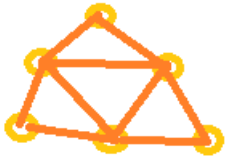
1. Start with a double square based pyramid.

2. For every triangle in the structure...

3. Select the midpoints of the lines making the triangle.



4. Extrude midpoints to sphere.



5. Make four triangles using these new vertices, and the old vertices.

6. Repeat the process for these new triangles to make the shape smoother and smoother.

I have programmed it in python.

```
# ========================================================= #

NUM_ITERATIONS = 1
RADIUS = 0.1

# Use iterative vertex generation method with triangles
# Every triangle becomes four triangles

# Vertex Position (3D) array. Every 3 floats are one coordinate.
_v = [
    0.0,  0.0, RADIUS,
    0.0,  RADIUS,  0.0,
    RADIUS, 0.0, 0.0,
    0.0,  0.0, -RADIUS,
    0.0,  -RADIUS,  0.0,
    -RADIUS, 0.0, 0.0,
]
```

```
# Triangles made using 3 indexed points. Indexes to _v array.
_i = [
    0, 1, 2,
    3, 2, 1,
    2, 4, 0,
    1, 0, 5,
    1, 5, 3,
    4, 5, 0,
    5, 4, 3,
    2, 3, 4,
]

# Above is a starting solid
# Below is excitement!

for hmm in range(NUM_ITERATIONS):
    # Make a new point for every pair of connected vertices
    # lines_visited stores [[v1, v2, newVertexIndex]...]
    lines_visited = []
    _i_new = []
    for tri_index in range(len(_i)//3):
        # For every side of triangle, check and make new vertex.
        # Store their indexes temporarily here as [[v1, v2, vN]...]
        midpoints = []
        for connection in [0, 1, 2]:
            line = [_i[3*tri_index + (connection)%3], _i[3*tri_index + (connection+1)%3]]
            result = find_Line_found_vN_reversed(lines_visited, line)
```

```
            if result[0] == True: # found
                vN = result[1] # mid vertex index
            else: # not found
                vN_coords = generate_new_vertex_coords_from_other_two(
                    [_v[3*line[0]], _v[3*line[0] + 1], _v[3*line[0] + 2]],
                    [_v[3*line[1]], _v[3*line[1] + 1], _v[3*line[1] + 2]],
                    RADIUS)
                vN = len(_v)//3
                _v.append(vN_coords[0])
                _v.append(vN_coords[1])
                _v.append(vN_coords[2])
                lines_visited.append([line[0],line[1], vN])
                pass
            # Add to list of midpoints of this triangle for next part
            midpoints.append([line[0], line[1], vN])

        # Now make new triangles
        # Middle Triangle
        _i_new.append(midpoints[0][2])
        _i_new.append(midpoints[1][2])
        _i_new.append(midpoints[2][2])
        # Triangle 1
        _i_new.append(midpoints[0][0])
        _i_new.append(midpoints[0][2])
        _i_new.append(midpoints[2][2])
        # Triangle 2
        _i_new.append(midpoints[1][0])
        _i_new.append(midpoints[1][2])
        _i_new.append(midpoints[0][2])
        # Triangle 3
        _i_new.append(midpoints[2][0])
        _i_new.append(midpoints[2][2])
        _i_new.append(midpoints[1][2])
    _i = _i_new

print(_v)
print(_i)

```

```
def find_Line_found_vN_reversed(v1v2vN_list, line):
    # Tries to find the line or the line reversed in a list of lines.
    # Returns found, the vertex associated with the line, whether the line is stored reversed.
    found = False
    reversed = False
    vertex = -1

    for v1v2vN in v1v2vN_list:
        if (v1v2vN[0] == line[0]) and (v1v2vN[1] == line[1]):
            found = True
            reversed = False
            vertex = v1v2vN[2]

        elif (v1v2vN[0] == line[1]) and (v1v2vN[1] == line[0]):
            found = True
            reversed = False
            vertex = v1v2vN[2]

    return [found, vertex, reversed]

def generate_new_vertex_coords_from_other_two(v1, v2, radius):
    mid = [(v1[0]+v2[0]), (v1[1]+v2[1]), (v1[2]+v2[2])]
    magnitude = (mid[0]**2 + mid[1]**2 + mid[2]**2)**(0.5)
    sf = radius/magnitude
    return [sf*mid[0], sf*mid[1], sf*mid[2]]

```

NUM_ITERATIONS is the number of times to run the loop for, and corresponds to the smoothness of the shape.

RADIUS is the maximum radius of the resulting object.

The program turns one array of triangles into a completely new one every iteration.

Here is the output for radius=0.1, and 1 iteration. First is the vertex array, and second is the triangle index array.

```
C:\Users\Maxicl\code\nice\A-spherical-world\code1>cd programs

C:\Users\Maxicl\code\nice\A-spherical-world\code1\programs>python3 makeasphere.py
[0.0, 0.0, 0.1, 0.0, 0.1, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, -0.1, 0.0, -0.1, 0.0, -0.1,
0.0, 0.0, 0.0, 0.07071067811865475, 0.07071067811865475, 0.07071067811865475, 0.0707
1067811865475, 0.0, 0.07071067811865475, 0.0, 0.07071067811865475, 0.070710678118654
75, 0.0, -0.07071067811865475, 0.0, 0.07071067811865475, -0.07071067811865475, 0.070
71067811865475, -0.07071067811865475, 0.0, 0.0, -0.07071067811865475, 0.070710678118
65475, -0.07071067811865475, 0.0, 0.07071067811865475, -0.07071067811865475, 0.07071
067811865475, 0.0, -0.07071067811865475, 0.0, -0.07071067811865475, -0.0707106781186
5475, -0.07071067811865475, 0.0, 0.0, -0.07071067811865475, -0.07071067811865475]
[6, 7, 8, 0, 6, 8, 1, 7, 6, 2, 8, 7, 9, 7, 10, 3, 9, 10, 2, 7, 9, 1, 10, 7, 11, 12,
8, 2, 11, 8, 4, 12, 11, 0, 8, 12, 6, 13, 14, 1, 6, 14, 0, 13, 6, 5, 14, 13, 14, 15,
10, 1, 14, 10, 5, 15, 14, 3, 10, 15, 16, 13, 12, 4, 16, 12, 5, 13, 16, 0, 12, 13, 16
, 17, 15, 5, 16, 15, 4, 17, 16, 3, 15, 17, 9, 17, 11, 2, 9, 11, 3, 17, 9, 4, 11, 17]
```

The start of the output for 2 iterations.

```
C:\Users\Maxicl\code\nice\A-spherical-world\code1\programs>
C:\Users\Maxicl\code\nice\A-spherical-world\code1\programs>
C:\Users\Maxicl\code\nice\A-spherical-world\code1\programs>python3 makeasphere.py
[0.0, 0.0, 0.1, 0.0, 0.1, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, -0.1, 0.0, -0.1, 0.0, -0.1, 0.0, 0.0, 0.0, 0.070
71067811865475, 0.07071067811865475, 0.07071067811865475, 0.07071067811865475, 0.0, 0.07071067811865475,
 0.0, 0.07071067811865475, 0.07071067811865475, 0.0, -0.07071067811865475, 0.0, 0.07071067811865475, -0.
07071067811865475, 0.07071067811865475, -0.07071067811865475, 0.0, 0.0, -0.07071067811865475, 0.07071067
811865475, -0.07071067811865475, 0.0, 0.07071067811865475, -0.07071067811865475, 0.07071067811865475, 0.
0, -0.07071067811865475, 0.0, -0.07071067811865475, -0.07071067811865475, -0.07071067811865475, 0.0, 0.0
, -0.07071067811865475, -0.07071067811865475, 0.040824829046386304, 0.08164965809277261, 0.0408248290463
86304, 0.08164965809277261, 0.040824829046386304, 0.040824829046386304, 0.040824829046386304, 0.04082482
9046386304, 0.08164965809277261, 0.0, 0.038268343236508975, 0.09238795325112868, 0.038268343236508975, 0
.0, 0.09238795325112868, 0.038268343236508975, 0.09238795325112868, 0.0, 0.0, 0.09238795325112868, 0.038
268343236508975, 0.09238795325112868, 0.0, 0.038268343236508975, 0.09238795325112868, 0.0382683432365089
75, 0.0, 0.08164965809277261, 0.040824829046386304, -0.040824829046386304, 0.040824829046386304, 0.08164
```

I have put the vertex and index arrays into models/planet.js .

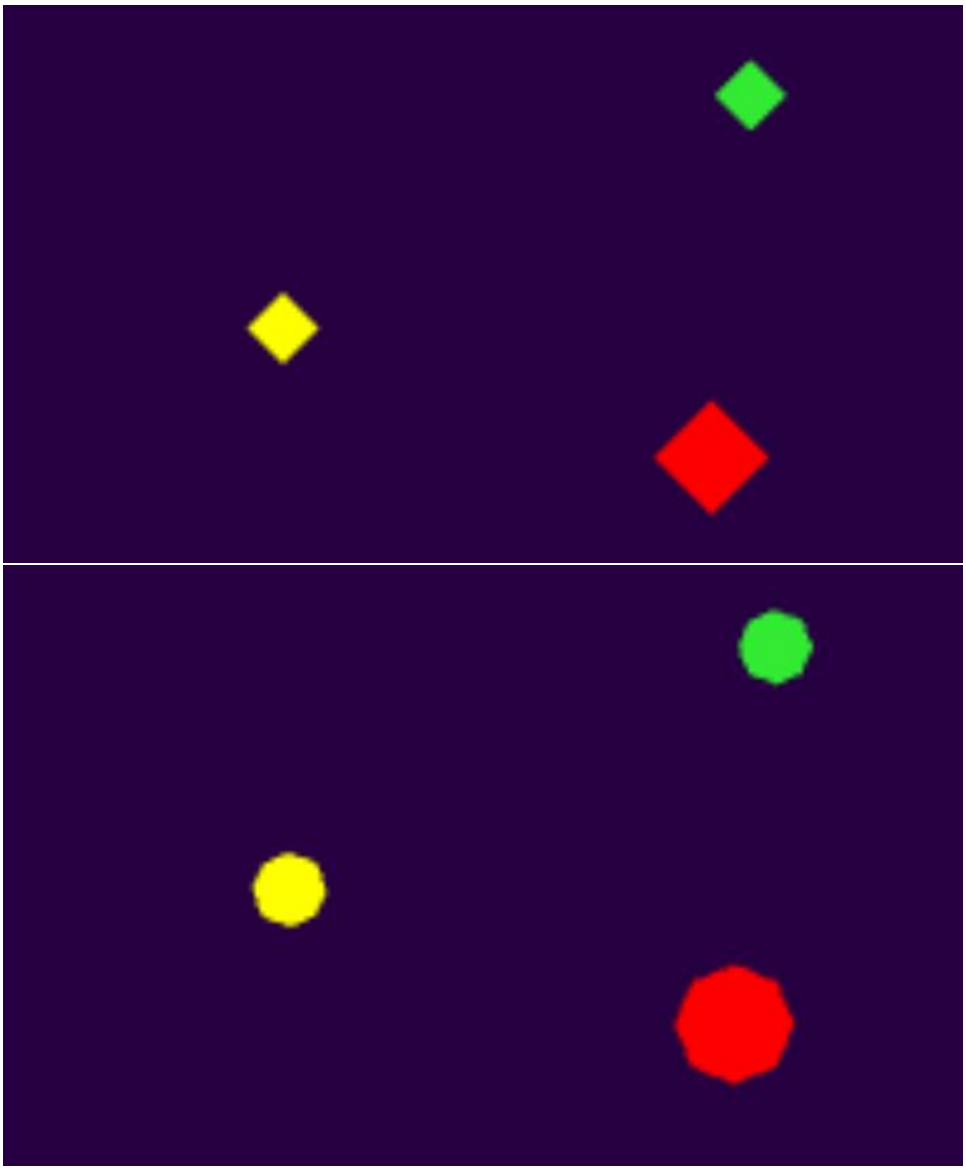
canvas.html | loader.js | makeasphere.py | *planet.js* | Settings

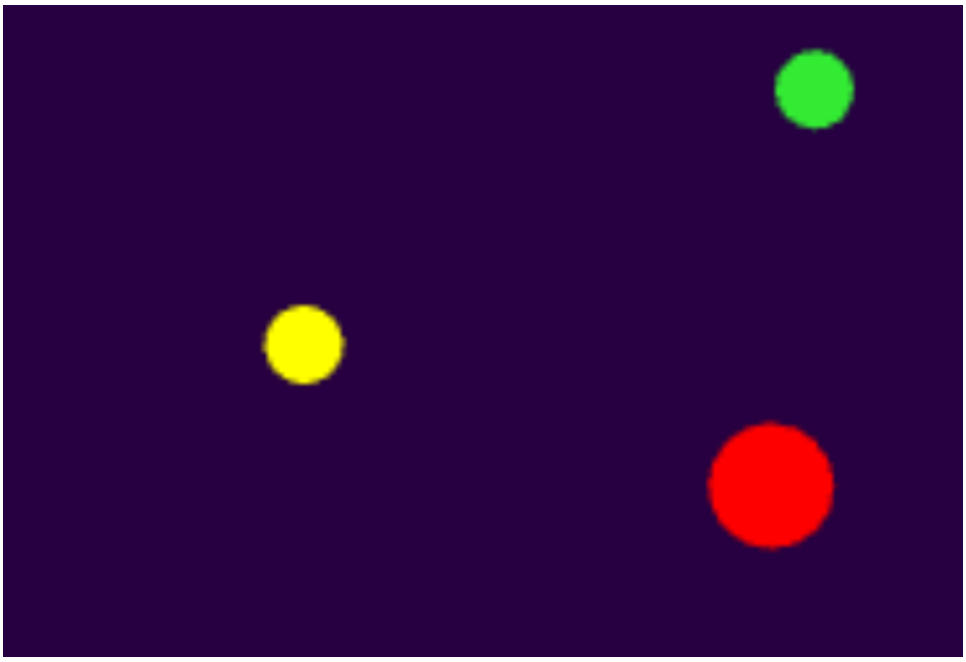
```
  1, 2, 5,
  1, 3, 4,
  1, 3, 5,
]

const pv_1 = [0.0, 0.0, 0.1, 0.0, 0.1, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, -0.1, 0.0, -0.1, 0.0, -0.1, 0.0, 0.0, 0.0, 0.07071067811865475
0.07071067811865475, 0.07071067811865475, 0.0, -0.07071067811865475, 0.0, -0.07071067811865475, -0.07071067811865475, -0.0707106781
const pti_1 = [6, 7, 8, 0, 6, 8, 1, 7, 6, 2, 8, 7, 9, 7, 10, 3, 9, 10, 2, 7, 9, 1, 10, 7, 11, 12, 8, 2, 11, 8, 4, 12, 11, 0, 8, 12,

const pv_2 = [0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, -1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.7071067811865475,
0.7071067811865475, 0.0, -0.7071067811865475, 0.0, -0.7071067811865475, -0.7071067811865475, -0.7071067811865475, 0.0, 0.0, -0.7071
0.3826834323650898, 0.9238795325112867, 0.0, 0.3826834323650898, 0.9238795325112867, 0.3826834323650898, 0.0, 0.816496580927726, 0.
0.816496580927726, 0.408248290463863, 0.408248290463863, -0.408248290463863, 0.816496580927726, 0.816496580927726, -0.4082482904638
0.408248290463863, -0.3826834323650898, 0.9238795325112867, 0.0, -0.3826834323650898, 0.0, 0.9238795325112867, -0.9238795325112867,
0.408248290463863, 0.408248290463863, -0.408248290463863, -0.408248290463863, 0.816496580927726, -0.408248290463863, -0.81649658092
0.3826834323650898, -0.9238795325112867, 0.408248290463863, -0.408248290463863, -0.816496580927726, 0.408248290463863, -0.816496580
const pti_2 = [0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, -1.0, 0.0, -1.0, 0.0, 0.0, 0.0, 0.7071067811865475
0.7071067811865475, 0.0, -0.7071067811865475, 0.0, -0.7071067811865475, -0.7071067811865475, -0.7071067811865475, 0.0, 0.0, -0.7071
0.3826834323650898, 0.9238795325112867, 0.0, 0.3826834323650898, 0.9238795325112867, 0.3826834323650898, 0.0, 0.816496580927726, 0.
0.816496580927726, 0.408248290463863, 0.408248290463863, -0.408248290463863, 0.816496580927726, 0.816496580927726, -0.4082482904638
0.408248290463863, -0.3826834323650898, 0.9238795325112867, 0.0, -0.3826834323650898, 0.0, 0.9238795325112867, -0.9238795325112867,
0.408248290463863, 0.408248290463863, -0.408248290463863, -0.408248290463863, 0.816496580927726, -0.408248290463863, -0.81649658092
0.3826834323650898, -0.9238795325112867, 0.408248290463863, -0.408248290463863, -0.816496580927726, 0.408248290463863, -0.816496580


const Model_planet = { vertices: pv_1, trianglesIndexed: pti_1 };
```

(pv = planet vertices pti = planet triangles indexed)

Here are the results for 0, 1 and 2 iterations.

These are as expected.





P2.3 Shading based on distance

The algorithm I chose to implement varies the colour of a point based on the distance to it between set bounds.

FragmentShader1.js

```
9     uniform vec4 object_colour;
10    uniform float depthFactor;
11    varying float depth; // -1.0 to +1.0
12
13    void main(){
14      float k = depthFactor;
15      float depth2 = (1.0-k) + (k)*(1.0-((depth + 1.0) / 2.0)); // From 0.0 to 1.0
16      gl_FragColor = vec4(depth2 * object_colour.xyz, object_colour.w);
17    }
18
```

A depth factor is set representing the contrast to go for.

Loader.js

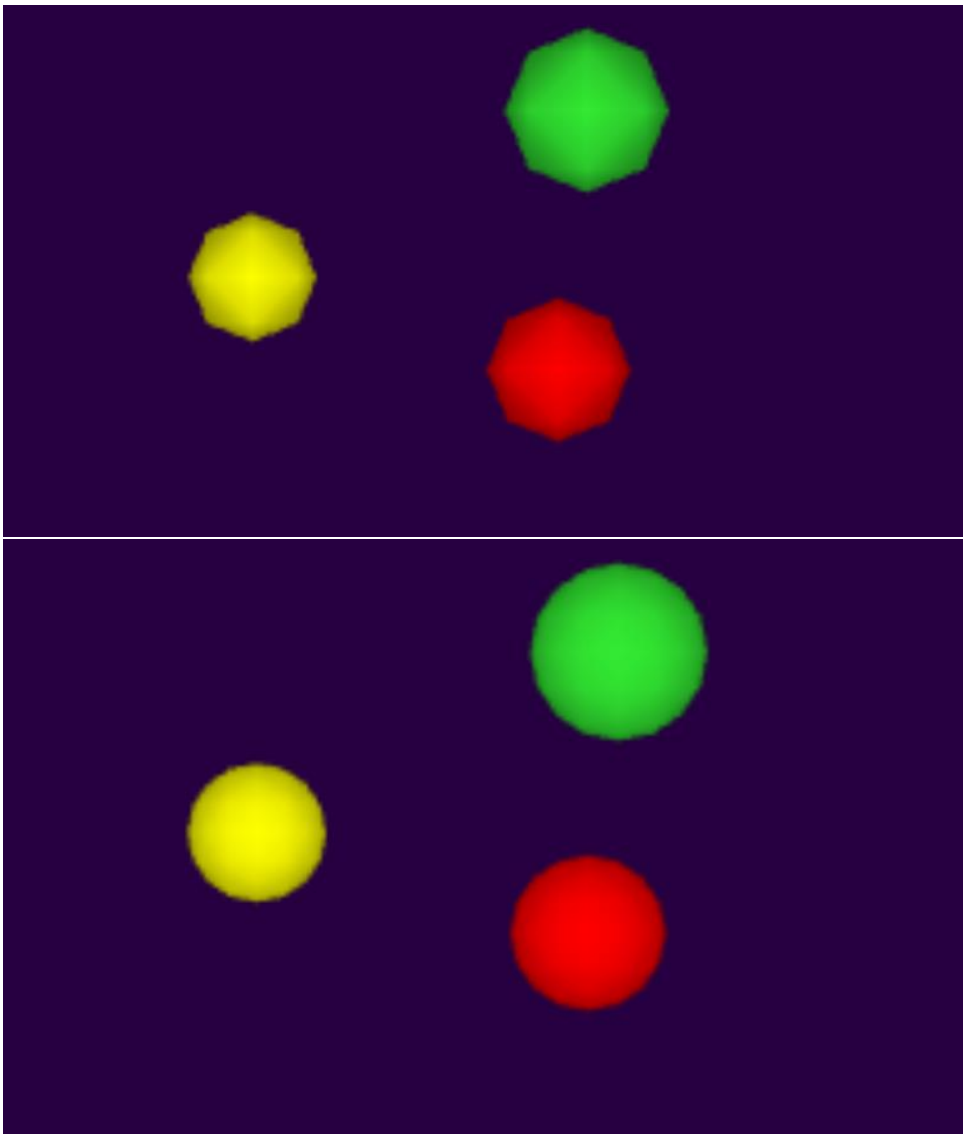
```
77    // Uniforms vary so must be recalculated every loop
78    // Time based Uniforms
79    function At_Draw_3a__setUniforms_General(canvasThis){
80        let gl = canvasThis.gl;
81        gl.uniform1f(canvasThis.uniformReferences['canvas_height'], canvasThis.canvas_height)
82        gl.uniform1f(canvasThis.uniformReferences['canvas_width'], canvasThis.canvas_width)
83        gl.uniform1f(canvasThis.uniformReferences['canvas_scale'], canvas_scale_factor)
84        gl.uniform1f(canvasThis.uniformReferences['depthFactor'], depthFactor)
85    }
```

The varying depth can be set in vertexShader1.js.

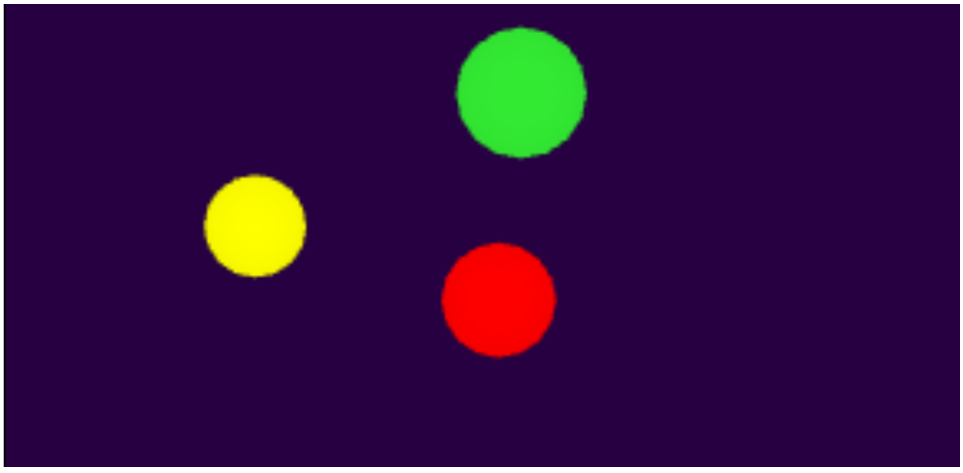
```
    gl_Position = scaleCoordsToCanvas(coords2);

    depth = coordinates.z;
}
```

The results...

Depth factor 0.9 for different models.



Depth factor 0.0 for circular model (no contrast shown).

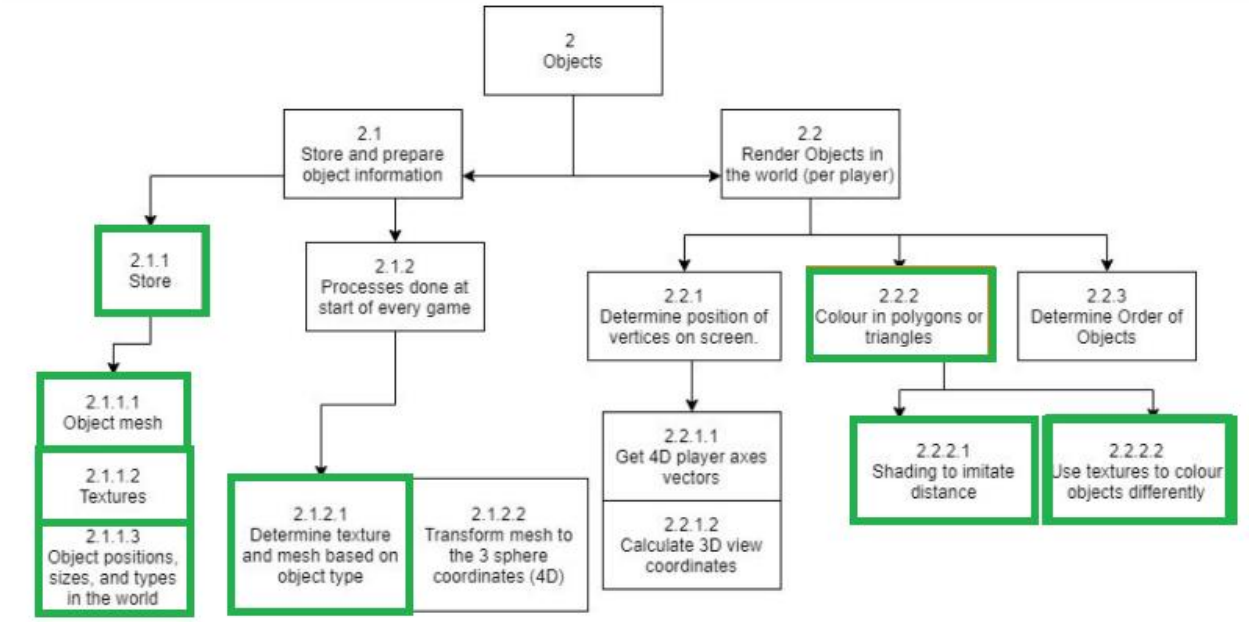


P2.4 Evaluation of iteration

All aspects of this iteration have gone well, and I am ready to move on to programming the 4D sphere effect.

I may still want to change the shading algorithm at a later point.

Key: **Completed**



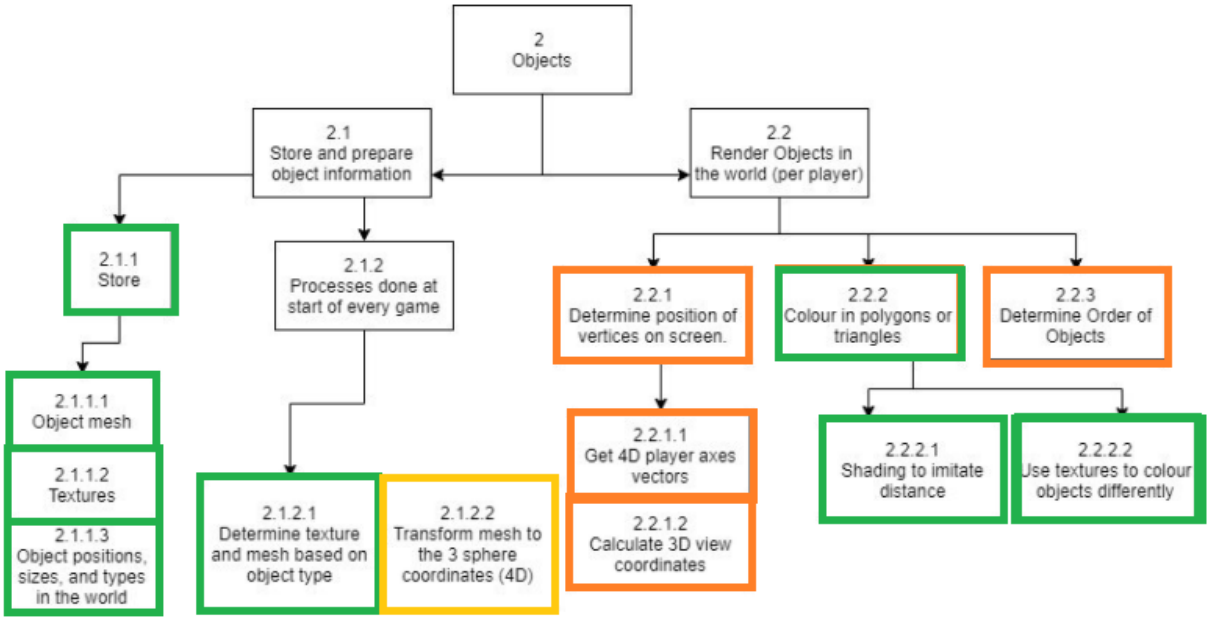
P3 Introducing the 4D sphere effect

The plan for this section is as follows:

P3.1 Display planets in 3sphere mode from the player's perspective.

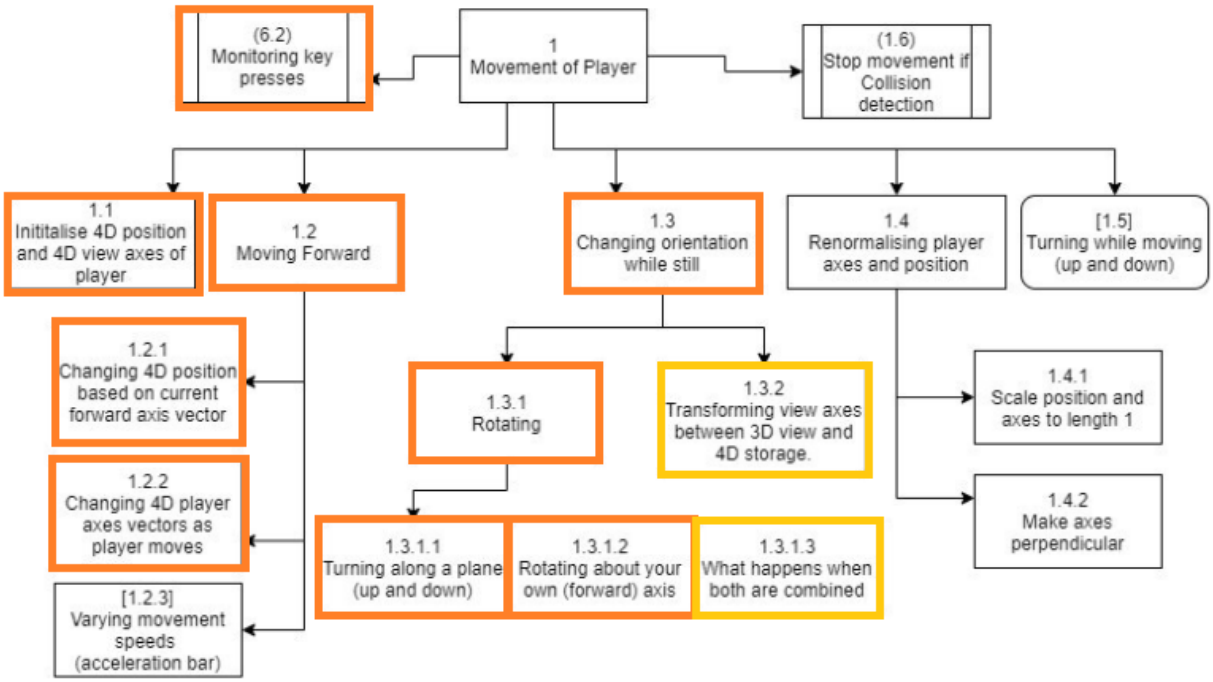
P3.2 Create the player class. This would allow the user to use selected keys to rotate and move themselves through the world.

Key: **To program now** **Changed** **Completed**



Change to 2.1.2.2: Instead of transforming meshes to 4D sphere coordinates for each object at the start, and using these in the vertex shader, I will calculate the 4D sphere coordinates for every object each time the vertex shader runs. I figured that, if meshes get very detailed for very smooth planets, all the different object meshes, especially stored in 4D coordinates would take up a lot of space. Instead, I will use one original mesh for all planets, and pass their pvuw coordinates to the vertex shader, for it to compute the resulting 4D coordinates. Although this may increase the time complexity per frame, it would decrease the memory complexity. Copying less data to the vertex shader buffer would also save time so time complexity may actually decrease.

Key: **To program now** **Changed** **Completed**



Change to 1.3.2: This is not necessary any more with the algorithm I am using.

Change to 1.3.1.3: To confirm, when both keys are pressed, the later one should be dominant, and activate only its rotation. A combination of rotations will not be possible.

P3.1 Display planets in 4D Sphere mode

VertexShader1.js

Now accepts a camera 4D coordinate and orientation, camera_pvuw.

And an object 4D coordinate and orientation, object_pvuw.

After that are some helper functions.

```
12  uniform mat4 camera_pvuw;
13  uniform mat4 object_pvuw;
14
15  varying float depth;
16
17  float magnitude(vec4 vector){
18    return sqrt(vector.x*vector.x + vector.y*vector.y + vector.z*vector.z + vector.w*vector.w);
19  }
20  float magnitude(vec3 vector){
21    return sqrt(vector.x*vector.x + vector.y*vector.y + vector.z*vector.z);
22  }
23
24  float det(mat2 matrix){
25    return ( matrix[0].x*matrix[1].y - matrix[0].y*matrix[1].x ) ;
26  }
27
28  vec4 unit(vec4 v){
29    return v / magnitude(v);
30  }
```

```
33   mat3 inverse(mat3 matrix) {
34       vec3 row0 = matrix[0];
35       vec3 row1 = matrix[1];
36       vec3 row2 = matrix[2];
37       vec3 minors0 = vec3(
38           det(mat2(row1.y, row1.z, row2.y, row2.z)),
39           det(mat2(row1.z, row1.x, row2.z, row2.x)),
40           det(mat2(row1.x, row1.y, row2.x, row2.y))
41       );
42       vec3 minors1 = vec3(
43           det(mat2(row2.y, row2.z, row0.y, row0.z)),
44           det(mat2(row2.z, row2.x, row0.z, row0.x)),
45           det(mat2(row2.x, row2.y, row0.x, row0.y))
46       );
47       vec3 minors2 = vec3(
48           det(mat2(row0.y, row0.z, row1.y, row1.z)),
49           det(mat2(row0.z, row0.x, row1.z, row1.x)),
50           det(mat2(row0.x, row0.y, row1.x, row1.y))
51       );
52       mat3 adj;
53       adj[0] =vec3(minors0.x, minors1.x, minors2.x);
54       adj[1] =vec3(minors0.y, minors1.y, minors2.y);
55       adj[2] =vec3(minors0.z, minors1.z, minors2.z);
56       return (1.0 / dot(row0, minors0)) * adj;
57   }
58
```

This function calculates the player view coordinates of a vertex of the model, as explained in the design section.

```
70  vec4 getCoords_4D_modified(vec3 vertex_model_coordinates, mat4 object_pvuw, mat4 camera_pvuw){
71
72          float Pi = 3.14;
73
74          // Transfer 3D model coordinate to 4D world coordinate
75          vec3 c = vertex_model_coordinates;
76          vec4 Direction_4D_to_point = unit( object_pvuw * vec4(0.0, c.xyz) );
77          float Distance_to_point = magnitude(c);
78          vec4 Object_center = object_pvuw[0].xyzw;
79          float cosD = cos(Distance_to_point*Pi);
80          float sinD = sin(Distance_to_point*Pi);
81          vec4 Vertex_4D_coordinates = (cosD * Object_center) + (sinD * Direction_4D_to_point);
82
83          // Calculating distance and direction camera to point
84          vec4 C_Pos4D = camera_pvuw[0].xyzw;
85          vec4 P_Pos4D = Vertex_4D_coordinates;
86          // Calculating angular distance D
87          cosD = dot(C_Pos4D, P_Pos4D);
88          sinD = sqrt(1.0-cosD*cosD);
89          vec4 camera_to_point = (P_Pos4D - (C_Pos4D * cosD))/( sinD );
90          if (dot(camera_to_point, camera_pvuw[1].xyzw) < 0.0) {
91            // Facing wrong way
92            // camera_to_point = -camera_to_point;
93          }
94
95          // Calculate the 3D direction from player to object
```

```
95            // Calculate the 3D direction from player to object
96            mat3 concatenated_axes;
97            concatenated_axes[0] = camera_pvuw[1].xyz;
98            concatenated_axes[1] = camera_pvuw[2].xyz;
99            concatenated_axes[2] = camera_pvuw[3].xyz;
100           mat3 invConcatAxes = inverse(concatenated_axes);
101           // Correspond to forward, right and up axes v, u, w.
102           vec3 abc = invConcatAxes * camera_to_point.xyz;
103
104           return abc;
105   }
```

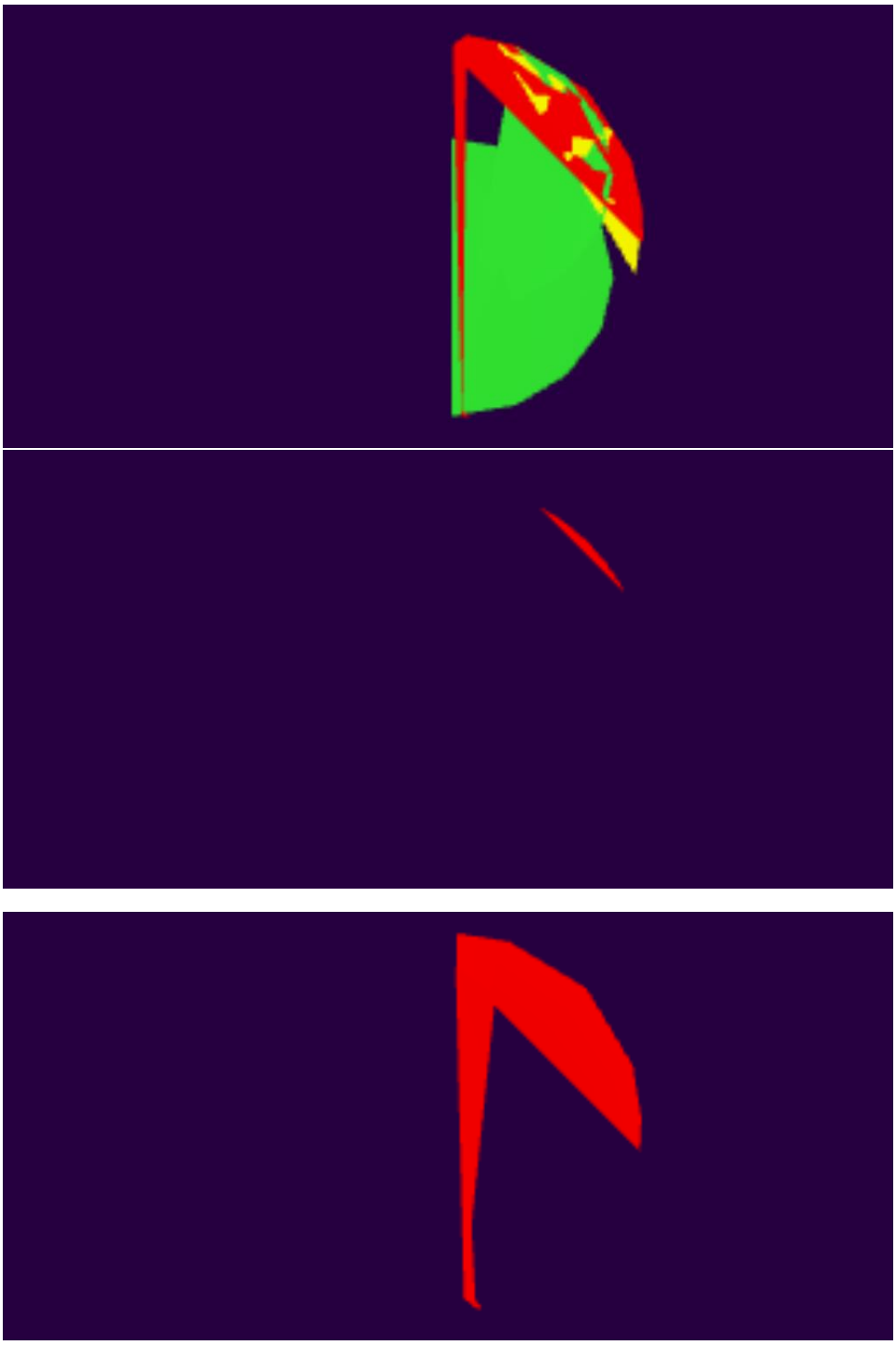
```
107  vec3 getCoords_3D_translated(vec3 coords, vec2 object_translate_vector){
108    return vec3(
109      coords.x + object_translate_vector.x,
110      coords.y + object_translate_vector.y,
111      coords.z
112    );
113  }
114
115  void shader_3D_flat(){
116    vec3 coords = coordinates * object_size;
117    vec3 coords2 = getCoords_3D_translated(coords, object_xy);
118    vec3 coords3 = scaleCoordsToCanvas(coords2);
119    gl_Position = vec4(coords3, 1.0);
120    depth = coords.z;
121  }
122
123  void shader_4D_spherical(){
124    vec3 coords = coordinates * object_size;
125    vec3 xyz = getCoords_4D_modified(coords, object_pvuw, camera_pvuw);
126    vec3 xyz_scale = scaleCoordsToCanvas(xyz);
127    gl_Position = vec4(xyz_scale, 1.0);
128    depth = gl_Position.z;
129  }
130
131  void main(){
132    shader_4D_spherical();
133    // shader_3D_flat();
134  }
```

The shader_4D_spherical method is used to render the planets in 4D sphere world.

The shader_3D_flat is the method we used before to render the planets using (x,y) coordinates.

I put the two methods into procedures, so that I could decide which to use, for testing purposes.

The results...



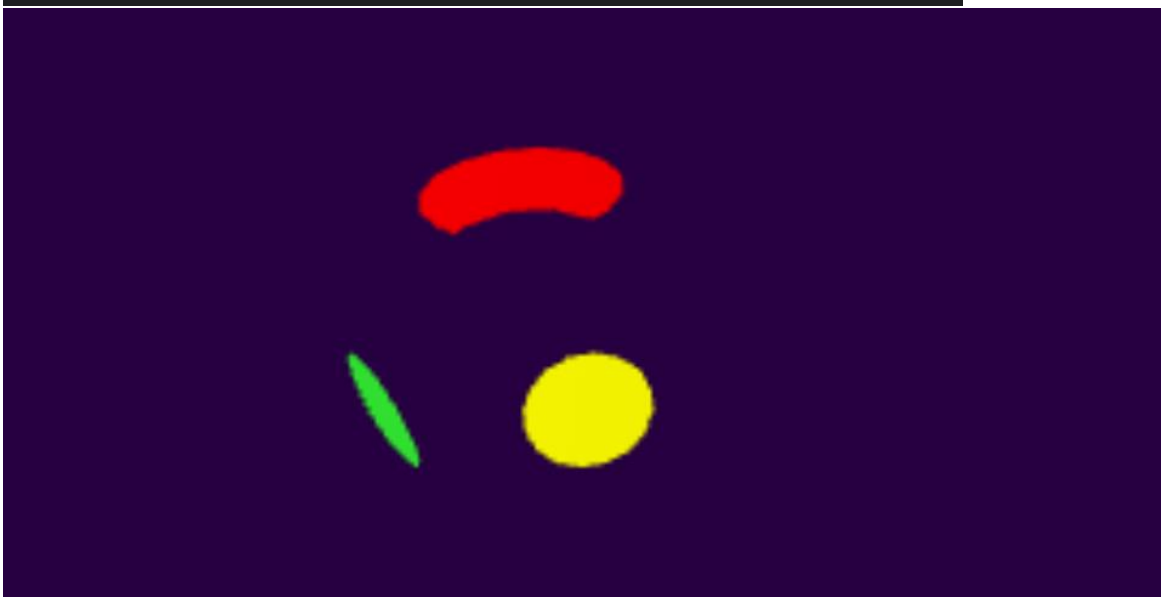
It seemed very weird to me that the coloured bits should only be on the right. Also, that no real spherical objects should be seen.

I decided to look through the code again and found the error to be here.

```
95          // Calculate the 3D direction from player to object
96          mat3 concatenated_axes;
97          concatenated_axes[0] = camera_pvuw[1].xyz;
98          concatenated_axes[1] = camera_pvuw[2].xyz;
99          concatenated_axes[2] = camera_pvuw[3].xyz;
100         mat3 invConcatAxes = inverse(concatenated_axes);
101         // Correspond to forward, right and up axes v, u, w.
102         vec3 abc = invConcatAxes * camera_to_point.xyz;
103
104         return abc;
105     }
```

abc actually corresponds to vuw – the **forward, right, up** axes. Therefore, these need to be shifted to correspond to **right, up, forward**.

```
        // Correspond to forward, right and up axes v, u, w.
        vec3 abc = invConcatAxes * camera_to_point.xyz;
        vec3 xyz = vec3(abc.y, abc.z, abc.x);
        return xyz;
 }
```



The result seems to be a lot more planet looking.

P3.2 Moving the player

P3.2.1 Implementation

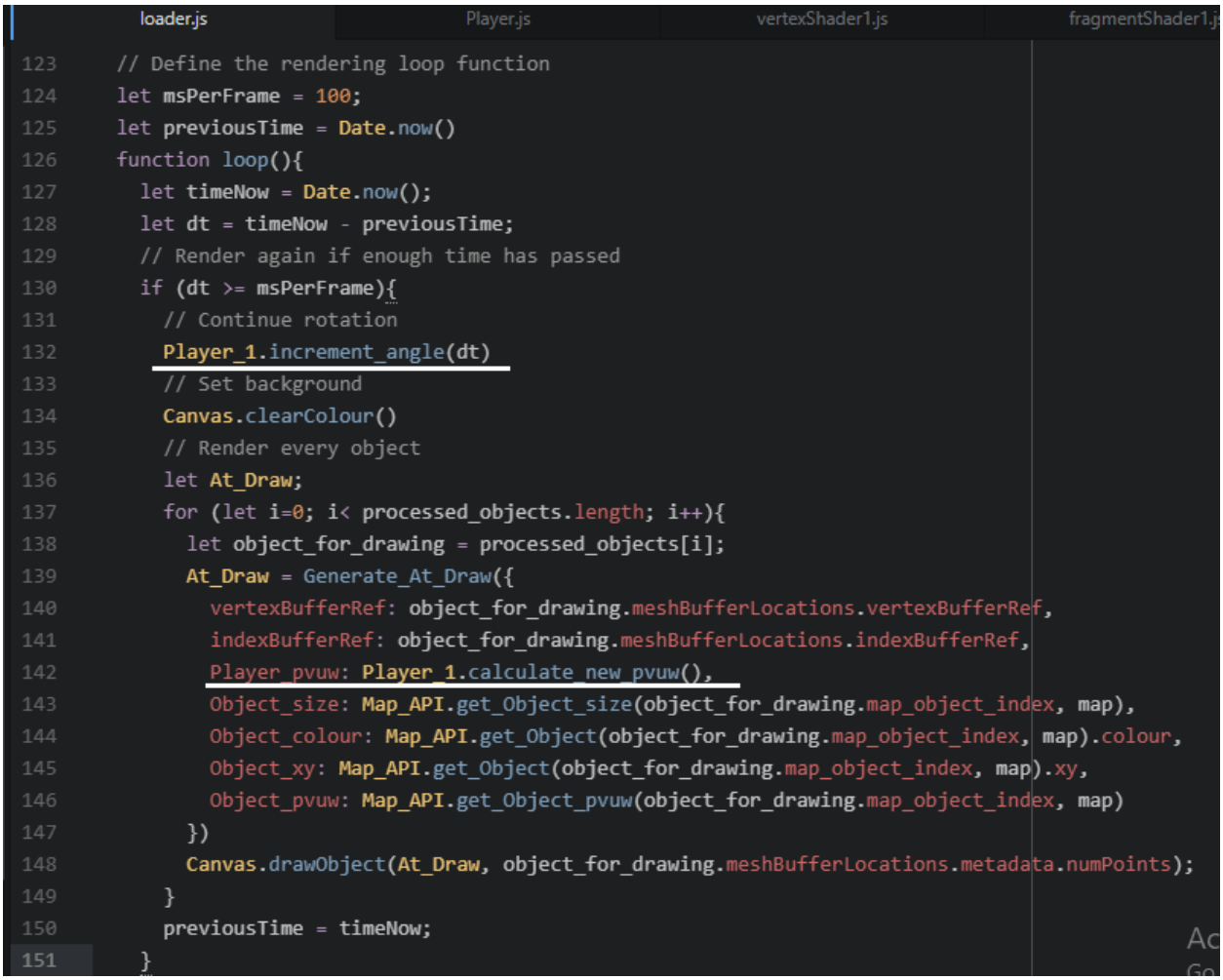
How to use the player object:

Players can start at different coordinates and use different keys to move / change orientation.

Loader.js

```
55      // Player with no mesh but acting as a camera
56      let Player_pvuw =
57      [
58        0.0, 0.0, 0.0, 1.0,
59        1.0, 0.0, 0.0, 0.0,
60        0.0, -1.0, 0.0, 0.0,
61        0.0, 0.0, -1.0, 0.0,
62      ]
63      let Player_rotation_set = [
64        ['KeyW', [1, 2]],
65        ['KeyA', [3, 1]],
66        ['KeyS', [2, 1]],
67        ['KeyD', [1, 3]],
68        ['KeyQ', [3,2]],
69        ['KeyE', [2,3]],
70        ['Space', [1, 0]]
71      ]
72      let Player_1 = new Player(Player_pvuw, Player_rotation_set)
```

Increment angle angle every frame if moving / rotating. Pass new calculated coordinates to vertex shader.



Player.js implementation

```
44  class Player {
45    constructor(pvuw_initial, rotation_set) {
46      this.pvuw_saved = pvuw_initial;
47      this.rotating = false; // not moving
48      this.rotation = [0, 1]; // forward
49      this.rotation_angle = 0.0;
50      this.rotation_speed = 0.5; // radians per second
51      for (let code_rotation of rotation_set){
52        this.add_listener_rotation_start(code_rotation[0], code_rotation[1])
53      }
54    }
55
56    // get and set pvuw
57    pvuw(){
58      return this.pvuw_saved;
59    }
60    update_pvuw(new_pvuw){
61      this.pvuw_saved = new_pvuw;
62    }
```

loader.js | Player.js | vertexShader1.js

```
65
66    add_listener_rotation_start(code, rotation){
67      document.addEventListener('keydown', (e)=>{
68        if (e.code === code){
69          if ((this.rotation !== rotation) || (this.rotating !== true)){
70            console.log("Start rotating "+code)
71            this.rotating = true;
72            this.rotation = rotation;
73          }
74        }
75      })
76      document.addEventListener('keyup', (e) => {
77        if (e.code === code){
78          console.log("Stop rotating "+ code)
79          // update to consolidate the rotation
80          let new_pvuw = this.calculate_new_pvuw()
81          this.update_pvuw(new_pvuw);
82          this.rotating = false;
83          this.rotation_angle = 0.0;
84        }
85      })
86    }
87
88    increment_angle(dt){
89      if (this.rotating){
90        let change_angle = this.rotation_speed * (1/1000) * dt;
91        this.rotation_angle += change_angle;
92        console.log(this.rotation_angle)
```

```
96      calculate_new_pvuw(){
97        let rotation = this.rotation;
98        let angle = this.rotation_angle;
99        let pvuw = this.pvuw_saved;
100       let a = get_vector_from_matrix_4D(rotation[0], pvuw)
101       let b = get_vector_from_matrix_4D(rotation[1], pvuw)
102       let [a_, b_] = rotate_4D_vectors(a, b, angle)
103       pvuw = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
104       pvuw = replace_vector_in_matrix_4D(rotation[1], b_, pvuw)
105       return pvuw;
106     }
107   }
108
109   export default Player;
```

These helper functions serve to rotate two axes about an angle. They also help to extract vectors from the pvuw matrix.

A new = Acos + Bsin

B new = Bcos – Asin

Where cos and sin represent the cosine and sine of that angle.

loader.js | Player.js | vertexShader1.js | fragmentShader1.js

```
function get_vector_from_matrix_4D(index, matrix){
  let startIndex = index * 4;
  let newVector = [matrix[startIndex], matrix[startIndex+1], matrix[startIndex+2], matrix[startIndex+3]]
  return newVector;
}
function rotate_4D_vectors(a, b, angle){
  let cos = Math.cos(angle)
  let sin = Math.sin(angle)
  // p = p*cos + v*cos
  // v = -p*sin + v*cos
  let a_ = [null, null, null, null]
  let b_ = [null, null, null, null]
  a_[0] = a[0] * cos + b[0] * sin
  a_[1] = a[1] * cos + b[1] * sin
  a_[2] = a[2] * cos + b[2] * sin
  a_[3] = a[3] * cos + b[3] * sin
  b_[0] = b[0] * cos - a[0] * sin
  b_[1] = b[1] * cos - a[1] * sin
  b_[2] = b[2] * cos - a[2] * sin
  b_[3] = b[3] * cos - a[3] * sin
  return [a_, b_]
}
function replace_vector_in_matrix_4D(index, vector, matrix){
  let startIndex = index * 4;
  matrix[startIndex] = vector[0]
  matrix[startIndex+1] = vector[1]
  matrix[startIndex+2] = vector[2]
  matrix[startIndex+3] = vector[3]
  return matrix
```


P3.2.2 Addition of distance shading

In order to shade planets by distance, I need to calculate the distances to them.

I already had the cosine of the distance (as an angle) stored in the vertex shader algorithm. I added an arcos function to calculate the distance, and save it as the depth. I also saved it as the final z-coordinate of the shape so that WebGL could order objects by depth.

I have implemented part of the taylor series of arcos here.

```
float arcos(float x){
  float Pi = 3.1415;
  float x2 = x*x;
  float arsin = (x)*(1.0 + (1.0/4.0)*(x2)*(1.0 + (1.0/2.0)*(x2)*(1.0 + (5.0/8.0)*(x2)*(1.0 + (7.0/10.0)*(x2)*(1.0)))));
  return (Pi/2.0 - arsin);
}
```

End of function getCoords_4D_modified

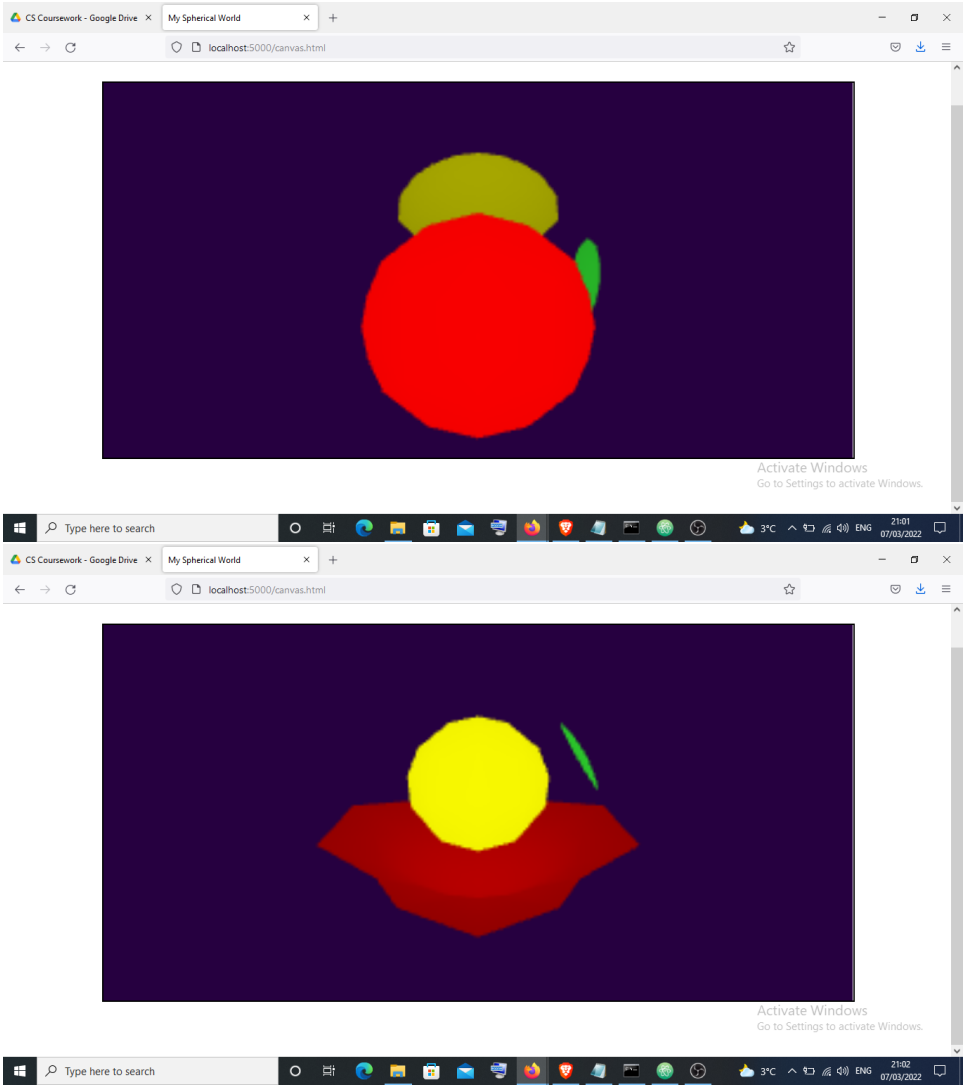
```
109             vec3 abc = invConcatAxes * camera_to_point.xyz;
110             float D = (arcos(cosD) / (Pi)) - 1.0;
111             vec4 xyzD = vec4(abc.z, abc.y, abc.x, D);
112             return xyzD;
113         }
```

```
133     void shader_4D_spherical(){
134       vec3 coords = coordinates * object_size;
135       vec4 xyzD = getCoords_4D_modified(coords, object_pvuw, camera_pvuw);
136       vec3 xyz = xyzD.xyz;
137       float D = xyzD.w;
138       vec3 xyz_scale = scaleCoordsToCanvas(xyz);
139       xyz_scale.z = (D+1.0)/2.0;
140       gl_Position = vec4(xyz_scale, 1.0);
141       depth = D;
142     }
```

Fragment shader as a reminder of distance shading algorithm.

```
13    void shade_by_distance_using_depth_factor(){
14      float k = depthFactor;
15      float depth2 = (1.0-k) + (k)*(1.0-((depth + 1.0) / 2.0)); // From 0.0 to 1.0
16      gl_FragColor = vec4(depth2 * object_colour.xyz, object_colour.w);
17    }
18
```

The result is not too clear but shapes are being shaded. Further investigation is needed.



P3.2.3 Testing and Alterations

I experienced quite a few bugs after testing the program. Since the image is now moving, I have decided a video format is best for this test evidence.

Initial Simulation Video

Evidence: https://drive.google.com/file/d/13iPSX6yFekvQv7w55BRqdHvtsiOwrZnr/view?usp=sharing

Bug A

Problem: The view starts changing too fast if a rotation or movement key is held down.

Evidence:
https://drive.google.com/file/d/1xQzhadGmVfOVVo71VMSSDfIfteyqmyNM/view?usp=sharing

Fixed Evidence: https://drive.google.com/file/d/1JBaKHswfMMhm5cQ4Bj92-LRPNHslj-rA/view?usp=sharing

Code change: resetting rotation every frame.

```
96     calculate_new_pvuw(){
97       let rotation = this.rotation;
98       let angle = this.rotation_angle;
99       let pvuw = this.pvuw_saved;
100      let a = get_vector_from_matrix_4D(rotation[0], pvuw)
101      let b = get_vector_from_matrix_4D(rotation[1], pvuw)
102      let [a_, b_] = rotate_4D_vectors(a, b, angle)
103      pvuw = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
104      pvuw = replace_vector_in_matrix_4D(rotation[1], b_, pvuw)
105      return pvuw;
106    }
107  }
108
109  export default Player;
```

Changed above to below. (Player.js)

```
94     calculate_new_pvuw(){
95       let rotation = this.rotation;
96       let angle = this.rotation_angle;
97       let pvuw = this.pvuw_saved;
98       let a = get_vector_from_matrix_4D(rotation[0], pvuw)
99       let b = get_vector_from_matrix_4D(rotation[1], pvuw)
100      let [a_, b_] = rotate_4D_vectors(a, b, angle)
101      pvuw = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
102      pvuw = replace_vector_in_matrix_4D(rotation[1], b_, pvuw)
103      this.pvuw_saved = pvuw;
104      this.rotation_angle = 0.0;
105      return pvuw;
106    }
107  }
```

Rotation speed test

Purpose: find out if changing rotation speed in code reflects in the simulation.

Successful Evidence:
https://drive.google.com/file/d/1qLKGjaXo2yd3uFXX4pOA9m7qZaj2cafK/view?usp=sharing

Result: It does.

Bug B

Problem: When the shape is too close, vertices are cut off as they fall off the canvas.

Evidence: https://drive.google.com/file/d/1oYz9loYSFWD8qWKEn8zbPPDCqluXEyD7/view?usp=sharing

Solution (planned): **Implement collision detection to prevent viewer getting too close to planet.**

Bug C

Problem: Planets appear to all be within a confined circle. No perspective is seen.

Evidence: https://drive.google.com/file/d/1EaV_MwFAluIEiR9emrffHZFgstuMznl6/view?usp=sharing

Solution: Increase the canvas scale factor to fully submerge into the view.

Evidence of fix:
https://drive.google.com/file/d/1cntHUsu4iD2tbg_CUtL6rj2CrXhu2Oby/view?usp=sharing

Bug D

Problem: Objects appear to be in front of you, even though they are behind you, and obscure other objects.

Evidence: https://drive.google.com/file/d/1sviGykmI4bCtbMquKurh2VUmJOGS23_C/view?usp=sharing

Explanation: The function arcos, by itself, can only tell you the absolute distance to the shape – not whether it is behind you or not. Another calculation needs to be done to determine this.

Solution:

```
89          vec4 C_Pos4D = camera_pvuw[0].xyzw;
90          vec4 P_Pos4D = Vertex_4D_coordinates;
91          // Calculating angular distance D
92          cosD = dot(C_Pos4D, P_Pos4D);
93          sinD = sqrt(1.0-cosD*cosD);
94          vec4 camera_to_point = (P_Pos4D - (C_Pos4D * cosD))/( sinD );
95          bool behind = false;
96          if (dot(camera_to_point, camera_pvuw[1].xyzw) < 0.0) {
97            // Facing wrong way
98            behind = true;
99          }
100
101         // Calculate the 3D direction from player to object
102         mat3 concatenated_axes;
103         concatenated_axes[0] = camera_pvuw[1].xyz;
104         concatenated_axes[1] = camera_pvuw[2].xyz;
105         concatenated_axes[2] = camera_pvuw[3].xyz;
106         mat3 invConcatAxes = inverse(concatenated_axes);
107         // Correspond to forward, right and up axes v, u, w.
108         vec3 abc = invConcatAxes * camera_to_point.xyz;
109         float D;
110         if (behind){
111           D = ( 2.0 - (arcos(cosD) / (Pi)) ) - 1.0;
112         } else {
113           D = (arcos(cosD) / (Pi)) - 1.0;
114         }
115         vec4 xyzD = vec4(abc.z, abc.y, abc.x, D);
116         return xyzD;
117     }
```

Successful evidence:
https://drive.google.com/file/d/1ZJsc_fl6sRURrK31Ec0bGAbR7nAKqb78/view?usp=sharing

Bug E

Problem: Forward key makes you go back.

Evidence: https://drive.google.com/file/d/1yU7scYcM1UbH7Q6E_QkcNVEDuU7HyZnQ/view?usp=sharing

Successful fix: https://drive.google.com/file/d/1yCwqKZJAh5tCBt0eUk54a7-PJ5KE2aoW/view?usp=sharing

Updated rotation set:

```
63      let Player_rotation_set = [
64        ['KeyW', [1, 2]],
65        ['KeyA', [3, 1]],
66        ['KeyS', [2, 1]],
67        ['KeyD', [1, 3]],
68        ['KeyQ', [3,2]],
69        ['KeyE', [2,3]],
70        ['Space', [0, 1]]
71      ]
```

Bug F

Problem: When inside a sphere, it is not shown from inside. Instead, its faraway outside surface is shown. Other objects appear although they should be blocked from view by the object walls.

Evidence: https://drive.google.com/file/d/1acChyc_IKEvAxbHHAT-E83CNePmkvqYo/view?usp=sharing

Solution: **Introduce collision detection to prevent the player from being inside a sphere.**

P3.3 Conclusion of iteration

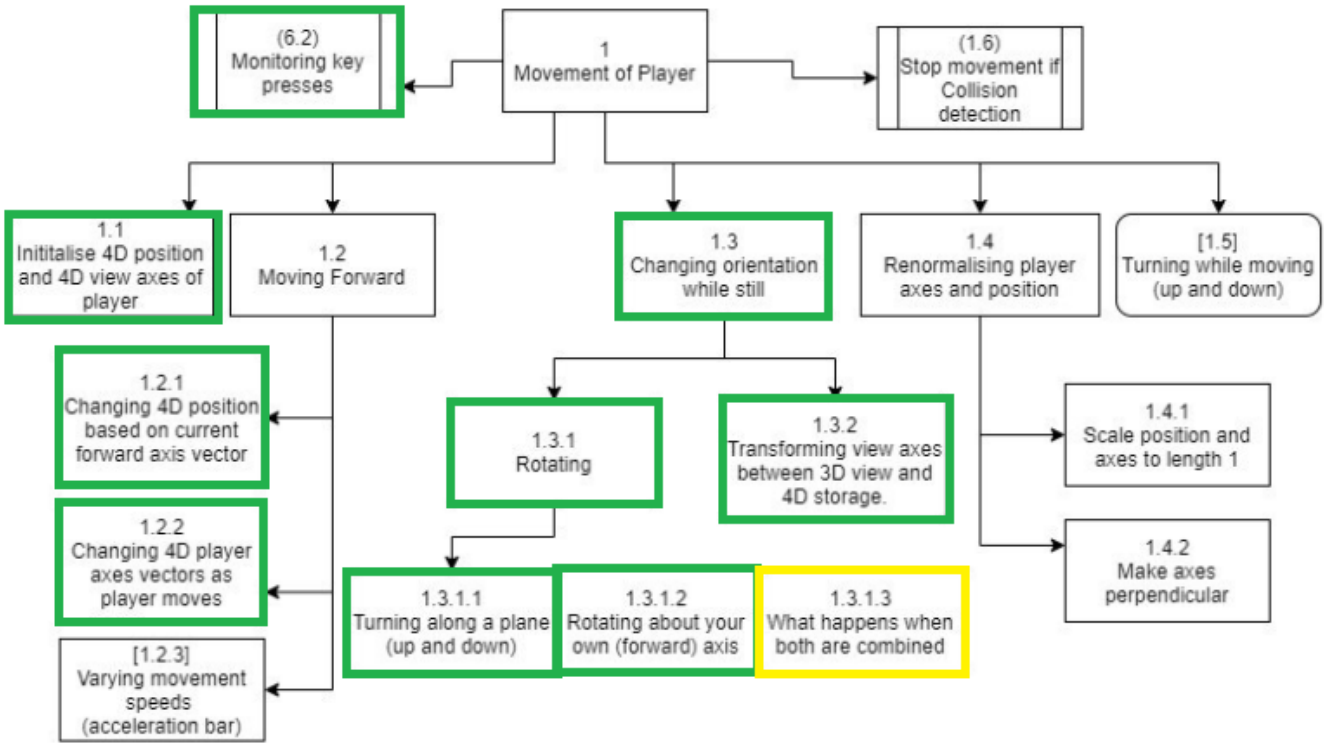
This iteration went quite well.

The player can now use the keys QWEASD and Space to move and change orientation.

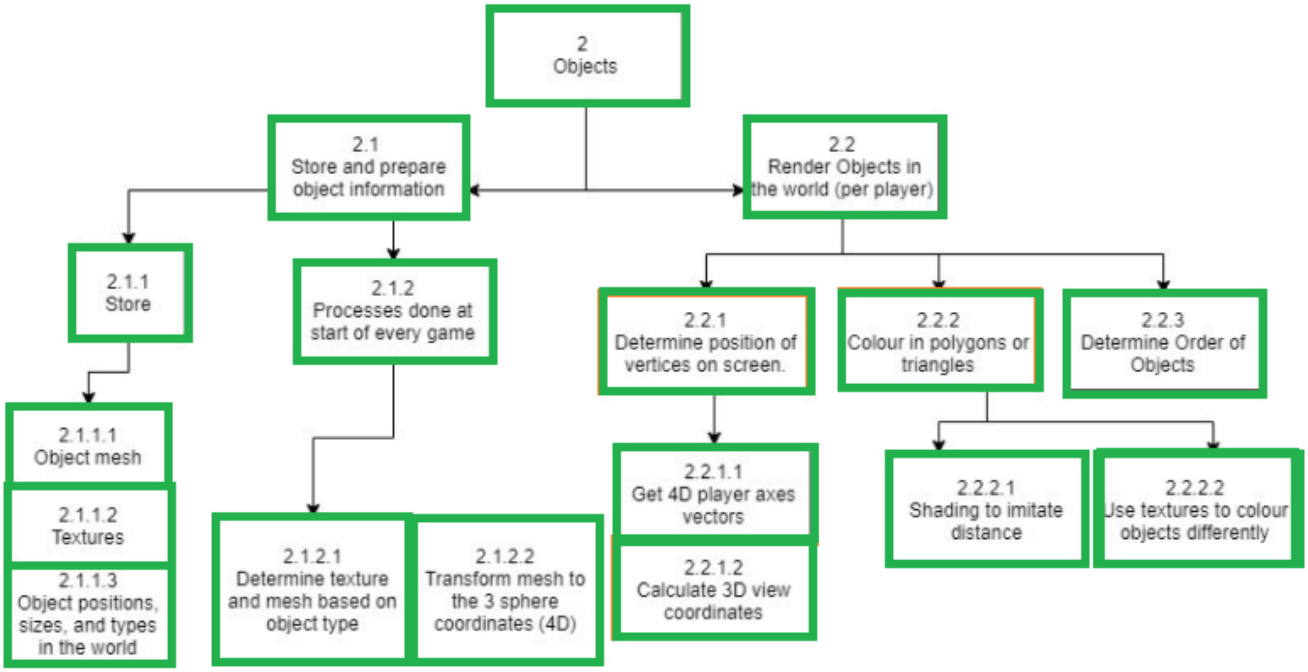
Some bugs are still present. For example, the planet merging bug.

I hope to fix some of these bugs by implementing collision detection in the next stage.

Here are the parts I have completed so far:

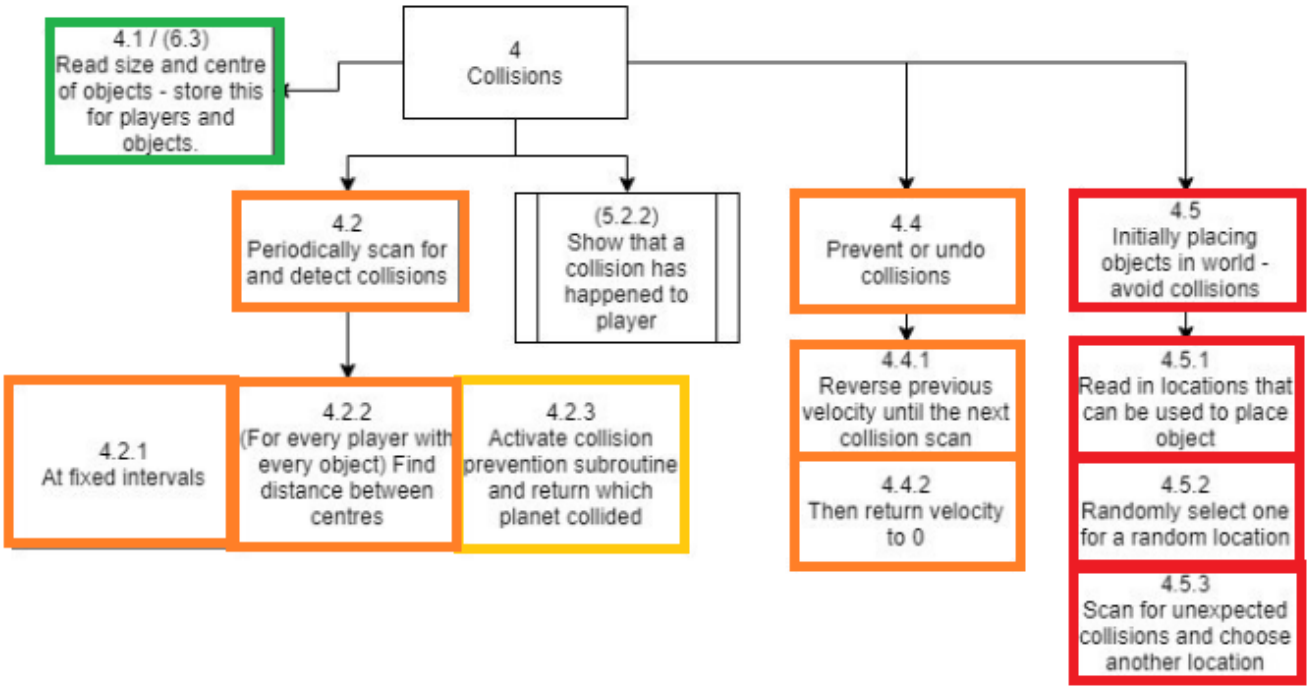


Note in 1.3.1.3: When rotation keys are combined, one type of rotation is dominant. Rotations cannot be combined.


2
Objects
2.1
Store and prepare object information
2.2
Render Objects in the world (per player)
2.1.1
Store
2.1.2
Processes done at start of every game
2.2.1
Determine position of vertices on screen.
2.2.2
Colour in polygons or triangles
2.2.3
Determine Order of Objects
2.1.1.1
Object mesh
2.1.1.2
Textures
2.1.1.3
Object positions, sizes, and types in the world
2.1.2.1
Determine texture and mesh based on object type
2.1.2.2
Transform mesh to the 3 sphere coordinates (4D)
2.2.1.1
Get 4D player axes vectors
2.2.1.2
Calculate 3D view coordinates
2.2.2.1
Shading to imitate distance
2.2.2.2
Use textures to colour objects differently

P4 Collision Detection

Key: **Completed** **To Do** **Changed** **Omitted**



Change to 4.2.3: I believe it is not necessary to return which planet collided for the purposes of this program.

Change to 4.5: I do not have enough time to program this collision prevention part, as I need to complete the base functionality of the program.

P4.1 Implementation

```
84    // Moves player with set rotation, returns new pvuw of player,
85    calculate_new_pvuw_and_confirm_movement(){
86      let pvuw_previous = this.pvuw_saved;
87      // Calculate rotation
88      let pvuw_new = this.calculate_pvuw_after_set_rotation();
89      this.pvuw_saved = pvuw_new;
90
91      // Check for collisions with planets
92      let collided = this.check_for_collisions(pvuw_new);
93      if (collided === true) {
94        // Reset to pvuw before collision.
95        this.rotation = [this.rotation[1], this.rotation[0]];
96        this.pvuw_saved = pvuw_previous;
97        // Display collision warning
98        this.Collision_display_function();
99        setTimeout(()=>{this.rotating = false}, 500)
100
101       return pvuw_previous;
102
103     } else {
104       this.rotation_angle = 0.0;
105       this.pvuw_saved = pvuw_new;
106       return pvuw_new;
107     }
108   }
```

Note: lines 98 and 99 were added later, after a mid-development prototype feedback (see next section).

```
130     check_for_collisions (player_pvuw){
131       let list_of_pvuws = this.map_list_of_pvuws;
132       let list_of_sizes = this.map_list_of_sizes;
133       for (let planet_index = 0; planet_index< list_of_pvuws.length; planet_index++){
134           let o_pvuw = list_of_pvuws[planet_index];
135           let o_size = list_of_sizes[planet_index];
136           let collided = this.check_for_collision(o_pvuw, player_pvuw, o_size, this.player_size);
137           if (collided){
138             return true
139           }
140       }
141       return false
142     }
143
```

```
110     calculate_pvuw_after_set_rotation(){
111       return calculate_new_pvuw(
112         this.rotation,
113         this.rotation_angle,
114         this.pvuw_saved
115       );
116     }
117
118     check_for_collision(object_pvuw, player_pvuw, object_size, player_size){
119       // sizes are radius where 1.0 is 90 degree turn
120       let cosD = object_pvuw[0]*player_pvuw[0] + object_pvuw[1]*player_pvuw[1] + object_pvuw[2]*
121       let min_D = (object_size + player_size) * Pi;
122       let max_cosD = Math.cos(min_D);
123       if (cosD < max_cosD){
124         return false;
125       } else {
126         return true;
127       }
128     }
```

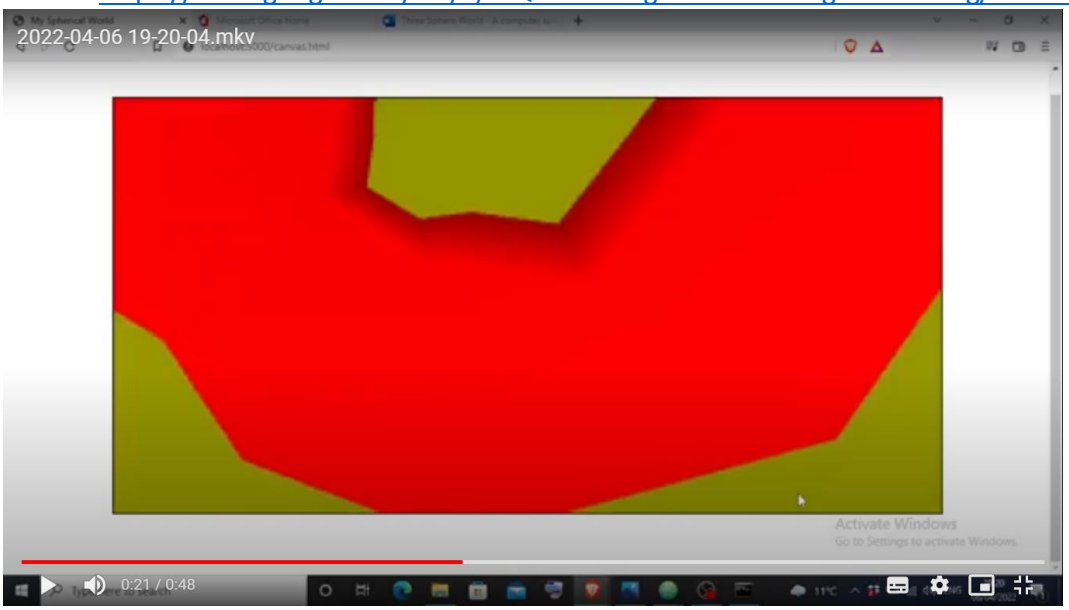
To allow me to also use rotation functions later in other places in my code, I have refactored them to be exported from a utils.js file:

```
function get_vector_from_matrix_4D(index, matrix){
  let startIndex = index * 4;
  let newVector = [matrix[startIndex], matrix[startIndex+1], matrix[startIndex+2], matrix[startIndex+3]]
  return newVector;
}

function rotate_4D_vectors(a, b, angle){
  let cos = Math.cos(angle)
  let sin = Math.sin(angle)
  // p = p*cos + v*cos
  // v = -p*sin + v*cos
  let a_ = [null, null, null, null]
  let b_ = [null, null, null, null]
  a_[0] = a[0] * cos + b[0] * sin
  a_[1] = a[1] * cos + b[1] * sin
  a_[2] = a[2] * cos + b[2] * sin
  a_[3] = a[3] * cos + b[3] * sin
  b_[0] = b[0] * cos - a[0] * sin
  b_[1] = b[1] * cos - a[1] * sin
  b_[2] = b[2] * cos - a[2] * sin
  b_[3] = b[3] * cos - a[3] * sin
  return [a_, b_]
}
```

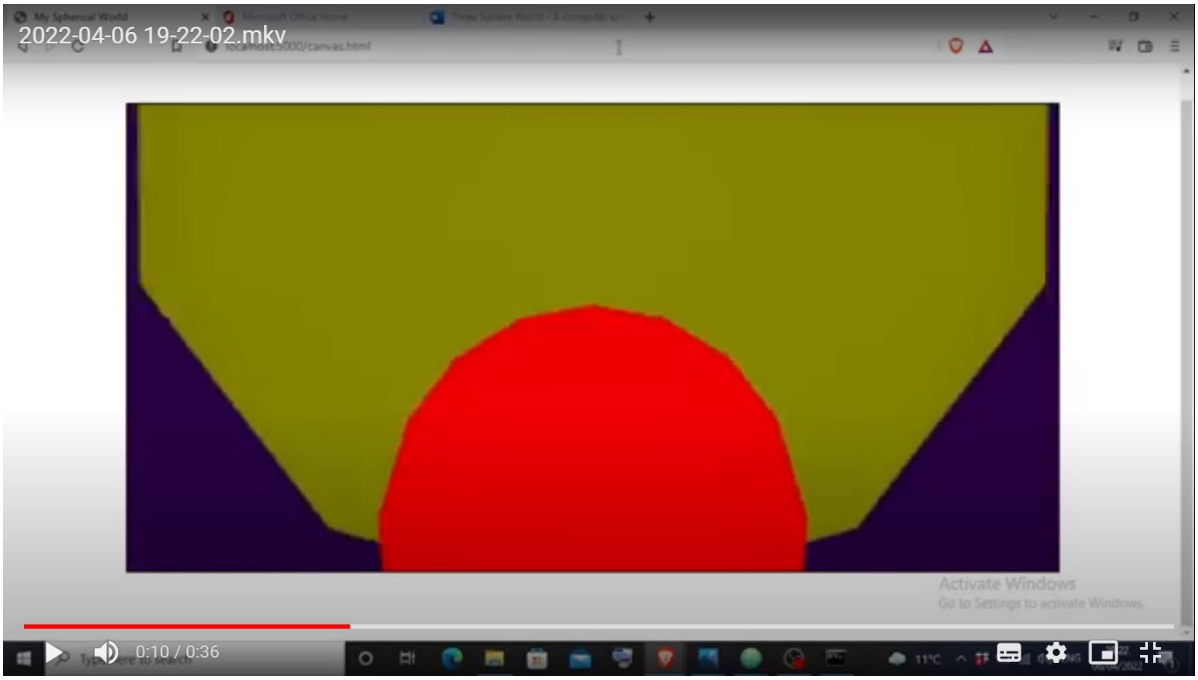
```
function replace_vector_in_matrix_4D(index, vector, matrix){
  let startIndex = index * 4;
  matrix[startIndex] = vector[0]
  matrix[startIndex+1] = vector[1]
  matrix[startIndex+2] = vector[2]
  matrix[startIndex+3] = vector[3]
  return matrix
}

function calculate_new_pvuw(rotation, angle, pvuw){
  let a = get_vector_from_matrix_4D(rotation[0], pvuw)
  let b = get_vector_from_matrix_4D(rotation[1], pvuw)
  let result = rotate_4D_vectors(a, b, angle)
  let a_ = result[0];
  let b_ = result[1];
  let pvuw2 = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
  pvuw2 = replace_vector_in_matrix_4D(rotation[1], b_, pvuw2)
  return pvuw2;
}

export {rotate_4D_vectors, replace_vector_in_matrix_4D, get_vector_from_matrix_4D, calculate_new_pvuw};

```

Here are the results. I have recorded two videos to show before and after.

Before: https://drive.google.com/file/d/1IQ-dOX0Rmg7Jb2MswYk6JgZ9h44YcPng/view?usp=sharing



After: https://drive.google.com/file/d/1mGG5cwjIgVqZOD6OD4fyBnIxa24RoPQk/view?usp=sharing



P4.2 Prototype Tests

I have decided to conduct some prototype user tests, to see how the user might feel about the program at its current stage.

Lily agreed to test out the program and here's what happened.

- She quite liked navigating the world.
- She was confused at stages – she thought that she was bouncing off objects when in fact the object was growing, then shrinking due to the 4D sphere effect.
- The actions corresponding to different key needed some explaining.
- She suggested that, as an idea, something cool should happen when all keys are pressed, for example, colours being inverted.

Conclusions:

- I think the inclusion of a collision warning effect would be a more effective way of telling people they have collided.
- I think the display of the rocket would make navigation and movement easier to follow.
- The last idea, while cool, does not come under the base functionality of the project. It could be included as a later update of the program, if the project continues.

P4.3 Additions

I have decided to add the collision overlay.

For that, every time there is a collision, an external function needs to be called, to trigger the HTML UI change. After some time, the UI needs to revert to normal. This is the job of the onCollision function.

```
21    <body>
22      <div id='collision_background'>
23        <div id='WebGL_Fallback' hidden='true' class='info_yellow'>
24          Web GL is not supported by you or your browser.
25          Please give it some support. A kind word or two will make him feel happier.
26        </div>
27        <canvas id='canvas'></canvas>
28      </div>
29    </body>
```

makeasphe... | fragmentSh... | loader.js | styling.css

```
1
2   canvas#canvas{
3     height: 80%;
4     width: 80%;
5     margin: 5% 10%;
6     border: 2px solid black;
7   }
8
9   .info_yellow{
10    width: 80%;
11    margin: auto 8%;
12    padding: 2%;
13    background: #eeee77;
14    border: solid 1px #dddd66;
15    border-radius: 5px;
16  }
17
18  .background-collided{
19    background-color: #ffcccc;
20  }
```

In loader.js.

```
186  function onCollision(){
187    let background = document.getElementById('collision_background');
188    background.classList.add('background-collided');
189    setTimeout(()=>{background.classList.remove('background-collided')},1000);
190  }
```

```
87     let Player_1 = new
88     Player (
89       Player_pvuw,
90       Player_size,
91       Player_rotation_set,
92       map,
93       use_collision_overlay?(onCollision):(()=>{}));
```

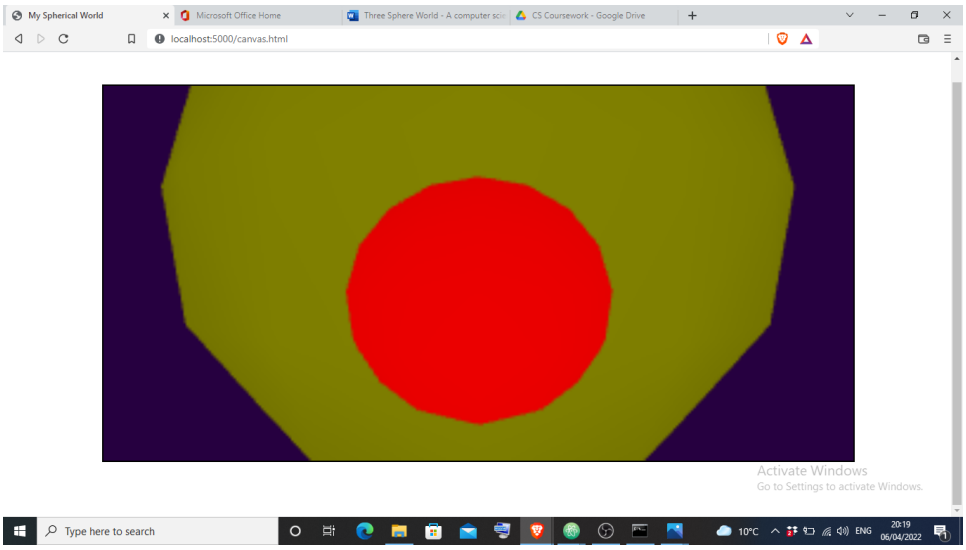
In Player class constructor method.

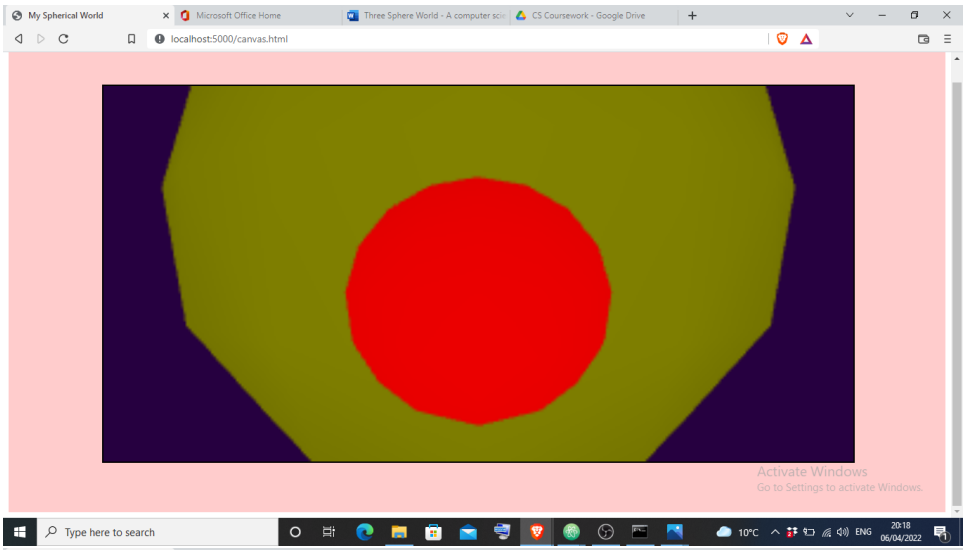
```
class Player {
  constructor(pvuw_initial, player_size, rotation_set, map, Collision_display_function) {
    // 1.0 is Pi
    this.player_size = player_size;
    this.pvuw_saved = pvuw_initial;
    // Store object coordinates and sizes on map for collision detection
    this.map_list_of_pvuws = Map_API.get_all_Object_pvuws(map);
    this.map_list_of_sizes = Map_API.get_all_Object_sizes(map);
    if(this.map_list_of_pvuws.length !== this.map_list_of_sizes.length){
      console.error('Something in the code is going wrong.');
    }
    this.Collision_display_function = Collision_display_function;
```

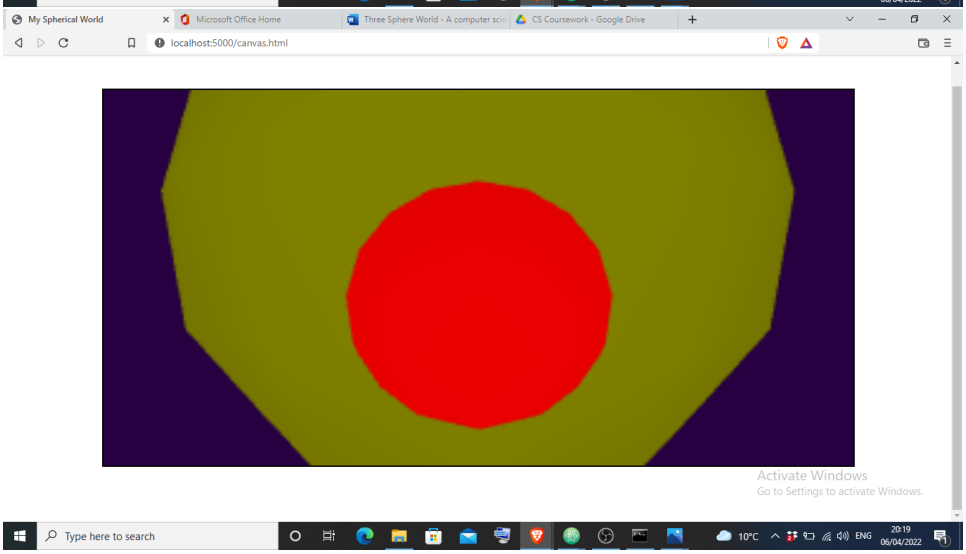
In Player class calculate_new_pvuw_and_confirm_movement method.

```
90
91       // Check for collisions with planets
92       let collided = this.check_for_collisions(pvuw_new);
93       if (collided === true) {
94         // Reset to pvuw before collision.
95         this.rotation = [this.rotation[1], this.rotation[0]];
96         this.pvuw_saved = pvuw_previous;
97         // Display collision warning
98         this.Collision_display_function();
99         setTimeout(()=>{this.rotating = false}, 1000)
100
```

The result is as follows. (Before collision, just after collision, after collision)


My Spherical World
Microsoft Office Home
Three Sphere World - A computer sci
CS Coursework - Google Drive
localhost:5000/canvas.html
Activate Windows
Go to Settings to activate Windows.
Type here to search
10°C
ENG
20:19
06/04/2022


My Spherical World
Microsoft Office Home
Three Sphere World - A computer sci
CS Coursework - Google Drive
localhost:5000/canvas.html
Activate Windows
Go to Settings to activate Windows.
Type here to search
10°C
ENG
20:18
06/04/2022


My Spherical World
Microsoft Office Home
Three Sphere World - A computer sci
CS Coursework - Google Drive
localhost:5000/canvas.html
Activate Windows
Go to Settings to activate Windows.
Type here to search
10°C
ENG
20:19
06/04/2022

Video

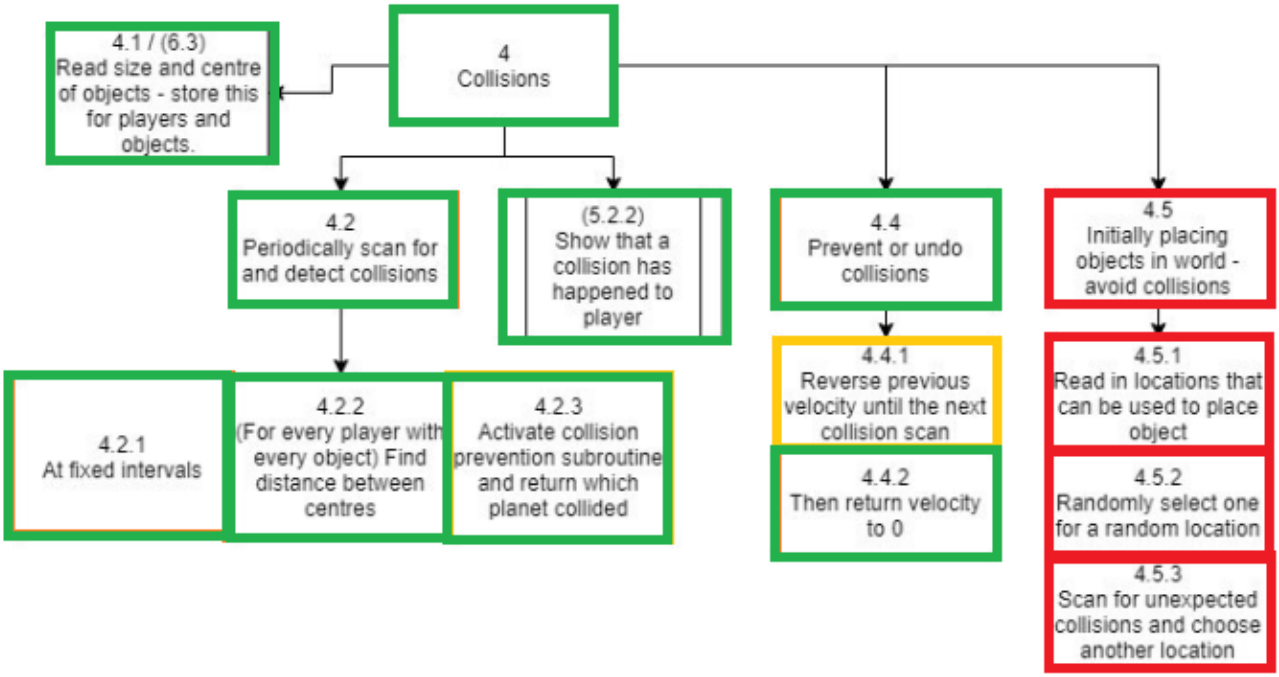
I have recorded a video showing it in action.

Collision overlay:
https://drive.google.com/file/d/1_bWgycGycaUTk0q4zYWa4cBs0RBwrNHj/view?usp=sharing

P4.4 Iteration Evaluation

The collision detection system has been implemented.

Collision prevention upon initial placing of player has not been developed.



Change to 4.4.1: The velocity reverses for 1000ms = 1 second, instead of until the next collision scan.

In the next iteration, I plan to make a rocket object, to indicate the player's position and movement.

P5 – Rocket

In this iteration, I plan to develop a rocket object that will show the player how they are moving.

P5.1 Implementation

The first part I needed was a new type of object mesh – a rocket. I have made a new file /models/rocket.js

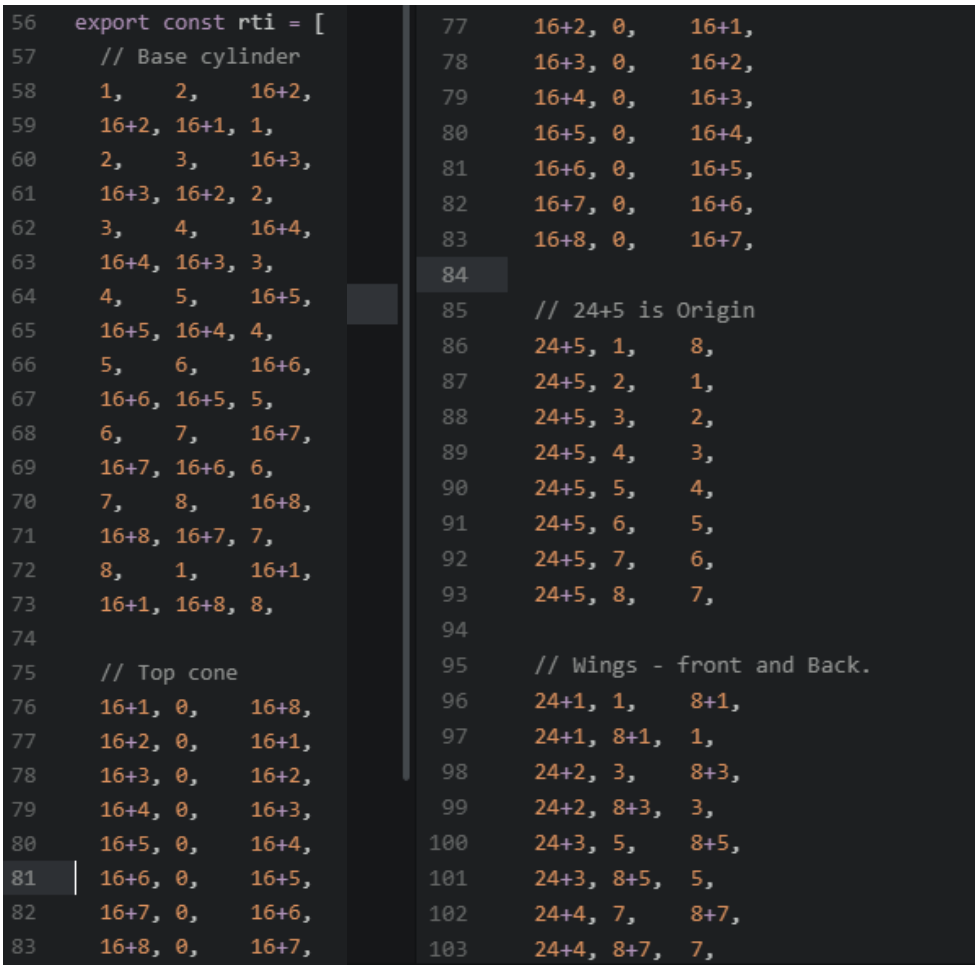
I have then hardcoded the vertex positions and triangles, based on parameters that the rocket can have.

loader.js | **rocket.js** | canvas.html | Player.js

```
1
2   /*
3
4   A: the axial length of rocket.
5
6   B: The width or radius of the rocket.
7
8   C: B divided by the square root of two.
9
10  D: The axial distance to wing attachment.
11
12  E: The axial distance to the cone part of the rocket.
13
14  F: The wingspan of the rocket.
15
16  */
```

```
1
2   let a = 1.0;
3   let b = 0.2;
4   let c = b * 0.707;
5   let d = 0.2;
6   let e = 0.4;
7   let f = b + (b*0.25);
8   let o = 0.0;
9
10
11  export const rv = [
12      // Centre point
13       o,   o,  a,
14
15      // Back octagon
16       b,   o,  o,
17       c,   c,  o,
18       o,   b,  o,
19      -c,   c,  o,
20      -b,   o,  o,
21      -c,  -c,  o,
22       o,  -b,  o,
23       c,  -c,  o,
```

```
// Middle Octagon
 b,   o,  d,
 c,   c,  d,
 o,   b,  d,
-c,   c,  d,
-b,   o,  d,
-c,  -c,  d,
 o,  -b,  d,
 c,  -c,  d,

// Far octagon
 b,   o,  e,
 c,   c,  e,
 o,   b,  e,
-c,   c,  e,
-b,   o,  e,
-c,  -c,  e,
 o,  -b,  e,
 c,  -c,  e,

// Winglets
 f,   o,  o,
 o,   f,  o,
-f,   o,  o,
 o,  -f,  o,

 // Origin
 o,   o,  o
```



```
export const rti = [
  // Base cylinder
  1,    2,    16+2,
  16+2, 16+1, 1,
  2,    3,    16+3,
  16+3, 16+2, 2,
  3,    4,    16+4,
  16+4, 16+3, 3,
  4,    5,    16+5,
  16+5, 16+4, 4,
  5,    6,    16+6,
  16+6, 16+5, 5,
  6,    7,    16+7,
  16+7, 16+6, 6,
  7,    8,    16+8,
  16+8, 16+7, 7,
  8,    1,    16+1,
  16+1, 16+8, 8,

  // Top cone
  16+1, 0,    16+8,
  16+2, 0,    16+1,
  16+3, 0,    16+2,
  16+4, 0,    16+3,
  16+5, 0,    16+4,
  16+6, 0,    16+5,
  16+7, 0,    16+6,
  16+8, 0,    16+7,

  16+2, 0,    16+1,
  16+3, 0,    16+2,
  16+4, 0,    16+3,
  16+5, 0,    16+4,
  16+6, 0,    16+5,
  16+7, 0,    16+6,
  16+8, 0,    16+7,

  // 24+5 is Origin
  24+5, 1,    8,
  24+5, 2,    1,
  24+5, 3,    2,
  24+5, 4,    3,
  24+5, 5,    4,
  24+5, 6,    5,
  24+5, 7,    6,
  24+5, 8,    7,

  // Wings - front and Back.
  24+1, 1,    8+1,
  24+1, 8+1,  1,
  24+2, 3,    8+3,
  24+2, 8+3,  3,
  24+3, 5,    8+5,
  24+3, 8+5,  5,
  24+4, 7,    8+7,
  24+4, 8+7,  7,
```

In the Player class, a new method was added .

It computes the position of the rocket relative to the camera, by turning it down, moving it forward, then turning it up. (lines 146-148)

Lines 150-157 are used to further rotate the rocket to indicate a turning movement.

```
144     determineRocketPVUW(){
145       let rocketPVUW = this.pvuw_saved;
146       rocketPVUW = calculate_new_pvuw([2, 1], Pi * 0.04, rocketPVUW);
147       rocketPVUW = calculate_new_pvuw([0, 1], Pi * 0.05, rocketPVUW);
148       rocketPVUW = calculate_new_pvuw([1, 2], Pi * 0.05, rocketPVUW);
149
150       if (this.rotating){
151         if (this.rotation[0] === 0 || this.rotation[1] === 0){
152           // Forwards or backwards
153           rocketPVUW = calculate_new_pvuw([1,0], Pi*0.01, rocketPVUW);
154         } else {
155           rocketPVUW = calculate_new_pvuw(this.rotation, Pi*0.12, rocketPVUW);
156         }
157       }
158
159       rocketPVUW = calculate_new_pvuw([3, 1], Pi * 0.5, rocketPVUW);
160       return rocketPVUW;
161     }
162
163   }
164
165   export default Player;
```

loader.js | rocket.js | canvas.html | Player.js | Map_API.js

```
61    // Add an additional rocket object to the world
62    let Rocket_Mesh_Locations;
63    if (use_rocket){
64      Canvas.add_Indexed_Mesh('rocket', RocketVertices, RocketTriangles);
65      Rocket_Mesh_Locations = Canvas.findMesh('rocket');
66    }
67
```

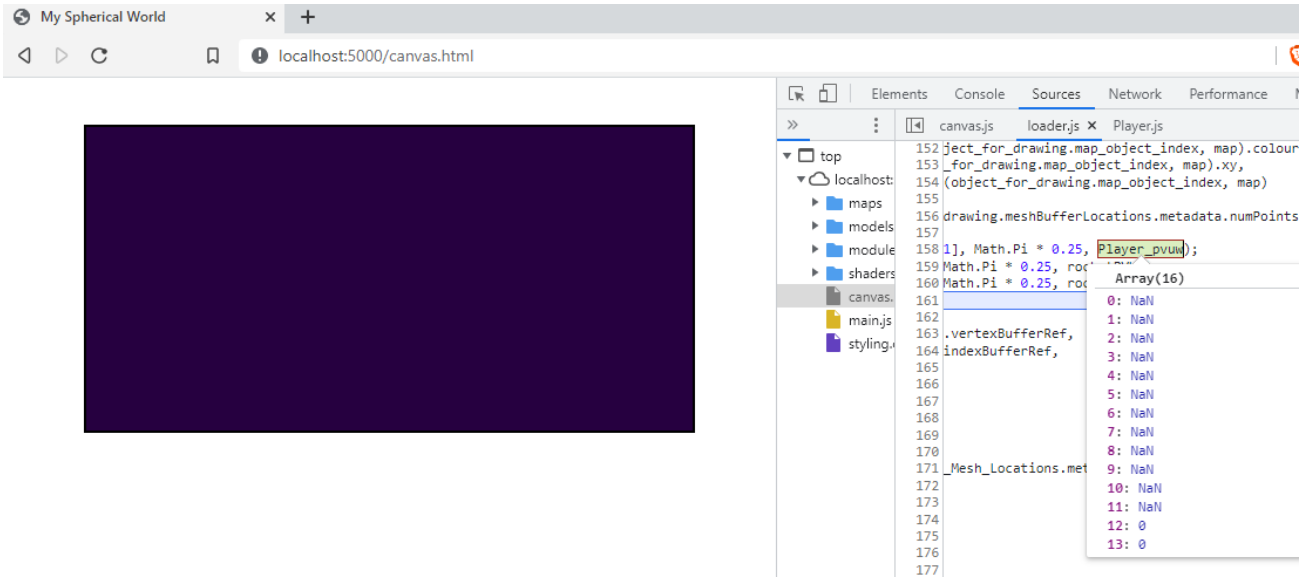
```
159         for (let i=0; i< processed_objects.length; i++){
160           let object_for_drawing = processed_objects[i];
161           At_Draw = Generate_At_Draw({
162             vertexBufferRef: object_for_drawing.meshBufferLocations.vertexBufferRef,
163             indexBufferRef: object_for_drawing.meshBufferLocations.indexBufferRef,
164             Player_pvuw: Player_pvuw,
165             Object_size: Map_API.get_Object_size(object_for_drawing.map_object_index, map),
166             Object_colour: Map_API.get_Object(object_for_drawing.map_object_index, map).colour,
167             Object_xy: Map_API.get_Object(object_for_drawing.map_object_index, map).xy,
168             Object_pvuw: Map_API.get_Object_pvuw(object_for_drawing.map_object_index, map)
169           })
170           Canvas.drawObject(At_Draw, object_for_drawing.meshBufferLocations.metadata.numPoints);
171         }
172         if (use_rocket){
173           let rocketPVUW = Player_1.determineRocketPVUW();
174           let At_Draw_rocket = Generate_At_Draw({
175             vertexBufferRef: Rocket_Mesh_Locations.vertexBufferRef,
176             indexBufferRef: Rocket_Mesh_Locations.indexBufferRef,
177             Player_pvuw: Player_pvuw,
178             Object_size: Rocket_size,
179             Object_xy: [0.0, 0.0],
180             Object_colour: [0.9,0.5,0.0,1.0],
181             Object_pvuw: rocketPVUW,
182           })
183           Canvas.drawObject(At_Draw_rocket, Rocket_Mesh_Locations.metadata.numPoints);
184         }
185         previousTime = timeNow;
```

I had to add another draw call as you see from lines 172-184.

P5.2 Testing

While testing, I found a bug:

It was a **runtime error**, and lead to NaN (Not A Number, equivalent to null) values being stored in Player_pvuw and further in rocketPVUW.



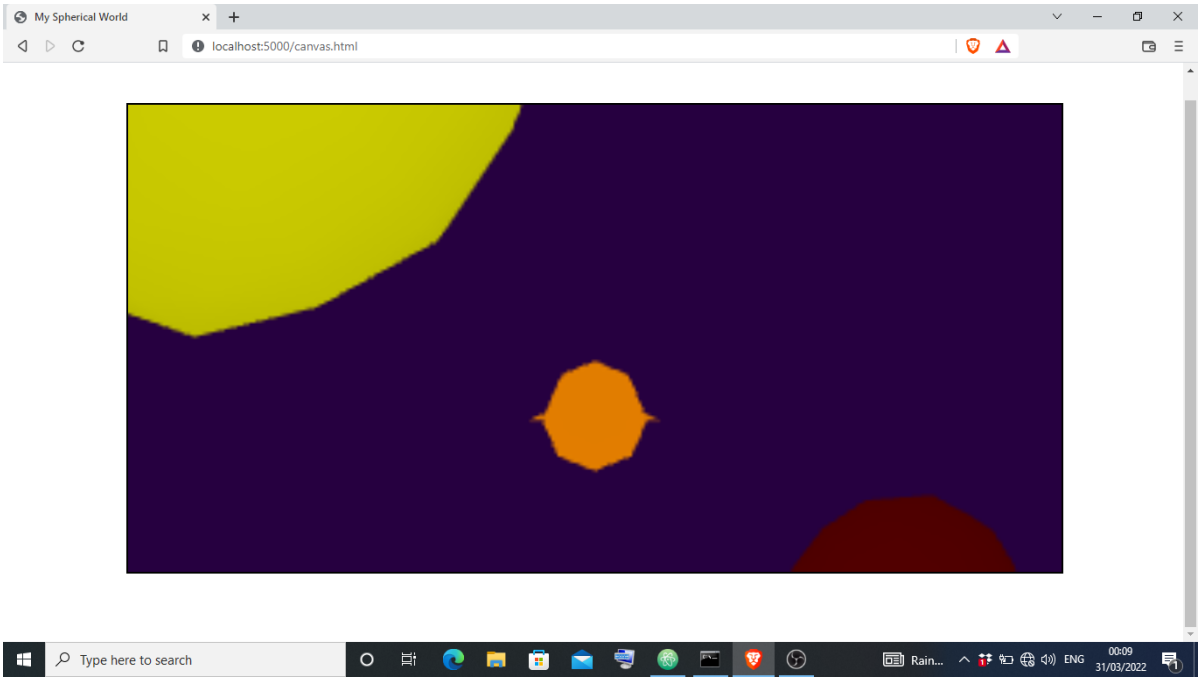
I realised that this was because the function replace_vector_in_matrix_4D was modifying the matrix by reference, not by value. It was passing back the reference to the original matrix as its output, leading to further unexpected modifications.

```
26  function replace_vector_in_matrix_4D(index, vector, matrix){
27    let startIndex = index * 4;
28    matrix[startIndex] = vector[0]
29    matrix[startIndex+1] = vector[1]
30    matrix[startIndex+2] = vector[2]
31    matrix[startIndex+3] = vector[3]
32    return matrix
33  }
34
35  function calculate_new_pvuw(rotation, angle, pvuw){
36    let a = get_vector_from_matrix_4D(rotation[0], pvuw)
37    let b = get_vector_from_matrix_4D(rotation[1], pvuw)
38    let result = rotate_4D_vectors(a, b, angle)
39    let a_ = result[0];
40    let b_ = result[1];
41    let pvuw2 = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
42    pvuw2 = replace_vector_in_matrix_4D(rotation[1], b_, pvuw2)
43    return pvuw2;
44  }
45
46  export {rotate_4D_vectors, replace_vector_in_matrix_4D, get_vector_from_matrix_4D, calculate_new_pvuw};
47
```

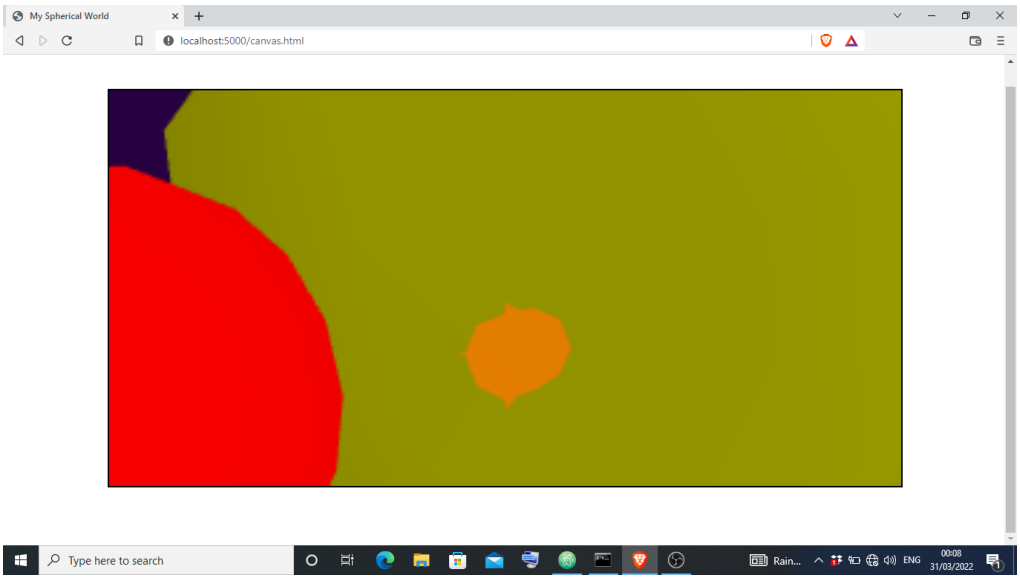
A deep copy of the array was needed to correct this. Look at lines 28-31.

```
function replace_vector_in_matrix_4D(index, vector, mat){
  // Copy contents to new matrix
  let matrix = [ mat[0],  mat[1],  mat[2],  mat[3],
                 mat[4],  mat[5],  mat[6],  mat[7],
                 mat[8],  mat[9],  mat[10], mat[11],
                 mat[12], mat[13], mat[14], mat[15] ]

  let startIndex = index * 4;
  matrix[startIndex] = vector[0]
  matrix[startIndex+1] = vector[1]
  matrix[startIndex+2] = vector[2]
  matrix[startIndex+3] = vector[3]
  return matrix
}

function calculate_new_pvuw(rotation, angle, pvuw){
  let a = get_vector_from_matrix_4D(rotation[0], pvuw)
  let b = get_vector_from_matrix_4D(rotation[1], pvuw)
  let result = rotate_4D_vectors(a, b, angle)
  let a_ = result[0];
  let b_ = result[1];
  let pvuw2 = replace_vector_in_matrix_4D(rotation[0], a_, pvuw)
  pvuw2 = replace_vector_in_matrix_4D(rotation[1], b_, pvuw2)
  return pvuw2;
}

export {rotate_4D_vectors, replace_vector_in_matrix_4D, get_vector_from_matrix_4D, calculate_new_pvuw};
```

This seemed to correct it.

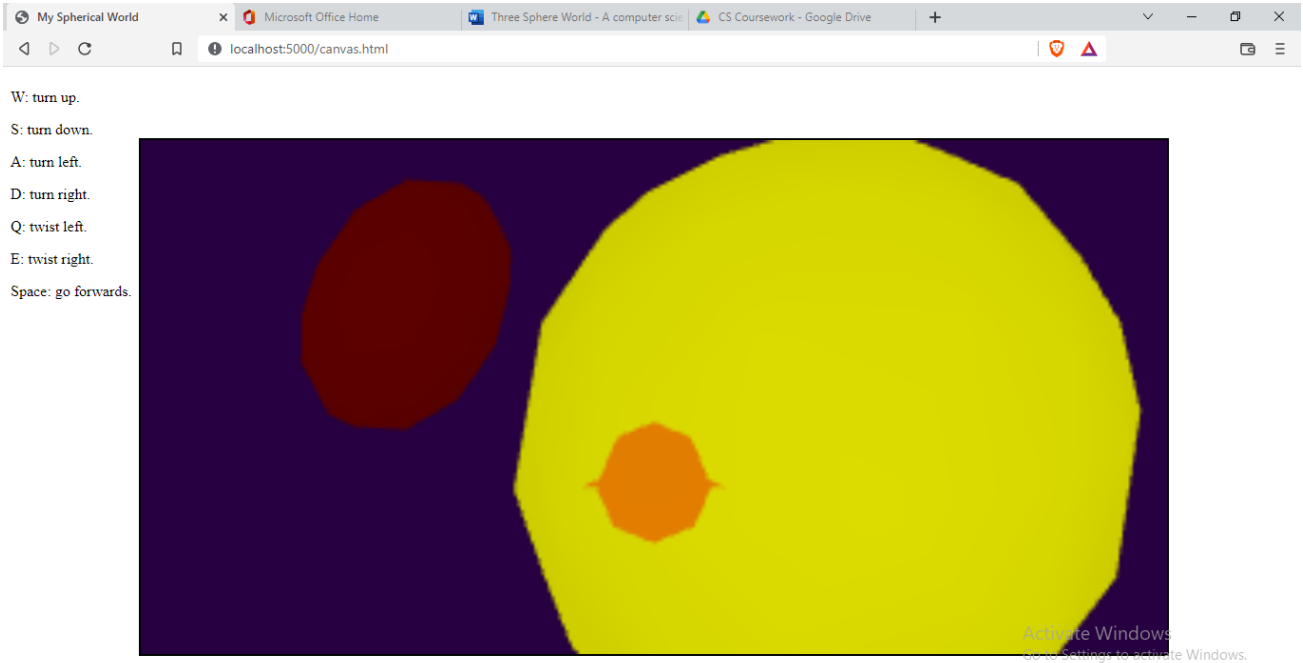


The rocket also turned, as expected.


My Spherical World
localhost:5000/canvas.html
Type here to search
Rain...
ENG
00:08
31/03/2022

P5.3 Iteration Evaluation

I have decided to include a tutorial, to help users get to know how to move.



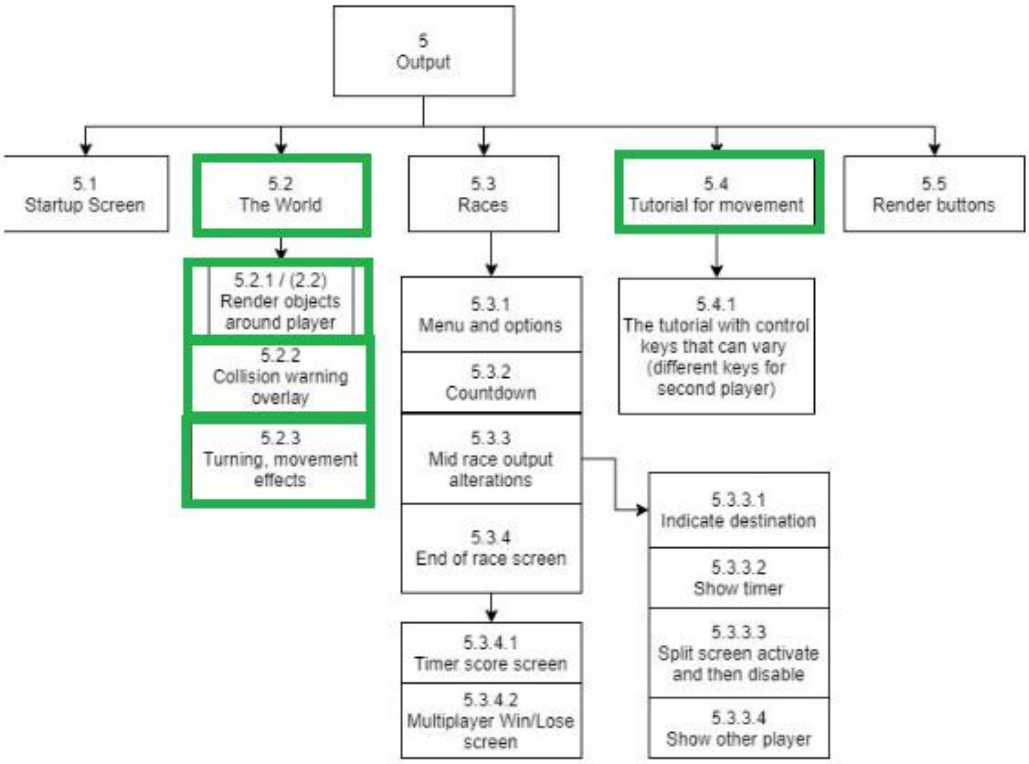

```
    <div id='collision_background'>
      <div id='WebGL_Fallback' hidden='true' class='info_yellow'>
        Web GL is not supported by you or your browser.
        Please give it some support. A kind word or two will make him feel happier.
      </div>
      <canvas id='canvas'></canvas>
    </div>
    <div class='tutorial'>
      <p>W: turn up.</p>
      <p>S: turn down.</p>
      <p>A: turn left.</p>
      <p>D: turn right.</p>
      <p>Q: twist left.</p>
      <p>E: twist right.</p>
      <p>Space: go forwards.</p>
    </div>
  </body>
```

```
22  body{
23    position: relative;
24    height: 100vh;
25    overflow: hidden;
26  }
27
28  .tutorial{
29    top: 0;
30    position: absolute;
31    background: #fff;
32    display: block;
33  }
34
```

I have added the tutorial css class. I have also added an overflow:hidden property to body, meaning the user will not be able to scroll down, which hinders the usability of the game, since Space is both used for scrolling and playing.

P5.4 Iteration Evaluation

The iteration went successfully. The rocket added usability and understanding to the simulation.



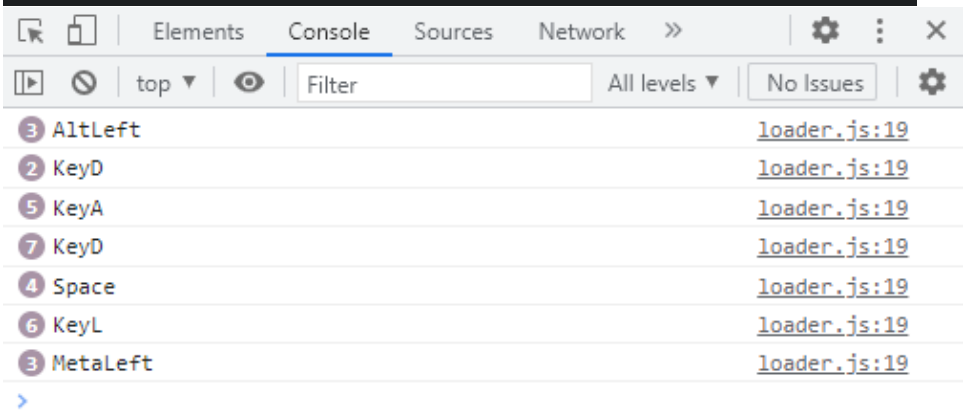
T1 Functionality Testing

T1.1 Debug Mode

In order to show what keys I am pressing while testing, I have decided to log the key presses, by using an event listener.

Keys are only logged if the debug mode flag is turned on.

```
11  const use_rocket = true;
12  const use_collision_overlay = true;
13  const debug_mode_on = true;
14
15  function loader(Canvas){
16
17    if(debug_mode_on === true){
18      document.addEventListener('keydown', (e)=>{
19        console.log(e.code);
20      })
21    }
```



T1.2 Blackbox Testing

Test ID	Input or Question and Expected Result	Actual Result	Pass **or** **Fail**
TB1	If you start the program, there is a starting screen. You can press a button to start the simulation.	No start screen.	**Fail**
TB2	Once in the simulation, press ESC. A) The pause screen should appear B) Once you re-enter the simulation, the rocket is no longer moving.	No pause screen.	**Fail**
TB3	In two player mode, if ESC is pressed, both players' games are paused and the start screen appears.	No two player mode.	**Fail**
TB4	Pressing (x) moves player (y) for: (x, y): - F, forwards - W, turn up - S, turn down - A, turn left - D, turn right - Q, spin ACW - E, spin CW This can be seen by closer planets moving the other way.	F replaced for Space. All other keys work.	Pass
TB5	Pressing, then releasing x for all the above stops the motion.	Passed.	Pass
TB6	Press two buttons, one after the other and hold both. The movement should change from one type to the other. - A then W - Q then F - F then Q - S then A then F - A then D	As expected. F is now space.	Pass
TB7	Press two buttons, one after the other (from above), and release the first after pressing the second. The second movement should continue.	Stops the second movement	**Fail**
TB8	Press two buttons, one after the other (from above), and release the first after pressing the second. Then release the second. The movement should stop after this.	Movement stops before this point.	**Fail**

TB9	Press a key that is not in (W, A, S, D, Q, E, Space). Nothing should happen.	Passed.	Pass
TB10	Press a key that is not in (W, A, S, D, Q, E, Space), after having pressed one of these valid keys. The movement should not stop.	Passed.	Pass
TB11	Press A, release A. Press F, release F. This should turn the rocket left, then move it forward.	Undecided. The effect seemed to show this.	Pass
TB12	Point camera at a planet far away and press F to move to it. This should either shrink, then enlarge the planet, or just enlarge the planet.	Passed.	Pass.
TB13	Point camera at an empty piece of space. Remember the starting image. Press and hold F to move forwards. You should reach your starting image after some time.	The image is the same	Pass
TB14	Planets should appear similar to their 3D models. This is a visual test.	Almost spherical. Like models.	Pass
TB15	There should be a rocket bottom center of screen to represent player.	There is.	Pass
TB16	Approach planet head on. You should stop before the planet and not see inside it. You should move back, then come to a halt.	Works for yellow big planet. Works for smaller red planet. But I can see inside red planet.	Pass
TB17	After collision, quickly release key and turn left till you face away from the planet. Does the user seem to be seeing the inside of the planet?	The user can see inside for small planets.	**Fail**
TB18	Place a planet close to the start position of the player. Is the player placed inside the planet? Can they move out of it? What happens?	Player stuck inside the planet. Collision overlay flashing.	**Fail**
TB19	The same planet should appear at different shades of darkness based on the distance to it.	The closer, the lighter.	Pass
TB20	When moving towards a planet (pressing Space), it should become brighter.	Yes.	Pass
TB21	Starting on the left of a close planet, it is possible for the player to continuously press forwards (SPACE), then press right (D) over and over again, and end up in the same place they started (circumnavigate the planet).	Didn't end up in the same position, but resulted in a similar image.	Pass

TB22	The player can turn upwards indefinitely, for at least 3 turns (repetitions of image). There should be no change in their motion.	No change to speed or image.	Pass
TB23	Start by facing a planet. Turn upwards until you see the same planet. It should now be a different shade. Turn the same way again. You should end up at the start image.	As expected.	Pass
TB24	There is a tutorial that shows the player how to move around the world.	Written tutorial.	Pass

Result: 17/24 tests passed.

T1.3 Whitebox Testing

Test ID	Input or Question and Expected Result	Actual Result	Result
TW1	Remove all planets from the map object. The screen should be a continuous colour for the player, with no objects. The rocket should still seem to turn when being moved.	Continuous colour.	
...	Add two planets close to each other, different in colour. Would the two planets overlap?	They overlap.	
	Add lots of small planets (20 of them) to the world. Does the game seem to slow down?		
	Add a planet with double values in one of the (v,u,w) axes. Does the planet appear squashed?	No. Same as before. Doing a tenth of a value makes its edges jagged.	
	Add a planet with double values in the p position vector. What happens?		
	Add a planet with some pvuw axes not perpendicular to each other.		
	Start the player with an invalid pvuw position, orientation: where vector lengths are not 1.0 or where vectors are not perpendicular to each other. What do you see? What happens as you move around?		

Lily, Marco

Mum's feedback

- Liked the game.

- Took 10s to find tutorial.

- Took 20s to learn how to move.

- Liked twists.

- Started experimenting with turning left and up at the same time.

- Said she didn't mind it rotating separately left then up.

- But said it would be a good improvement to make to combine movements into diagonal motion.

- Collisions were something she expected.

- Wanted a diagram to help with understanding position.

Dad's feedback:

- Tried clicking at first
- 10s found tutorial
- 5s learnt motions
- Did not realise that holding one key would have an effect on a different motion.
- Liked the game.
- Disorientated, but could move to red planet once he saw it.

Evaluation

End of next week

08.04