Handover and Competition Report.

Intended for Jonathan Weightman.

Prepared by

Group PA2207:

- *Djed Curtis (19744647)*
- *Hasan Karabork (19679091)*
- *Jarrod Baker (20487821)*
- *Toufic Tannous (18528419)*

Executive Summary

The aim purpose for the hand over report is identify changes to the design and have test results so that any issues within the system is resolved with a fix in the development. In the report will be each individual task on what they did in the project and how they implemented the models in the game. The goal is detecting any bugs or errors and how to improve future development in the Project. The team will also be talking about the function, screen design adaption, Bug fixing and future work improvement that can be taken in place to improve the overall system.

In the report the team intention is to control and handle any issues that arises and come with an easy fix to get the system working. We plan to focus on the test results with a clear breakdown on how we can improve the interaction for Clara as well as the functionality. It will help to control any system crash that can occur when the system is online and running.

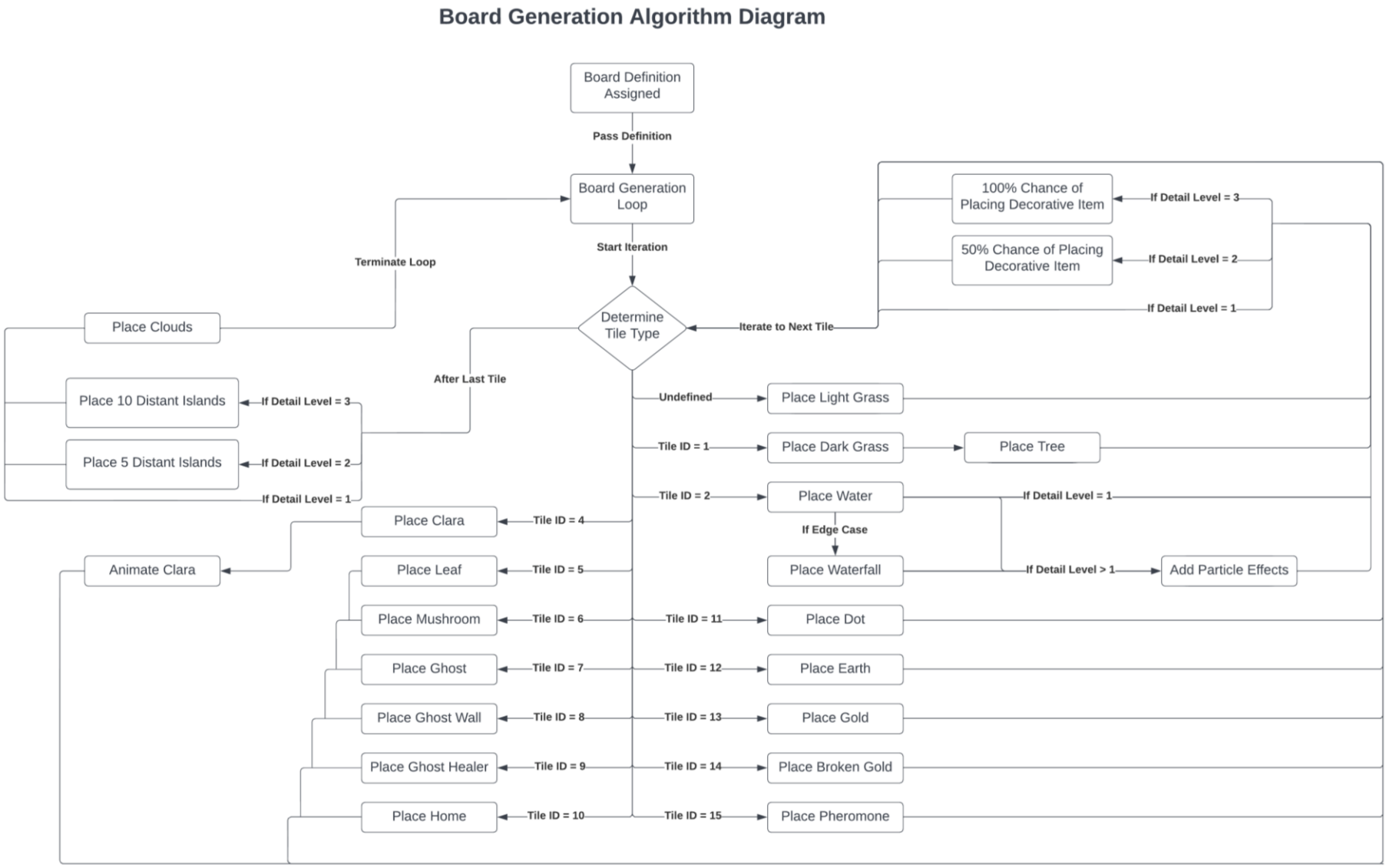
The factors that are mainly considered in the report is get the project test completed with no errors and have essential fixes that can occur within the system. The team task is to show what they have done and how they completed their task in the development of Clara 3D.

Table of Contents

Introduction

This report contains information regarding the Clara’s world 3D project with respect to any variations made to original designs. Within there will be relevant sections in regard to our implemented algorithms, test results, Variations throughout development along with any known bugs and plans for further development.

Implemented Algorithm



Test Results

Use Case / Feature	***Process User Code***
Test Purpose:	
Expected Results	
Success/Failure	
Test Data	
Movement Function	Outcome
Move	Clara moves forward one ‘tile’
Turn Right	Clara turns 90 degrees to the right
Invalid Move	Game stops an error is shown to user
Place Leaf	A leaf is placed underneath Clara
Grab Leaf	The leaf underneath Clara is removed

Use Case / Feature	***View Leader Board***
Test Purpose:	Confirms that the player can view the scoreboard for Clara’s World
Expected Results:	User is able to view their place on the leader board as well as their peers
Success/Failure:	N/A

Use Case/Feature	***Graphical Detail Section***
Test Purpose:	
Expected Results:	
Success/Failure:	
Test Data	
Options	Expected Outcome
High Level Graphics (3)	All models will be used in the board generation, on limit on file sizes
Medium Level Graphics (2)	A limited number of models will be used during board generation, some limit in file sizes used
Low Level Graphics (1)	Very limited selection of models used, and harsh limits placed on file sizes

Use Case/Feature	***Camera Movement***	
Test Purpose:	Ensure that cameras in the three-dimensional environment have limitations as to restrict the user from breaking their view angle	
Expected Results:	Scene camera should behave correctly and no go into out of bounds areas	
Success/Failure	Success	
Test Data		
Camera Function	Description	Result
Angle #1	Top-Down angle	Success
Angle #2	Side view angle	Success
Angle #3	Restricted free cam	Success

Variations Report

Functional / Use Case Design Variations

1. Process user code: Partially completed

   Movement is implemented and reliable, however processing user code from a file is yet to be fully implemented. MoveForward, turnRight, interact with leaf+mushroom and player death have all been implemented fully.
2. View User LeaderBoard: Moved Out of Scope

   This non-essential use case was moved out of scope for this project.
3. Level Selection: Partially completed

   Menu implemented allowing for level selection. Currently not restricted to levels that a user has unlocked.
4. Graphics detail selection: Implemented

   Users have options to choose from low, medium and high graphic settings, this limits the visual output of the user's computer, allowing for higher performance.
5. Move Camera: Implemented Differently

   This use case was changed from the original intention of having a free-roaming camera (that can go wherever the user wants), to having different camera angles that look at clara from different angles. There is a free-roaming camera setting, but has its movement restricted to the top of the board.

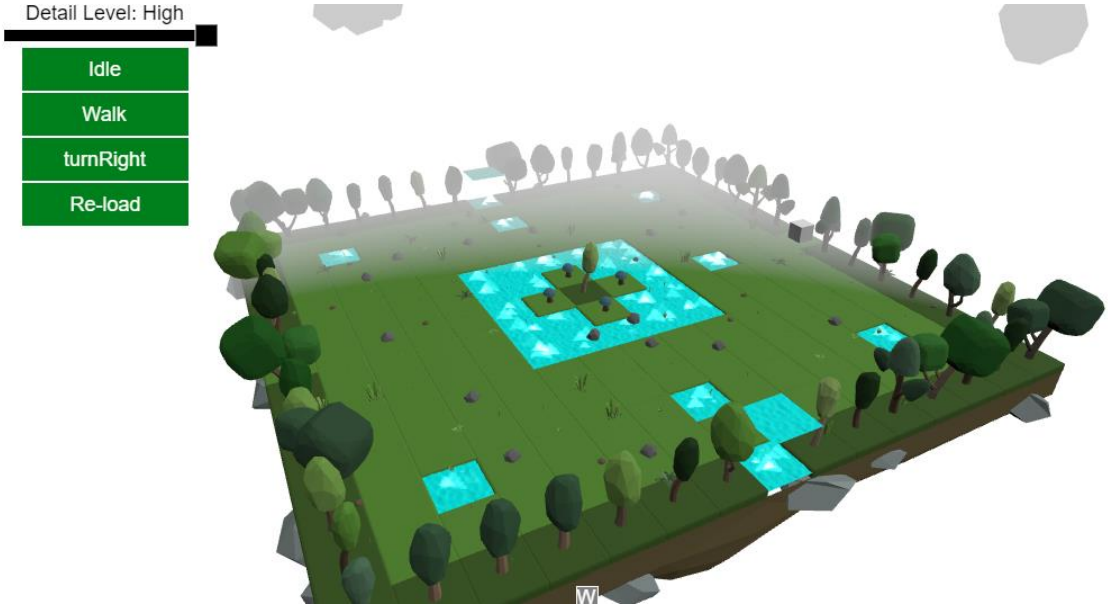
Screen Design Variations

Over the course of this project, we have changed the screen design quite a bit. Most notably the Graphical User Interface, or GUI. This section gives a quick recap of the variations of our GUI, and the final product as of this report's completion.

Seen below is our original screen design, no GUI was present in this iteration and many of the actual game elements were missing.

In this screenshot, a simple GUI overlay has been added. Including three camera option buttons on the right with placeholder icons, a detail slider on the left (which had many issues) and two buttons to control the Clara model's animations.

This iteration includes the same as the prior, however while movement was being implemented, buttons to control said movement were added to the GUI. As well as many updates to the game in regard to models, elements, and animations.



Here we have added new smoother looking icons for the camera angle buttons, replacing the placeholder icons seen prior.

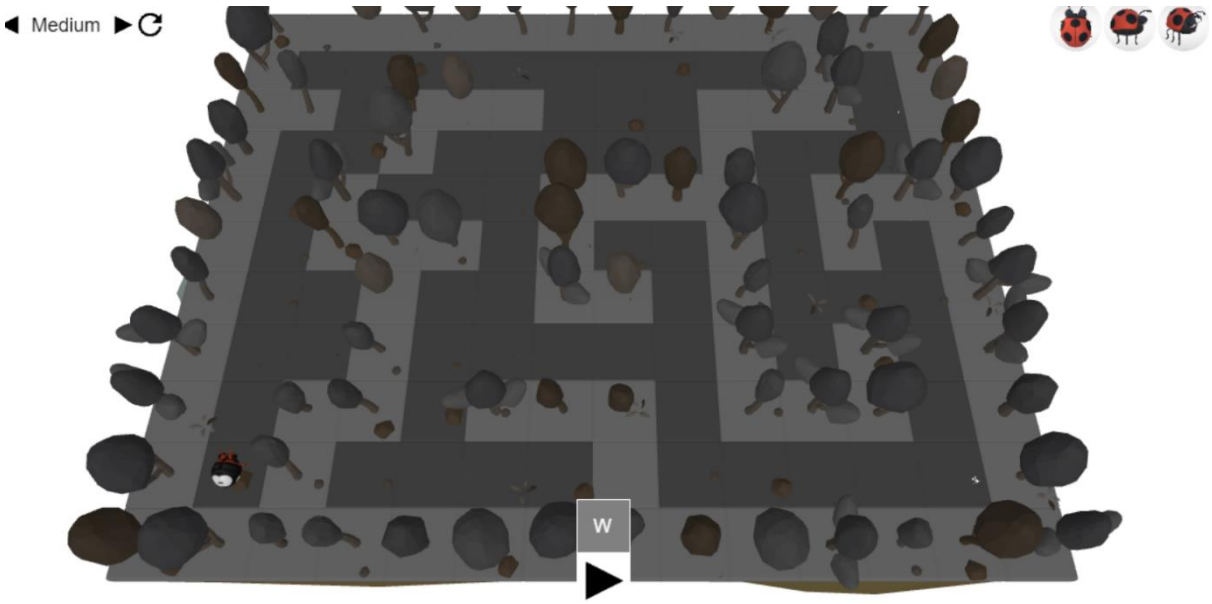
In the final iteration, seen on the next page, the GUI has had an overhaul, replacing some sections entirely, and adding speed options in the bottom.


Reload Button
Detail
High
Detail Options/Picker
- High
- Medium
- Low
Camera Options
Camera Options
- Angle 1 (Left) = Top Down View
- Angle 2 (Middle) = Side View
- Angle 3 (Right) = Restricted Free View
x0.5
x1
x2
x5
Speed Options
- x0.5
- x1
- x2
- x3

Known Bugs

During our development process we encountered many different bugs, majority of which we were able to fix and patch, however a few minor / rare bugs remain. One bug is in relation to the UI that we have developed, there have been some instances that when the page has been reloaded the UI flips upside down and will sometimes fix itself or will only be fixed upon reloading the page once more. The frequency of this bug is very low as it only happens very rarely (once in 70 reloads).

Another bug that occurs extremely rarely, causes all models and elements in the game to have their colours inverted. For reference see the image below. This bug has only been experienced when hosting the server locally, and when deployed, this bug has not been encountered yet.



We also experienced a duplication bug with meshes when the game is reloaded, however like the last bug, this has only been observed when hosting locally and on the deployed game it has still not been seen.

Outstanding Development Work / Further Improvements

For this particular project, there may be a few features that are left to be desired, one of which is implementing the ability to have an administrative section where different levels can be edited or new levels can be added, along with having other administrative related features (adding/removing users etc). Another feature that will need to be developed will be allowing the backend of the client's current system to interact with out project, that being where users will be able to enter whatever pseudo code is being used and having that translated to moving Clara within the game or performing different tasks. This backend implementation will be made easier through the code explanation section of this report.

Code Breakdown and Descriptions

Model Exporting & Converting

1. Write click the item you dragged and click exported to FBX, Click Binary for Export Format on Option.
2. Open CMD
3. Write command cd C:\Users\PC\Desktop\Clara\unity\project\Assets
   Press Enter
4. Write Command
   FBX2glTF-windows-x64.exe -i “name”.fbx -o "NewName.glb"
   Press Enter
5. Open assets in Clara folder and open it in paint

Board Generation

Relevant Variables and Functions.

Variable / Function	Description
SceneDefinition	Public object that contains the details of the given board (rows, columns, and an object called tiles)
treeArray[]	Two-dimensional array of the board storing tree locations
houseArray[]	Two-dimensional array of the board storing possible locations for larger models (houses, windmills)
existingTile	The currently selected tile from the SceneDefinition variable
createDistantIsland()	Function that creates and places distant islands
placeClouds()	Function that places 4 clouds above the map on each polar direction
checkNFOAvailability()	Function that runs through the movementGrid[][] array to check for valid placements for large objects
checkSurroundingTiles()	Function that checks the surrounding positions of a given position to determine whether or not a large object can be placed there
placeTrees()	Function that goes through the treeArray[] and places trees in given locations
getTree()	Function that loads an instance of a tree model from MeshLoader.ts
chooseItem()	Function that places a miscellaneous item on a tile

Note: The checkNFOAvailability(), getTree(), createDistantIsland(), chooseItem() and placeClouds() functions all rely on the detail level variable to determine how many items to place, as well as what items to place based on their file size.

Program Flow

Once the createScene() function is complete, the loadScene() function is called. This loadScene() function is the parent of the Board Generation Algorithm. The Board Generation Algorithm contains a nested for loop that runs based on the defined SceneDefinitions rows and columns. The loop iterates through all the tiles in the rows by columns grid, and checks if the current tile exists in the board definition, if it does, it will place an item based on it's tile ID, otherwise, it will place a blank grass tile.

Functions of Interest – Descriptions

createDistantIslands() – Runs at the end of the Board Generation Algorithm.

1. Checks the current detail level selected
    a. If the detail level is high, it will randomly generate 10 distant islands
    b. If the detail level is medium, it will randomly generate 5 distant islands
    c. If the detail level is low, no distant islands will be generated
2. Islands are created the same way the board is generated, however, the rows and columns are defined randomly at the time of the function

placeClouds() – Runs at the end of the Board Generation Algorithm.

1. See the 'Clouds' section

checkNFOAvailability() – Runs directly after the Board Generation Algorithm

1. Runs a nested for loop based on the rows and columns of the board, starting from an index of 1 on each and a maximum value of rows and columns minus 2. This is in order to prevent NFO placement on edge cases.
2. Each iteration checks the selected tile and it's surrounding tiles using the checkSurroundingTiles() function
3. If the 4 tiles checked are valid, it will store the index of that spot in the houseArray[] and update the treeArray[] tiles to 0 to prevent trees from being place inside the NFO

checkSurroundingTiles() – Child function of the checkNFOAvailability() function

1. Returns either true or false based on if the 8 surrounding tiles of a given tile are all valid locations (meaning they're all equal to 1, being trees)

placeTrees() – Runs after the checkNFOAvailability()

1. Runs a nested for loop based on the rows and columns of the board
2. Each iteration determines if the selected tile's ID is 1, and if so, calls the getTree() function

getTree() – Runs during the placeTrees() function loop

1. Received the coordinates of the desired location for a tree
2. Determines the selected detail level, only placing lower poly trees if the lowest has been set
3. Instances a tree model and places it at the coordinates passed to it

chooseItem() – Runs at the end of each iteration in the Board Generation Algorithm

1. Checks the selected detail level
    a. If high, 100% of placing an item
    b. If medium, 50% chance of placing an item
    c. If low, no item will be placed
2. Determines if the current tile is a water or grass tile
    a. If water tile, selects water item
    b. If grass tile, selects generic item
3. Randomly selects an item and places it at the given coordinates

Detail Level Selector

Relevant Variables and Functions

Variable / Function	Description
detailLevel	Global variable, either 1-3, defining the detail level of the game as either low, medium, or high respectively
passedDL	Local detail level variable, used when reloading the scene to apply detail changes
claraGUI	Graphical User Interface variable, see the 'Graphical User Interface' section
optionsPanel	GUI panel / section to group the game options
detailHeading	GUI textblock element for the heading of the detail options
detail	GUI element that changes based on the detail level
lowerDetailBtn	GUI button to select a lower detail level
higherDetailBtn	GUI button to select a higher detail level
reloadBtn	GUI button used to reload the game

Program Flow

Once the SceneLoader class is run, the initCanvas() function runs first, in which the detailLevel variable is set to High (3),

On the initial loading of the game, the detailLevel global is set to High (3) by default, this is done in the initCanvas() function. Following this, the createScene() function is called, and is passed the global detailLevel variable as passedDL, which is then assigned to the global. On the first load of the game, this is redundant, but is necessary in order to change the detail level of the game once the scene has been created.

Within the createScene() function, the claraGUI is created and the optionsPanel is assigned to it, followed by the detailHeading which is placed inside the optionsPanel. The detail GUI element is then created, with the respective selected detailLevel (low, medium, or high). The lowerDetailBtn and higherDetailBtn are then created and placed on either side of the detail element in the GUI. Each of these buttons has a handler that will either decrease or increase the global detailLevel variable respectively. Once the reloadBtn is hit by a user, the current scene is disposed of, and a new one is created, with the detailLevel of the old scene that was selected being passed through to createScene() so that the new scene will be created with the new detailLevel value.

Functions of Interest – Descriptions

initCanvas() – See ‘initCanvas() function’ under ‘SceneLoader Class’

createScene() – See ‘createScene() function’ under ‘SceneLoader Class’

Graphical User Interface

Relevant Variables and Functions

Variable / Function	Description
claraGUI	A Babylon AdvancedDynamicTexture variable – Fullscreen GUI overlay
cameraPanel	GUI panel / section to group camera buttons
optionsPanel	GUI panel / section to group the game options
speedPanel	GUI panel / section to group speed settings
detailHeading	GUI textblock element for the heading of the detail options
cameraHeading	GUI textblock element for the heading of the camera options
cam1	GUI button for the top-down camera angle
cam2	GUI button for the side-view camera angle
cam3	GUI button for the restricted free cam camera angle
detail	GUI element that changes based on the detail level
lowerDetailBtn	GUI button to select a lower detail level
higherDetailBtn	GUI button to select a higher detail level
reloadBtn	GUI button used to reload the game
halfSpeedBtn	GUI button to change game speed to x0.5
defaultSpeedBtn	GUI button to change game speed to x1
doubleSpeedBtn	GUI button to change game speed to x2
fastSpeedBtn	GUI button to change game speed to x5

Program Flow

The GUI is created in the createScene() function, and is created in the following order:

1. claraGUI is created and defined as a full screen overlay
2. The panels of the GUI are defined, cameraPanel first, optionsPanel second, and speedPanel lastly. Then they're added to the GUI variable (claraGUI)
3. Headings for GUI sections are then created (detailHeading and cameraHeading) and then placed in their respective panels
4. Then buttons are created, their handlers are setup and then they're added to their respective panels
    a. lowerDetailBtn, higherDetailBtn, and reloadBtn = optionsPanel
    b. cam1, cam2, and cam3 = cameraPanel
    c. halfSpeedBtn, defaultSpeedBtn, doubleSpeedBtn, and fastSpeedBtn = speedPanel

Functions of Interest – Descriptions

- createScene() – See 'createScene() function' under 'SceneLoader Class'
- loadScene() – See 'loadScene() function' under 'Sceneloader Class'

SceneLoader Class

Relevant Variables and Functions

Variable / Function	Description
initCanvas()	Initializes the HTML canvas element, allowing it to be used to create the scene
createScene()	Function that creates and sets up the required variables for the scene to be loaded
loadScene()	Function that actually creates all models, objects, and elements and lays out the game
scene	The scene variable is stored both as a local and a global, and contains all the meshes, models and so on that make up the game
light	The light variable is stored as both a local and a global, and is the point of light within the scene that can be moved or altered
engine	The engine variable is a reference to the actual BABYLON.JS engine that is created when the initCanvas() function runs
camera	The camera variable is stored both locally and globally and is a ArcRotateCamera type
assetsManager	The assestsManager variable contains all meshes and models loaded into the scene

Program Flow

Once the SceneLoader class is loaded, the initCanvas() function runs first, which is passed the HTMLCanvasElement as ‘canvas’. This canvas variable is then used in the creation of the engine which will be used to create the scene. After the initCanvas() function is finished, the screen window is resized, and a scene is created by calling the createScene() function. Once the createScene() function is complete, the loadScene() function runs, loading all the meshes into the scene that was created.

Functions of Interest – Descriptions

createScene() – Used to create a scene, contains all the required functions to create an empty scene with GUI elements

loadScene() – Loads all models and elements into the scene that’s created by createScene()

MeshLoader Class

Relevant Variables and Functions

Variable / Function	Description
Atlas	An object containing maps of many types of meshes to be loaded into a scene
Tiles{[][]}	An object containing objects that contain pairs of strings, being the index, and the name of the models
preloadMeshes()	Function that preloads all meshes into the scene so that they can be instanced to enhance perfomance

Program Flow

The MeshLoader class is first imported into the SceneLoader class and is then created within the canvas initialisation. Once called in initCanvas() in SceneLoader, the preloadMeshes function runs. In which all models stored in the Tiles object are returned back to the SceneLoader class, making them all accessible through the ‘Atlas’ call.

Functions of Interest – Descriptions

preloadMeshes() – Loads all meshes in the Tiles object such that they can be used

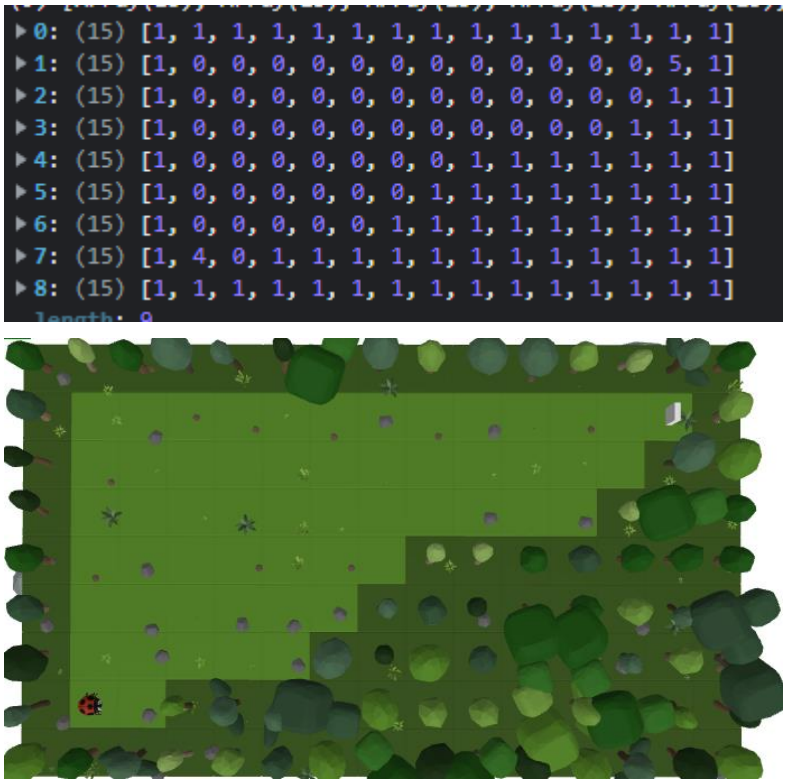
1. The total amount of objects is found and stored in totalToLoad
2. A nested for loop adds each individual mesh’s root mesh into the assetsManager
3. The preloadMeshes() function then returns the assetsManager if the above is successful

Island Bottom Placement & Scaling

In order to successfully have the bottom island section of each level placed and scaled dynamically, a few variables needed to be considered. The first step would be finding the centre of each level which is done through the following function (Rows * 2.1 - 2.1) / 2 & (columns *2.1 – 2.1)/2. Once the centre of the level is found then, the island model is then scaled according to the map by the following line of code “root.scaling = new BABYLON.Vector3(rows/4.5, 1.8, columns/4.5)”. It is important to scale by the number of rows and columns to ensure that the model fits flush underneath the level, the reason for why each value of row and column is divided by 4.5 is because the model itself is 4.5X4.5 tiles in size. A static value of 1.8 was used to scale the island in the y axis.

Two-Dimensional Movement Grid / Tile Map

The tile map is used in a lot of different functions across the project, it is a 2D array which forms a map of the level consisting of the tile ID’s which is visualised as follows:



The 1’s indicate where the dark tiles are placed, and the 0’s indicate where Clara can actually walk and move around. The creation of the tile map is as follows:

First a global array is created to ensure that the required functions can have access to the array. The array is then initialised to be filled with 0’s, the reason for this is because the map definitions don’t specify the areas on which Clara can move on, the logic behind this is that the undefined areas are the light tiled block which Clara moves on. After the array has been initialised, the other tile IDs are placed into the array through the board generation loop.

Reload

The reload function is quite simple and utilises the createScene and loadScene functions. The reload function itself is linked to a button that the user can press, once the button is pressed the scene is disposed using BabylonJS inbuilt function “dispose();”, a new scene is created using the createScene() function which requires a global definition of the scene to be passed into it, finally the loadScene() function is called which also has the global scene definition passed into it.

Cloud Placement

The clouds placed into Clara’s world are quite simple, the centre of board is first found, once found a cloud is loaded to the north, south, east, and west of the board.

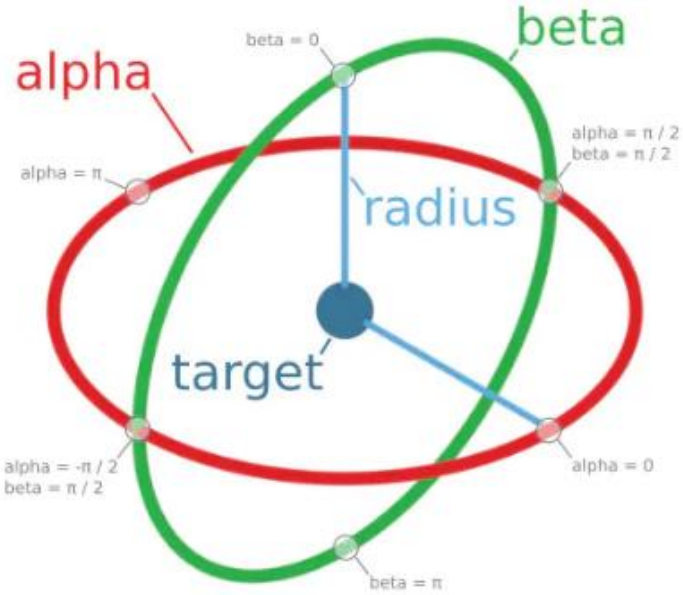
```
// function to place 4 static clouds around the island on all angles
 placeClouds(rows: number, cols: number) {
   var rCentre = (rows * 2.1 - 2.1) / 2;
   var cCentre = (cols * 2.1 - 2.1) / 2;
   let x,
     y = -20,
     z;
   for (let i = 0; i < 4; i++) {
     switch (i) {
       // bottom cloud
       case 0: {
         x = (rCentre + rows) * 2.1 + 10;
         z = cCentre;
         break;
       }
       // top cloud
       case 1: {
         x = (rCentre - rows * 2.1) * 2.1;
         z = cCentre;
         break;
       }
       // right cloud
       case 2: {
         x = rCentre;
         z = (cCentre + cols) * 2.1;
         break;
       }
       // left cloud
       case 3: {
         x = rCentre;
         z = (cCentre - cols * 2.1) * 2.1 - 10;
         break;
       }
       default: {
         break;
       }
     }
     this.importMesh("Cloud", x, y, z, Math.random() * 180, true, 7, 7, 7);
   }
 }
```

Water & Waterfall Placement

Water is placed during the board generation loop when the tile ID is equal to 2, the water cell is positioned by multiplying the row and column index by 2.1, its y value is static with value 0.9 and scaled by 1.2 in the x and z axis. When each water cell is placed, a particle emitter is also placed at the same location to make the water look like it's moving. For edge cases of each board (where row value is 0 or number of rows -1, column value = 0 or number of columns -1) a waterfall is placed. Waterfall placement first checks for a water edge case, when found, a water cell is rotated and placed in a manner to ensure that it fits flush with the sides of each board, five more water cells are then placed below it to make the waterfall appearance. During the placement of each water cell for this waterfall, particle emitters are also placed, these emitters display more particles per frame to make the water look more aggressive.

Restricted Free-Camera

This feature involved changing a few variables with the camera. When the camera is initially placed, the user can click on the third camera icon which enables the functionality of the restricted free camera. When pressed, the lower and upper beta values are changed to 0 and PI/2, these beta values determine how high and low the camera can go. Then the maximum and minimum radius values are changed to 50 and 30, these values determine how much the user can zoom in and zoom out. Below is a diagram to help visualise these explanations.



Movement

Relevant Variables and Functions.

Variable	Description
Clara	The Clara mesh
playerAnimator	this allows for the animations attached to the Clara model to be accessed and used
directionFacing	Used to move through movementGrid[]
movementGrid[]	2d array of the board (the logical representation of the board)
startPosition	Where Clara is loaded in (also needed for when the page reloads)
currentPosition	This value represents where the mesh currently is (returning a vector3 of x,y,z values)
claraXCoord	Where Clara is on the movementGrid[]
claraYCoord	Where Clara is on the movement grid[]
leavesCollected	Count of how many leaves Clara has collected in this level so far (resets when the page is loaded)
justMoved	Used to ensure user input cannot be spammed
alive	Used to check if Clara has died (hence stop taking user input)
deathspeed	Timer used to control the death animation speed (in milliseconds). Without this, the death animation gets skipped.
currentTurnSpeed	Timer used to control the turning speed. Without this, user turn input can be spammed, turning very fast.
currentFrameSpeed	How many frames being used while moving an object (affects how fast the objects move).
currentMoveWaitSpeed	Timer used to control moving forward. Without this timer, a user can spam movement input, which will put Clara in the wrong position on the board. On the logical board she will be in the correct position, but in the game, she will be on the incorrect tile (or in between tiles).
Leafs[][]	A 2d grid containing every leaf location
leafAnimatorArray[][]	A 2d grid containing every leaf location but pointing to the animations contained within the leaves. When leaves is updated, the relevant leaf animation is called as well.

Mushrooms[][]	A 2d grid containing every mushroom location
currentLeafAnimationSpeed	Timer used to control the delay before deleting a leaf (if a wait is not implemented, the leaf will be deleted immediately, looking like the animation never played.

Notes: the arrays of 'speeds' are to do with how fast the game runs. There are currently 4 speeds in the game, half, normal, double and 8x speed (in that order within each array).

Functions of interest

claraMoveForwardFunction();

checkMove();

checkMushroomMove();

claraTurnRightFunction();

newLeafAnimation();

newMushroomAnimation();

delay();

keyboard event listeners: this code is inside the "intiCanvas" function. To search for this code, ctrl+f: events (there are very few mentions of events in the code currently).

playClaraDeathAnimation();

playClaraIdleAnimation();

playClaraWalkAnimation();

playClaraTurnRightAnimation();

playClaraTurnLeftAnimation();

DEPRECATED FUNCTIONS:

castRayMushroom();

castRayLeaf();

animateRandomObject(mushroom);

animateLeaf(leaf);

Note: the above deprecated functions have been left in for anyone else who works on the project and would like to implement ray-casting.

Note: the ‘turn right/turn left’ animations in the above functions are not implemented in the game. The turn left, only turns 30(ish)degrees, hence cannot be run just before actually turning the model (would need to turn 90 degrees for the turn to look seamless). Turn right is the same problem (note that turn right begins facing 30 degrees to the left (because it is run after the turn left function, Clara is already facing that direction when the animation starts, this isn’t a problem, just something to be aware of).

Program flow

Current implementation waits for a key to be pressed (“w” to move forward, “d” to turn right). When w is pressed, move function is called. For a move to occur, the following conditions must be met:

- **Clara is alive** (Boolean variable, when Clara dies, all movement is locked)
- **Clara has not just moved** (Once Clara has moved, the Boolean “justMoved” is set to true, it is set back to false after a set delay, “currentMoveWaitSpeed”, and Clara can move again. This is necessary to prevent Clara from moving out of sync with the tiles on the board.)
- **The forward move is valid** (check that there is not a tree/water tile ahead or that a mushroom can be moved to the tile 2 spots ahead of Clara etc.)

Inside the “checkmove” function, if the move is valid, Clara is moved forward, and the “movement grid” with Clara’s location is updated to reflect her current position. If a mushroom or leaf was interacted with, the relevant 2d array is updated to reflect the current state of the board (leaves, mushrooms)

Functions of interest – Descriptions:

ClaraMoveForwardFunction() – movement begins with this function.

1. First, check if the player has just moved and if they are alive.

   Just Moved Variable justification

   The just moved variable is required to control user input. The logical implementation of the board (movementGrid[]) and the locations of objects in the game are not explicitly linked.

   When Clara moves forward, she moves forward 2.1 (units) in game, and the movement grid is updated to show her new position. Both of these are updated individually (rather than Clara's position being generated by reading the movementGrid[]).

   When she moves, her current x,y,z position (henceforth referred to as her Vector3 (a Babylon.JS term)) is the beginning point of movement, and she moves 2.1 units forward from there. If she is already moving between points 0 and 2.1 (1.4 for example), she will move to position 3.5 (1.4 + 2.1) instead of moving to position 4.2 (2.1 + 2.1) as intended.

   To control this, the justMoved variable acts as a timer that waits for the movement to finish, then allows movement to occur again. This prevents Clara from being in the wrong position on the board.

   Once a player dies, they must not be able to continue moving Clara. A death occurs when a player either walks into water, a tree, or a ghost. Alive is switched to false, which prevents further user movement.

2. Check move: this function checks if Clara is able to move forward or not. She cannot move forward if a tree/water/ghost is directly in front of her (Clara will die in this case), and also checks if a mushroom can be moved (in this case, the tile in front of the mushroom must be available to move to (e.g., Can't be a tree or water).)

2.1 If there is a mushroom in front of Clara (on the movement grid), first check if the mushroom can be moved to the next position (e.g., the tile in front of the mushroom is just grass, not a tree or water). If it is a valid move, then call the function newMushroomAnimation().

2.2 The newMushroomAnimation() selects the mushroom mesh at Clara current x/y coordinates (inside the movement grid). That mushroom is animated forward 2.1

units (in the same direction as Clara), and the 2d array of mushrooms is updated to reflect the new state of the board.

2.3 For a leaf interaction, the newLeafAnimation() is called. The relevant leaf is selected using the current Clara x/y coordinates from inside the 2d array of leaves. The leaf is animated for a set time, then disposed of (note: if a delay is not put between the animation and disposal, the animation will not be given time to display, and will immediately be disposed. This is why this function is asynchronous, to allow the delay function to await completion (delay is defined at the top of the SceneLoader).

3. Once the above code has completed, user input is still blocked (or else the user will be able to move Clara between tiles, which is unwanted). Hence the "justMoved" variable is switched back to false after a delay timer (defined by currentMoveWaitSpeed).

Unreliable methods of movement:

1. **Ray casting:**

   The ray is cast from Clara for 1 frame and returns the first object found. This is a problem when miscellaneous items generate in front of a mushroom or leaf, as they will be moved in place of the mushroom/leaf. Another problem was that while the leaf is bobbing, it is possible for the ray to miss the leaf (it would be possible to keep casting the ray until the leaf is hit, but this method is far from ideal/reliable).

2. **Working with x/y coordinates in Babylon js:**

   This method is also unreliable (using this.clara.position == this.mushroom.position), as some objects (for unknown reasons) are slightly misaligned. For example, all objects are loaded in at positions (row*2.1, column*2.1), so Clara would be loaded in at x=2.1, y=2.1 (for example). But a mushroom would be loaded in at x=4.200000000000001, y = 4.2000000000001 (instead of it being a clean x=4.2, y=4.2). Hence the check of "if(this.clara.position == this.mushroom.position)" would fail, as the coordinates are not "exactly" aligned. (Again, this could be overcome by passing a range of numbers for the check, but this method is far from reliable, and would require extensive testing).

   The time complexity of this search is not ideal. You have to check through every object in the scene (for(this.scene.meshes), only work with the mushroom objects (objects with a name=mushroom), THEN check whether it is the correct mushroom (based on location of x/y), hence the time complexity is very bad.

Final Movement Method (Implemented):

Using additional 2d grids for mushrooms and leaves:

Using this method, developer can reliably locate the relevant leaf/mushroom using the coordinates of Clara in the “movementGrid”. Clara’s current location allows you to directly fetch the relevant mesh (O(1) time complexity), and there is no chance for errors with this method (as with the previous method of checking coordinates of meshes).

Deprecated functions - Descriptions

RayCastMushroom: this function is only called during a mushroom move. A ray is cast from Clara for 1 frame and returns the first object in front of her. I attempted to prevent objects being selectable (e.g., floor tiles, random items that are generated on the floor), but this method is not consistent (sometimes it works, sometimes it doesn’t. the method used was making the mesh pickable = false when it is generated, but again, does not always work).

The ‘origin’ in this function, refers to Clara herself, and it is generated from the centre of Clara’s model, but at the bottom (at her feet). Hence when the origin is moved up 0.5 units (to avoid hitting other obstacles), the Clara model itself moves up. To avoid Clara’s position changing, I set the origin.y += 0.5, then once the function was complete, origin.y -=0.5. This happens fast enough that it is imperceptible.

This method is however unreliable, as sometimes rocks/long grass can get in the way of the ray, hence they are moved instead of the mushroom. Another problem was while the leaf is bobbing up and down, the ray has a chance to miss (you could cast multiple rays until you find something, but this method is far from ideal)

This function has been left in for future developers who could make use of the functionality, along with rayCastLeaf().

Conclusion

This report covers all functions and variables utilized in code, as well as deprecated code that is not currently being used, but may be used by future developers. The main points of this document included board generation, cloud generation, waterfall effects, movement, and camera controls. Also covered are the functional requirements that this project successfully implemented, known bugs and test results. Using this document, any future developer will be able to adapt and make further changes to the code in alignment with future project goals.