Tab 1

**Documentation**

**Contents:**

* [Scene Management](#kix.lchp79nw9icq)
* [Connections](#kix.z5q4w6csuwx7)
* [Objects](#kix.nzjl0ywydjjt)
* [UI](#kix.4t2m73dfhgbv)
* [Coding](#kix.l9mfzyj25ydu)
* [Level Creation](#kix.jl4goy1j8omw)

**Scene Management:**

**Entry Point Scene (Root Scene):**

This is the first scene and MUST be loaded at all times or the game will not work. It contains the three primary classes that need to be present for gameplay to function:

* Camera - this is responsible for displaying the game on the screen. It also creates and manages the CamTarget which can move around the level
* UI Manager - this is responsible for getting information about other UI components. It has functions to open and close the main UI as well as to reset it
* Connection Manager - this is responsible for creating the various kinds of connections and maintains information about all connections

The Entry Point Scene also contains a MainMenu loader object, whose sole purpose is to load the main menu scene. This is done with the level loader script.

**Level Loader (script):**

This script is responsible for loading scenes additively (similarly to adding on top of each other). It works by using the object’s name as the name of the scene to load. When loading a new scene, first it resets the current level state, then it unloads the previously active scene (but never the Entry Point Scene), then it loads the scene to be loaded.

**Level Manager (script):**

This script is responsible for maintaining information about the level and giving flexibility over the type of level. For example, it takes an enumerator LevelType, which corresponds to what the nodes will look like. Similarly, it takes a background colour, which will change the colour beyond the grid border. More importantly, it is responsible for spawning in the robot as well as its starting, milestone (activated by completing location’s (must be a Gate) goals), and ending actions.

**Note:** Gates MUST be passed in to the “Level Goals” list for them to count as part of the level’s goals. Gates contain individual goals, LevelManager contains a list of ALL the goals. Make sure to set EACH Gate that is passed into the “Level Goals” list.

**Connections:**

**Connection (script):**

The connection script controls a singular line prefab that represents a connection between two nodes. It has methods to start and end a connection, which take in a node point and a color. There are also methods to have it travel to a point, follow a point or mouse, even flip the ends.

**Connection Manager:**

The connection manager is the singular largest class in the project. Its primary purpose is to create and manage connections. It contains the logic for starting, ending, and the relevant validation for each of those points. The validation is used by the Robot class to send error messages to be displayed as dialogue to the user. In addition, there are also methods for manual versus automatic connections. Manual requires user input (e.g. clicking on nodes) whilst automatic is used primarily by the RobotController class and is invisible to the user.

**Objects:**

**Nodes:**

Nodes are structurally the same as one another, with only minor differences between them. They are organized with a parent object containing the relevant node class, box collider component, and a highlight script. They usually have children with their graphics, a gem display, and a status prefab.

Hierarchy

Node

- GFX

- Gem Display

- Status

**Node** parent keeps track of relevant information to node type

**GFX** child has sprite renderer component that gives the node its graphics

**Gem Display** child is responsible for displaying the large gem for the node. It has one method, UpdateDisplay() that updates the node’s gem to match what is passed into it

**Status child** is responsible for bringing up a UI specific to the selected node. Its attached script is set up to handle the menu and ensuring it displays the correct gem

**Node Specific Components:**

* **Lasers** have an additional laser pointer child which is an extra sprite that follows the direction the mouse points to. It is controlled by the Laser script
* **Gates** have an additional Gate\_UI child and Goal Progress grandchild that displays the amount of gems it currently has. **Note:** Gates MUST be passed a gem goal otherwise it will not accept any other gem.

**Node (Script):**

The Node class/script is responsible for how the Node objects behave in game. It is the parent class for all the different kinds of nodes, which inherit from it. The Node class contains methods for updating the display when new connections are added or removed, adding and removing connections, and resetting the node to its original state.

Here is a list of the Node’s children and types of nodes within the game, along with their modifications (if any) to the Node class:

**Gate (Script):** This takes in a goal gem, which it will then check against when receiving another gem. If the gems match, it will change its display and update the level goal UI

**Mine (Script):** This doesn’t modify the node class outside of making it not updatable

**Display (Script):** This changes to now accept two gems. Once it has two gems, it combines it into a singular gem

**Laser (Script):** This changes how connections are updated because of the nature of laser connections. It deals with some of the various cases that arise based on the type of connection the laser has

**Arrays:**

Arrays share some components with nodes but are abstracted in the editor to make level design easier. The prefab only contains an Array script that can set the location, size, and other properties of the array.

**Array (Script)**

The array script works by initializing the head as a laser node, then the body of the array which is made up of the array\_type nodes (set in the editor). Then, the array connects the laser to the body. Each element of the array is protected (they cannot be directly connected to but instead must be connected with laser connection). The script contains an OnDrawGizmos, which allows the array to be visualized in the editor before playtime.

**Rooms:**

Rooms are similar to Arrays in that they are initialized within the editor. The prefab only contains a Room script that can change the bounds of the room, location, and parameters. Rooms have logic to set all nodes inside them to have the “in\_room” status.

**Room (script)**

The room script works by initializing the walls of the room along the grid. Because of how the game is laid out on a grid with each element representing a 1x1 space, this works. Note: if larger sprites are to be used, this will need to be accounted for. In the logic for placing the walls, the room will also set up the parameters specified in the editor. This has been modified to take in shapes (before it was colors), but can also be reverted to take in other types of parameters.

**FX:**

This prefab has only one job: play its animation and destroy itself. It is currently used by Gates and Rooms when the correct gem/parameter is passed to it. It is set up to have multiple animations be associated with it for different kinds of effects.

**Camera Controller/Target:**

The Cam Target prefab has an attached movement script that moves it based on horizontal and vertical axes (in Unity, this corresponds to WASD or Arrow keys, but can be configured in the project settings or with custom key scripts). The main camera has an attached [Cinemachine](https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html) component, which is set to follow the Cam Target.

**Robot:**

Similar to Nodes, the Robot prefab has an attached Robot Controller script and three children: GFX, DialogueCanvas, and Gem Display.

Hierarchy

Robot

- GFX

- DialogueCanvas

- Gem Display

**Robot** is responsible for controlling the robot’s actions. The script can change the robot’s speed and cooldown between actions.

**GFX** has sprite renderer component that gives the robot its graphics

**Dialogue Canvas** child is responsible for displaying any dialogue the robot speaks. It has options for choosing if the dialogue is clickable and whether to resize the dialogue box based on the text

**Gem Display** child is responsible for displaying the large gem for the robot. It has one method, UpdateDisplay() that updates the node’s gem to match what is passed into it

**Robot (Script)**

This is a large controller class that has several generic actions the robot can do as well as specific actions. Each action has an ActionType enumerator, which is located at the top of the class. The way the actions work is in the Update() function, if the robot is not already in an action and there are actions in the queue, perform the next action.

To make doing actions easier, a separate RobotAction class has been created which keeps track of the specific action’s information. Each RobotAction must have: a location and an action type. Based on the action type, you can specify additional data for the action. For example, if the action is “speak,” then you can pass a string of dialogue. For actions that require a secondary node (move gem, modify/break/add connection), you can specify the node as well.

**Note:** there are no checks in the action class to ensure that the action has the correct information (e.g. you specify the ActionType to be “move gem” but don’t pass in a node for the secondaryNode parameter)

Specific actions for the robot have already been pre-defined. These include:

* Speak - robot speaks. Dialogue will last for 1.5 + (length of dialogue \* 0.1) seconds
* SpeakClickable - robot speaks. Dialogue must be progressed by clicking
* MoveGem - moves gem from location to secondary node
* PointLaser - points laser from location (which is a laser) to secondary node
* TransferToTransfer - does a transfer-to-transfer connection from loc to secondary node
* TransferToDeref - does a transfer-to-deref connection from loc to secondary node
* DerefToDeref - does a deref-to-deref connection from loc to secondary node
* DerefToTransfer - does a deref-to-transfer connection from loc to secondary node
* BreakConnection - breaks connection at location

**UI:**

**UI\_Manager:**

A scene management class responsible for getting information about other UI components. It has functions to open and close the main UI as well as to reset it (used by level manager).

**Terminal:**

The terminal essentially works as a custom button that opens the menu when clicked on, and otherwise highlights when hovered over. When the terminal is open, it checks every time a newline character is entered or the user has clicked somewhere on screen with a LineEntered() method. This method then creates and calls the RobotInterpreter class to handle the parsing of the line.

**Pause Menu:**

This class is responsible for opening and closing the pause menu located at the top right of the screen. In addition to OpenMenu() and CloseMenu(), it contains methods for each of the options it opens to: reset level and settings.

**Note:** exit level is a scene changer prefab so it has its own dedicated class to deal with exiting a level

**Level Goal:**

This class is responsible for displaying the goal gems in the top left of the screen. It also contains text and methods for updating the text to give players information about their current progress in the level.

**Note:** There are also methods created for when a player clicks on the goal icon, it will open a drop-down menu with more details about each of the required gems. It is currently disabled with a comment left with instructions on how to enable it.

**Dialogue System:**

This class is responsible for displaying and updating dialogue. It is designed to be flexible with the selected options, those being: auto-resize which will resize the dialogue box based on the amount of text and clickable dialogue, which will prevent the dialogue from progressing to the next line until the user has clicked on the box.

**Note:** this class requires dialogue to be in a string[] array, and has the method UpdateDialogue() which can take in a char separator (default is set to ‘\n’)

**Completion Menu:**

This pulls up a menu with two buttons: Main Menu and Next Level, both of which transition the scene to the corresponding button name. Upon opening the completion menu, it checks for valid scenes by going through the LevelLoader class’ scene list.

**Coding:**

**Robot Interpreter:**

The Robot Interpreter is responsible for taking a string and parsing it to check if it is a valid line of code. If the statement contains two valid nodes, it will then attempt to make a connection. Because of the validation logic in the connection manager, it doesn’t need to check for if the code is valid since the connection manager will display the relevant error to the player. The Robot Interpreter uses a series of boolean values to check for the type of connection, then creates and sends a RobotAction to the robot.

**Level Creation:**

Level creation is made to be as straightforward as possible. In order to create a level, there must be at least three things present:

* A Level Manager - this will configure the level, spawn the robot, and set the goals with Gates
* A Grid - this is how the level looks and defines the space of the level. It also should have walls to bound the Cam Target
* Nodes - these are the components players will interact with during gameplay. Without them, there would be no level.

**Note:** when creating a level, it is easiest to copy over an existing level, delete the nodes, then repaint the grid.