Rochester Steampunk AR Unity Project Manual v1

Spring 2020

Table of Contents

I - Getting Familiar With The Project

What is Rochester Steampunk AR?

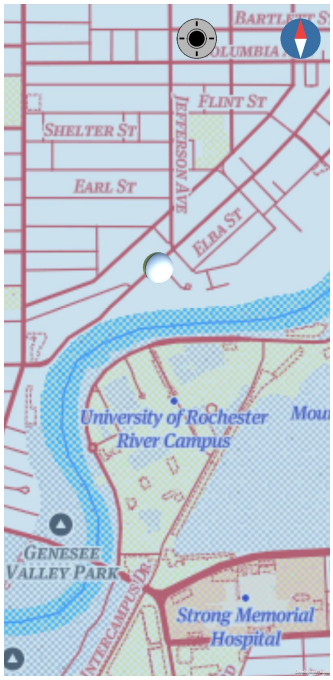
A location-based narrative experience for mobile devices and a tool that can be used to create new narrative content/experiences.

What Tools Does Rochester Steampunk AR Use?

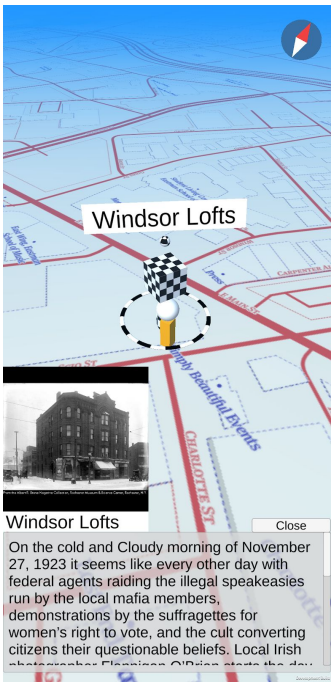
Unity game engine for development and Mapbox for location-based functions.

What Does Rochester Steampunk AR Look Like?

Main Menu



Map View



Waypoint Interaction

[Link to Gameplay Video]

[Link to Android Build (APK)]

What Is the Status of This Project?

[Link to Current Project Summary]

II - Documentation

Here are useful sources to consult when working on this project:

Technical:

Unity Documentation

Unity Forum

Unity3D Subreddit

Mapbox Documentation

Mapbox Unity SDK Info Page

Mapbox Forum (Stack Overflow)

Mapbox Subreddit

Mapbox Studio Manual

Design, Process, and Storytelling Practices

Designing Games, Tynan Sylvester (Print)

Collaborative Worldbuilding for Writers and Gamers, Trent Hergenrader (Print)

Interactive Storytelling, Chris Crawford (Print)

Digital Storytelling, Carolyn Miller (Print)

Jessika Weber's Thesis on Location-Based Games for Tourism (PDF in drive)

III - Unity and Project Setup

Software needed:

Win10/macOS:

- Unity Hub
- Unity version 2019.3.7f1
- Rochester Steampunk AR Project Folder

Android/iOS:

- Unity Remote 5 (Android/GooglePlayStore) (iOS/AppStore)
- Static Fake GPS Position App (Android/GooglePlayStore) (iOS/AppStore)
- Fake GPS Path Simulator App (Android/GooglePlayStore) (iOS/AppStore)

Hardware needed:

- Win10/macOS computer for working with Unity
- Android/iOS mobile device (phone/tablet) for builds and running Unity Remote
- Cable for connecting mobile device to computer running Unity

Additional:

- Access to the Rochester Steampunk AR Google Drive

Notes:

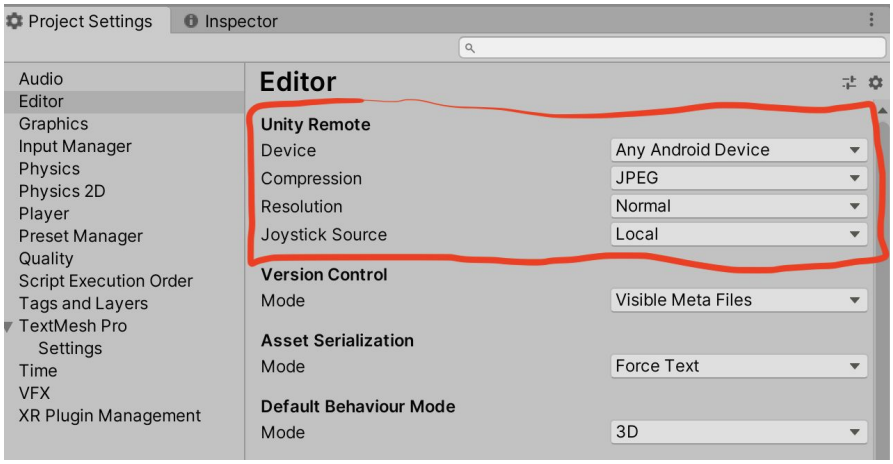
After installing Unity Hub and downloading the correct version of unity with android, iOS, windows/mac/linux standalone modules, download and unzip the Rochester Steampunk AR project folder. Point Unity Hub to the folder and open it.

IV - Testing & Running the Project In Unity Editor

Step 1 - Unity Checklist:

- Make sure Unity Remote settings are enabled,

  Edit > Project Settings > Editor > Unity Remote > Any Android Device OR Any iOS Device (change based on test device)



- Make sure you have a valid Mapbox Access Token loaded into your project

  Mapbox > Setup > Access Token (Make an account at Mapbox.com if you don't have a valid token, access tokens can be found under the Account tab)

Step 2 - Connect Mobile Device to Computer Running Unity

- If using Android make sure developer options are enabled & USB File Transfer is enabled

Step 3 - Run Unity Remote App

- If needed, run fake GPS app as well

Step 4 - Click Play Button in the Editor

- Make sure current scene open is ScenarioSelect menu scene
- If Unity game view doesn't appear on your phone, try restarting Unity to apply Editor Unity Remote settings

Notes on Unity Remote 5 for Testing:

- Touch Input isn't processed, touches are processed as mouse clicks
    - This affects the QuadTreeCameraManagement script that handles panning & zooming, mouse and keyboard input is handled differently than touch input

Notes on Testing:

- Mapbox has faux gps coordinate input capabilities, but they are harder to implement than using unity remote
- Use builds to test touch-related functionality
- Getting stuck on the loading screen indicates that Mapbox has failed to initialize the map, make sure that you give the app location service permissions

V - App Features

Opening the app for the first time prompts the user for storage and location permissions. Storage permissions are used for the system that parses game data from files, while location services are accessed by Mapbox.

Scenario Selection Menu

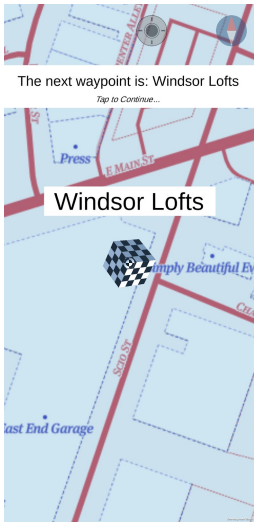
This is the first screen users see. There are three buttons not yet implemented: tour mode, collection, and settings. There is a scrolling menu that contains buttons used to select a scenario to play. The initial build has the first scenario repeated to demonstrate the scrolling menu and positioning of buttons.

Information on each button displays the scenario, it's title, and the completion percentage. Tapping a scenario button loads the next scene.

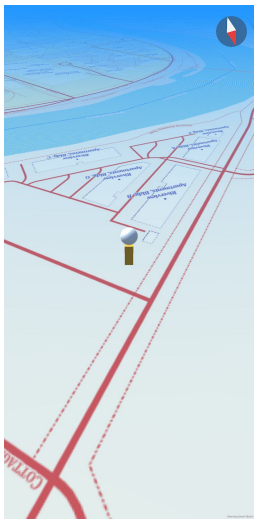
Main Game

(Intro)The main game display is used for the rest of player interactions with this app. When the scene loads, the player receives a scenario intro message, and is then presented with the path selection menu. After selecting a path, a path intro message is displayed.

(New Waypoint) Each time a new waypoint is unlocked, the map pans to the waypoint location with the overhead camera view. The user is prompted with "The Next Waypoint is *Location Title"*. After displaying that message, the camera remains centered on the waypoint. The user can pan and zoom around the area to see where the waypoint is, or return to their position/character by tapping the compass or snap button.

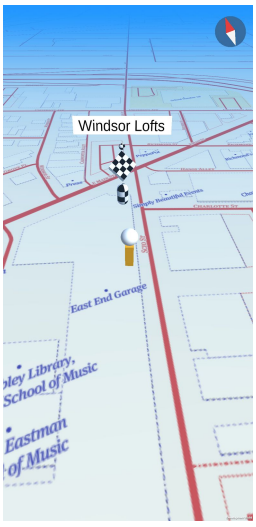


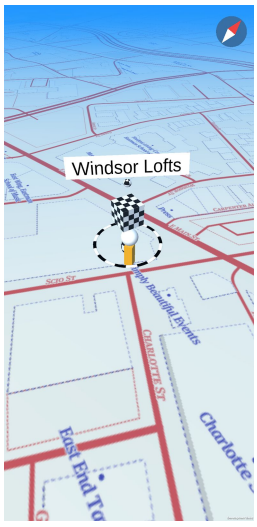
(Camera Views) There are two camera views the user can switch between, a close view that follows the player character, and an overhead view that displays more of the map and initially follows the character.



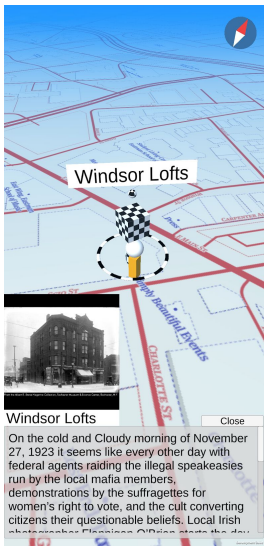
(Close) The close view has a better view of the character 3D model, and allows the player to orbit the camera around their player by swiping left or right. After 5 seconds of not touching the screen, the camera will return to the original orientation behind the player. There is a single UI element, the compass, which rotates based on the user's heading or the direction their device is facing. Tapping this compass icon changes the view to the overhead camera.

(Overhead) The overhead view is more useful for searching the map for waypoints and planning a route. This view will follow the player by default, but after the user makes a big swipe on the screen, the camera will 'break' from following the character, allowing the user to pan around the map. Dragging with a single finger will pan the map around, and there is no limit to how far the user can pan the camera. While in panning mode, a target button UI element will appear, which will snap the overhead camera back to the user's location. In this mode, the compass icon will constantly face North. An additional function in the overhead view is zooming. Using common pinch-to-zoom controls, the map can be zoomed within constrained levels. This is useful for comparing the user's position to waypoints.





(Waypoint Range) As the user gets in range of a waypoint, the waypoint model will transition from a resting animation to an active animation. When the waypoint completes the transition, the user can interact with the waypoint by tapping on it.



(Waypoint Interaction) When a waypoint is in range and the user taps it, the narrative UI will appear with an image representing the location or narrative event, the title of the location, and the narrative segment at that waypoint. Tapping on the waypoint again will hide the UI, but tapping on the 'close' button will cause the waypoint to be collected. When a waypoint is collected, it will disappear from the map, and the camera will pan to the next waypoint.

(Game Loop) After the user reaches the last waypoint and closes the narrative UI, the game displays a message telling them they have completed the current scenario path. The scene reloads, skipping the intro UI and going right to the path select menu, which will prompt the user to select another faction to play as. When all paths are complete, the app will reload to the main scene. At this point there is no way to return to the main menu while playing, so users have to complete all the paths in a single play session. When progression saving is implemented, the user will be able to close the app and continue where they left off at a later time.

VI - The Unity Project

Project Organization:

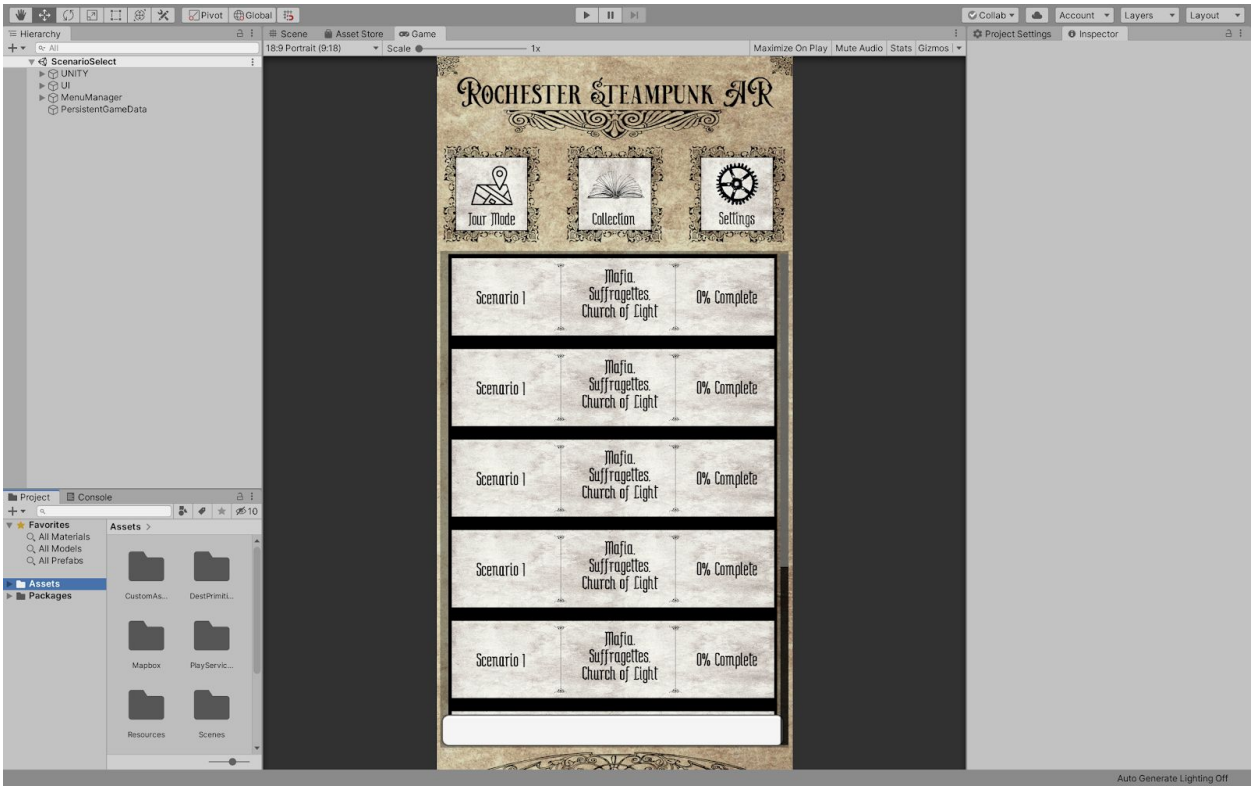
Assets Folder - contains everything related to developing this project

- **Custom Assets Folder** - assets used in the project that aren't from unity packages or third party plugins
  - **2D Assets** - textures, UI elements, and scenario images
  - **3D Assets** - empty (will hold POI models)
  - **Fonts** - OTF file types and Text Mesh Pro font assets
  - **GameData** - versions of coordinate and narrative files for editing
  - **Materials** - materials made from assets in textures folder
  - **Prefabs** - prefab assets like waypoints, path buttons, etc.
  - **Scripts** - scripts written for this project in C#
- **DestPrimitives** - third party content from asset store, used to generate primitive 3D models (the torus on the waypoint prefab)
- **Mapbox** - Unity SDK 2.1.1 notable folders include prefabs and examples, which contain insight on how Mapbox is meant to function
- **Play Services** - editor-based Google Play Services
- **Resources** - contains Mapbox configuration file
- **Scenes** - project scenes, there are currently 3: MainGame, ScenarioSelect, and ScrollRectSnapDev
- **Streaming Assets Folder** - used to load files at runtime on mobile devices, the coordinate and narrative files are stored here
- **TextMeshPro** - Unity's TextMeshPro plugin, a high-quality and more efficient way to handle text in UI elements than the default Unity UI Text
- **ThirdPartyAssets** - contains a line ending fixer that comes with Mapbox (WoLfulus)

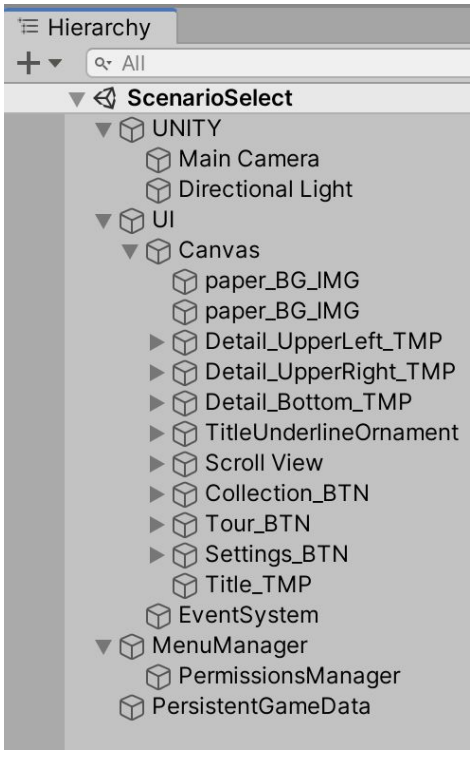
Packages Folder - contains and manages content added via Unity package manager

Scenes

Scenario Selection Scene - prompt for permissions, find and parse game data files, pass scenario index player selects and parsed gamedata into the next scene; UI focused.



Scenario Selection Hierarchy:



UNITY: main camera, and directional light for the scene.

UI: holds majority of components in this scene.

Canvas: Unity UI canvas object.

paper_BG_IMG: canvas background .

image, currently set to paper sprite in 2D folder.

Detail…: embellishments created with the victorian ornament TMP asset.

TitleUnderlineOrnament: another victorian ornament TMP asset.

Scroll View: Unity scroll rect object, contains scenario buttons and handles scrolling.

Collection/Tour/Settings_BTN: not yet implemented button objects.

Title_TMP: title text mesh pro field.

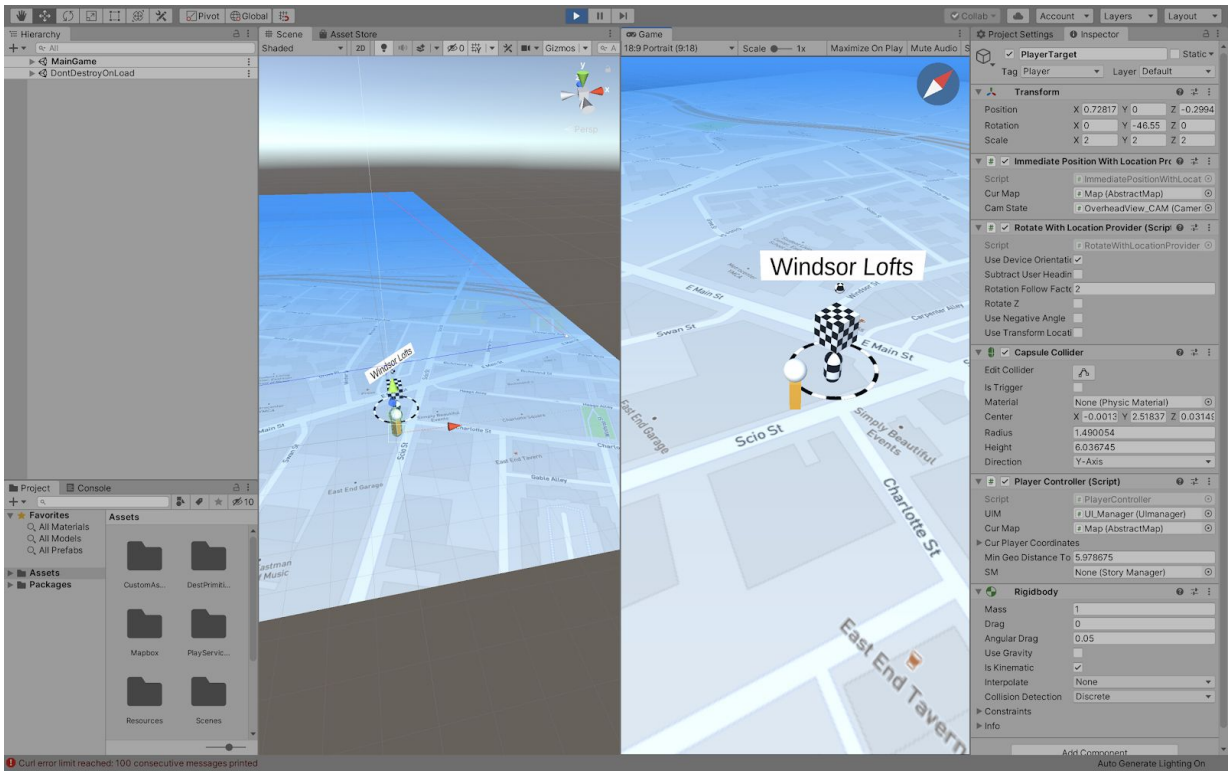
EventSystem: handles input/UI interactions.

MenuManager: used for organizing, holds object with permissions checker script.

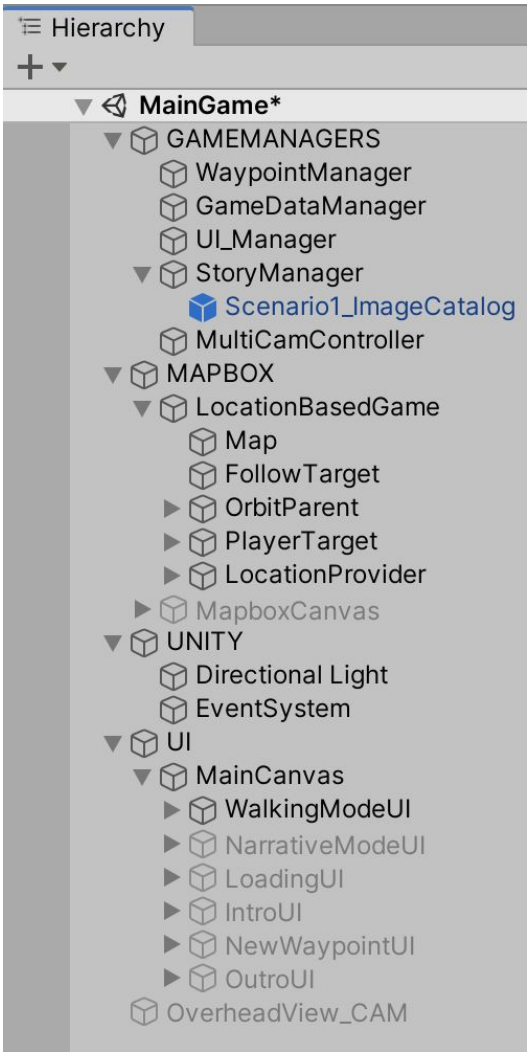
PermissionsManager: has script component that ensures require permissions are granted before starting the game.

PersistentGameData: GameObject that isn't destroyed in between scenes, holds game data file paths, parsed file information, and other relevant information that needs to exist between each scene.

Main Game Scene - generates waypoints based on coordinates linked to selected scenario, populates waypoints with narrative story segments, images, and location titles. Render and update map visual based on player location, camera position, and pan/zoom levels. Track session path progression and reload the scene when the last waypoint is reached.



Main Game Hierarchy:



GAMEMANAGERS: gameobjects with manager scripts.

WaypointManager: organizes waypoints and the data stored at each location.

GameDataManager: handles storage of persistent game data in the scene .

UI_Manager: organizes and controls UI states.

StoryManager: sequences and tracks progression through the narrative.

Scenario1_ImageCatalog: prefab containing an array of the images relevant to the current scenario

MultiCamController: controls different camera states.

MAPBOX: location-based game Mapbox elements.

LocationBasedGame: parent object.

Map: numerous Mapbox scripts, responsible for rendering the map data, spawning waypoints on the map, computing pan/zoom commands, and adjusting map settings in the editor and at runtime

FollowTarget: unused follow transform for cameras.

OrbitParent: rotating this game object orbits the camera around the player, follows player via script, and contains the close view camera object.

PlayerTarget: player gameobject, has Mapbox scripts for controlling movement/rotation, rigidbody and player controller script to register collisions with waypoint objects. Contains the 3D elements of the demo character, as well as transforms the main cameras use to follow the player.

LocationProvider: handles position data input from different sources and formats/passes data to Mapbox systems.

MapboxCanvas: disabled, carried over from the mapbox prefab.

UNITY: directional light and event system objects

UI: parent element.

MainCanvas: single UI canvas for the game.

WalkingModeUI: parent of compass button and location snap buttons.

NarrativeModeUI: parent of narrative image, location field, and story text field.

LoadingUI: loading screen displayed before Mapbox fully initializes.

IntroUI: scenario and path intro messages, path selection menu.

NewWaypointUI: message displaying the location title of the next waypoint.

OutroUI: message telling player they have completed the current path/scenario.

OverheadView_CAM: camera view that better displays map information, used for map panning/zooming.

Project Prefabs

There are currently 6 types of prefabs used in the project:

- **1_0_ChurchOfLightBTN:** path selection menu UI object.
- **1_1_SuffragettesBTN:** path selection menu UI object.
- **1_2_MafiaBTN:** path selection menu UI object.
- **Intro_PathSelect:** contains path selection menu UI elements.
- **Scenario1_ImageCatalog:** an array containing images linked to scenario 1.
- **WaypointPrefab:** waypoint gameobject, with scripts that hold the fields populated with parsed waypoint location, story, and image data.

Script Summaries

There are 25 C# scripts made specifically for this project:

- **CameraFacingText:** controls the camera facing text label within the waypoint prefab, currently has issues tracking camera in orbit mode.
- **CameraFollow:** essentially a camera manager. Handles camera states for specific cameras, enables player following. Most functionality and control transferred to MultCamController.
- **CameraManager:** camera pan/zoom controller, was replaced with Mapbox QuadTreeCameraMovement for better functionality with the map.
- **CamOrbitController:** rotates the orbit camera parent object based on delta of a touch input drag.
- **DontDestroyOnLoad:** used for keeping persistent data between scenes.
- **DontDestroyOnLoadAccessor:** no longer used, served as a fix for an issue with the persistent data object.
- **FollowObject:** used to allow the orbit parent to follow the player object without parenting.
- **GameDataManager:** populates location list, holds LocationData class, most functionality moved to SpawnOnMap.
- **GameDataParser:** finds filepaths based on runtime device, parses data from files. Holds StorySection class, which contains the scenario tag, faction tag, location name, narrative text, and sequence index of each story segment. Note: the JSON parser needs the value names and class name of StorySection to be the same as in the StoryText.json file to parse successfully.

- **LoadingIcon_Animator:** animators the radial fill amount of the loading icon image.
- **MultCamController:** manages different camera states, switching between physical camera objects, enabling orbit camera function, and lerping the camera to waypoints.
- **PermissionsChecker:** ensures that the user grants the application relevant permissions before launching (Android devices).
- **PlayerController:** sends waypoint data to populate UI elements when waypoint detects a collision with the player gameobject.
- **RotateCompassIcon:** rotates the compass icon UI object based on the player's heading/direction
- **ScenarioImageCatalog:** stores an array of images and an index tag for accessing and organizing the narrative images in a scenario.
- **SecnarioPathBTN:** used to send selected path data to StoryManager when a button is clicked
- **ScrollRightSnap:** handles the scrolling and 'snap to element' function of the path selection menu.
- **StateManager:** unused.
- **StoryManager:** organizes the story sections relevant to the selected path, spawns an image prefab based on the current scenario, handles story progression and finding the next waypoint in the path sequence.
- **TextSizer:** unused. Resizing text in editor.
- **UImanager:** manages the different UI elements in the main game scene, populates the path selection menu with button elements, populates the narrative UI with waypoint relevant text, hides loading screen UI when Mapbox AbstractMap is initialized.
- **UpdateCamRotator:** unused. Created when building out orbit camera functionality.
- **WaypointAnimator:** animates the different 3D models in the demo waypoint prefab, waypoint states, detects if player enters waypoint activation range.
- **WaypointData:** holds waypoint coordinates, location name, label text, current StorySection at the waypoint, and a collection of StorySections that exist across all paths at that location. Detects taps/clicks on the waypoint if the player is within range.
- **WaypointManager:** populates SpawnOnMap with the location coordinates to spawn waypoints, assigns each waypoint StorySections based on their location title.

Three of the Mapbox scripts were edited for this project, take care not to overwrite them when importing new versions of Mapbox or reimporting the MapboxSDK.

- **AbstractMap:** didn't directly edit the script, but attached UImanager.HideLoadingUI, UImanager.ShowIntroUI, and WaypointManager.SpawnWaypoints to the OnInitialized event. This allows greater control over the order of events reliant on Mapbox initializing the map.

- **SpawnOnMap:** moved spawning gameobjects from the Start method to control the timing of when the gameobjects are spawned, adjusted the scaling of gameobjects based on the zoom levels of the current AbstractMap.
- **QuadTreeCameraMovement:** limited when processing input can occur (ie. panning can't occur when the camera is in close view), direct input detected from single touches in close view camera mode to the OrbitController.

VII - Working With Mapbox

Getting Mapbox Functionality in Scripts

Depending on what functions you need, add the following namespaces to the script you are creating ('using' prefix):

- **Mapbox.Unity.Map** : access AbstractMap.cs, useful for checking & adjusting the zoom level, camera being used for map tile rendering bounds, and accessing other variables on the current map.
- **Mapbox.Unity.Utilities** : allows you to use the Conversions class, which is important when converting coordinates from Unity world space to Geo-Coordinates on the map and vice-versa.
- **Mapbox.Examples** : used for referencing the scripts Mapbox includes with their example scenes. This was utilized for accessing the QuadTreeCameraMovement script, which simplified the process of zooming and panning the camera.

Location-Based Games and Mapbox

The initial build of Rochester Steampunk AR is based on the location-based game prefab included with the Mapbox SDK. This prefab comes with an AbstractMap, Player object that updates based on GPS coordinates, and a Location provider component that converts GPS data to values usable in the Unity scene.

The AbstractMap gameobject has SpawnOnMap and QuadTreeCameraMovement components added. SpawnOnMap handles instantiating the waypoint prefab on the map, and scaling/adjusting their positions based on the zoom level of the map. Adjusting the zoom level of the map while doing a task like lerping or panning the camera can cause issues with lerp positioning, as the coordinate space scales with the map zoom level. QuadTreeCameraMovement handles the panning and zooming functionality of the map.

AbstractMap loads map graphics as tiles, and there are different bounding options that can be used to control how many tiles are loaded at one time. Rochester Steampunk AR uses camera bounding, which tells mapbox to fill the view of the camera with map tiles. This is fine except for when the main camera gets a large sight line in the scene, Mapbox tries to load a large/infinite amount of map tiles and the app/editor can crash.

The player object has been given empty gameobjects as children that the CameraFollow script uses to track the player's position from different angles.

Custom Map Styles

Custom map styles help stylize location-based games and create unique themes. Everything from colors, textures, and information displayed on the map can be edited with the Mapbox Studio tool. Rochester Steampunk AR's initial build features a custom map style from the Mapbox website. Standard is a map style based on 1920's maps created by the Standard Oil Company, fitting for the game's steampunk aesthetic. Mapbox Studio is a straightforward tool that lets artists

and graphic designers easily create custom map styles to fit narrative content for the app. Switching the map style out is as simple as adjusting the URL in AbstractMap.

Adding 3D Buildings to a Map

The development chart for Rochester Steampunk AR includes two types of 3D models to be added to AbstractMap. The first type, at places of interest (POIs), can be added to the map with the same SpawnOnMap technique currently used for adding waypoints. This process gets tricky if builds are extruded from AbstractMap, and the POI models will need to be added with the ReplaceFeatureModifier aspect of AbstractMap.

Extruding simple 3D buildings from the map is also simple if done separately from adding buildings with SpawnOnMap:

- Open the MainGame scene
- In the Hierarchy window go to MAPBOX > LocationBasedGame > Map
- Select Map and in the Inspector window, click MAP LAYERS
- Open the FEATURES menu > Add Feature > Buildings
- Select the new buildings feature in the Map Features chart to access modeling and texturing menus

The only downside to generic 3D buildings is that they tend to detract from the information on 2D maps, and may make it difficult for users to find waypoints and plan routes.

Example Scenes

Use these scenes as reference when trying to figure out how the different pieces of Mapbox interact with each other. The panning and zooming mechanics in Rochester Steampunk AR are derived from the 6th example scene, Zoomable Map.

VIII - Process for Adding New Narrative Content

Adding Narrative Text

The dynamic narrative text system allows new stories to be added through a specially formatted JSON file. The narrative is broken up into segments based on the location of the events they convey. By using pointer values within the JSON file, each segment can be further classified based on: scenario, sequence within the scenario or path, faction/path, location string, and an image from the image catalog.

The following is an example of the format within the StoryText.json file:

```
{
        "scenario":"1",
        "sequence":"0",
        "faction":"0",
        "location":"corinthianhall",
        "image":"0",
         "text":"Corinthian Hall, On the cold morning of November 27, 1923…"
},
```

Even with the pointer values represented as numbers, there is a verbal tag that makes the information in the file easy to read for people. The file itself can be edited in notepad, and doesn't require programming experience. The example above belongs to the initial scenario, is the first piece of text in the scenario path sequence, is tied to the Church of Light faction previously assigned the index value of 0, is placed at the location Corinthian Hall, utilizes the entry of the scenario's image catalog at the index of zero, and has a single plain text string containing the story segment.

Besides adding narrative text, the pointer values can be manipulated to add scenario or path specific text for UI elements. The snippet below uses negative sequence, faciton, and image pointers to add an entry that includes text for the initial scenario intro:

```
{
        "scenario":"1",
        "sequence":"-1",
        "faction":"-1",
        "location":"",
        "image":"-1",
        "text":"Follow three Democrat & Chronicle reporters in 1920's Rochester, NY…"
},
```

It is important to have the factions, sequences, and images sequenced before making additions to the StoryText.json file. This means making a list of each data type, and assigning values starting at 0 to each element. Retaining the original order when implementing these lists in Unity is important to prevent mixing up content. These lists can be created with tables in a google document as part of the pre-development content creation process.

The information from StoryText.json is found and parsed with the GameDataParser script. There are different paths depending on what the runtime device is, the script detects and applies the correct path modifiers. GameDataParser then converts the input from each section of the JSON file into StorySection objects that mirror the format of the file (same variable names and types) and adds the object to a collection. From this point, the WaypointManager script assigns each story segment to a waypoint by comparing the location field of the StorySection with that of the waypoint.

Future teams may find that it is easier to keep separate story files for each scenario. This would make it easier to manage specific scenarios, reduce the likelihood of errors impacting other scenarios, and require minimal development to implement. The downside would be more files to manage and parse, which creates more room for errors.

Adding Coordinates and Locations

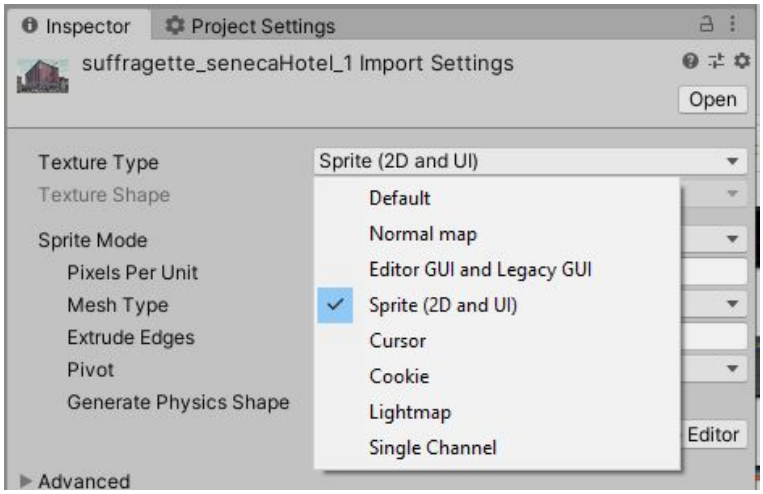
The current system for adding locations and coordinate points is dynamic and will work for any scenario, but isn't as refined as the process used to add narrative text. Coordinate entries can't be filtered by scenario within the file, but the StoryManager script completes the filtering process during runtime. The data tied to each coordinate point is limited to a location title and the coordinate pair, each entry is added to a new line in the file:

Corinthian_Hall,43.1562269,-77.61291840000001

This coordinate entry represents Corinthian Hall, with a latitude of 43.1562269 and longitude of -77.61291840000001. Mapbox utilizes latitude and longitude coordinate pairs as strings, which it then parses into numerical values. The text component of each entry is used to assign the waypoint StorySection data objects.

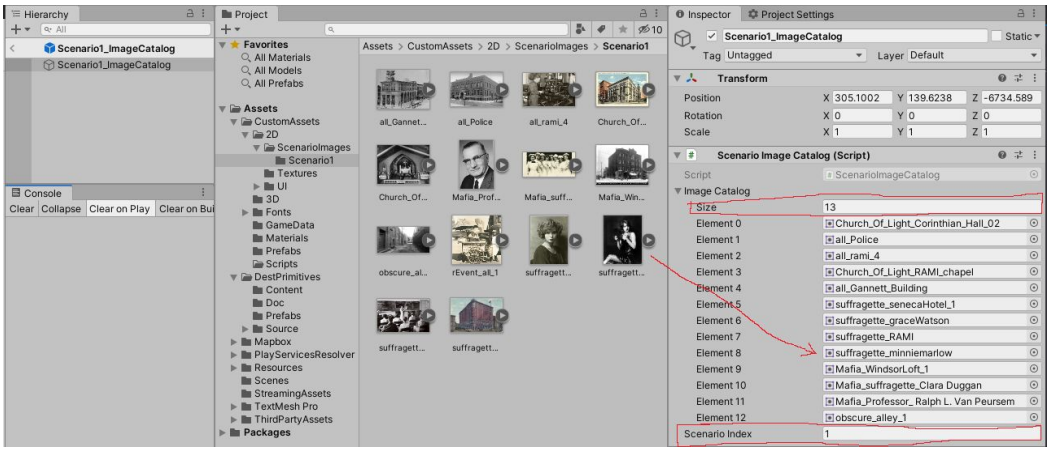
Adding Images

When a player accesses the story section at a waypoint, an image is displayed with the narrative text and location name. The images for each scenario are stored in separate folders (Assets > CustomAssets > 2D > ScenarioImages > Scenario#). When adding images to a Unity project, make sure to set the texture type for each image to Sprite (2D and UI) by clicking the image in the project window and editing the texture type field in the inspector window.

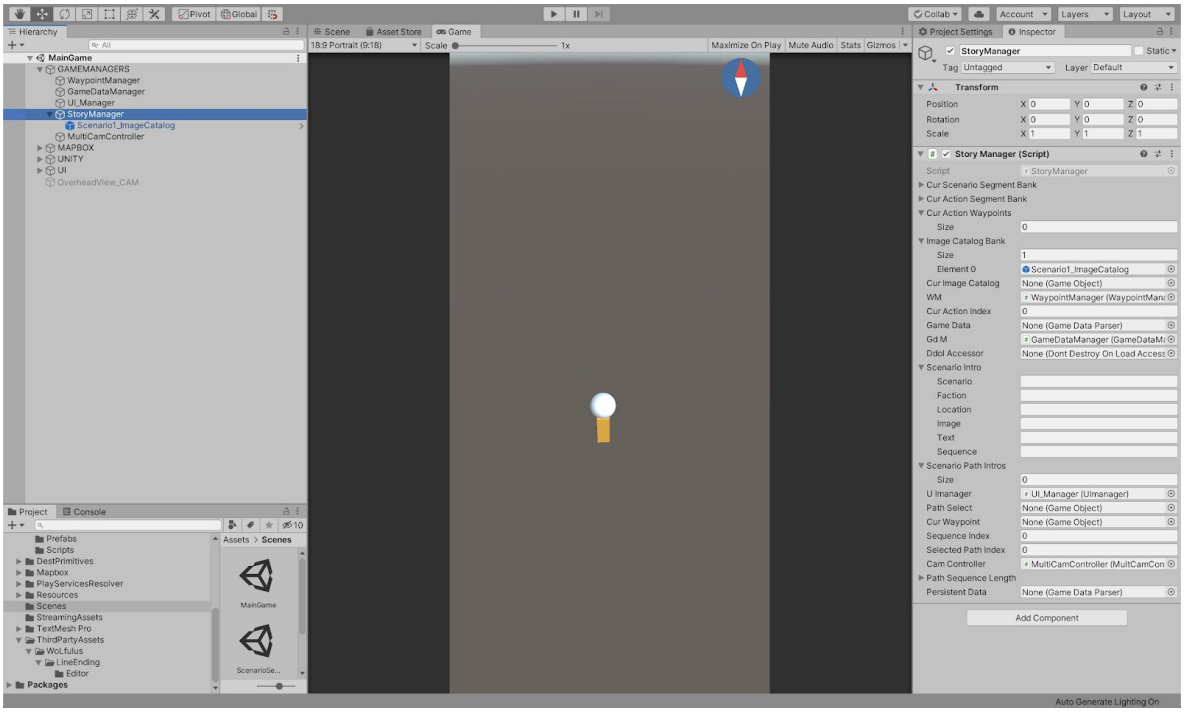


Use the previously created indexed list with the location/event images to create a new CatalogImagePrefab (CustomAssets > Prefabs > Scenario1_ImageCatalog). Duplicate a pre-existing prefab and select the 'open prefab' option in the inspector window to access the prefab editor. With the main prefab selected, use the inspector window to edit the Scenario Index

and Image Catalog fields. Define the number of images used in your scenario with the Size field under the Image Catalog drop down menu, and drag images from the project window to their corresponding indexed locations. Save the prefab (Ctrl+S).



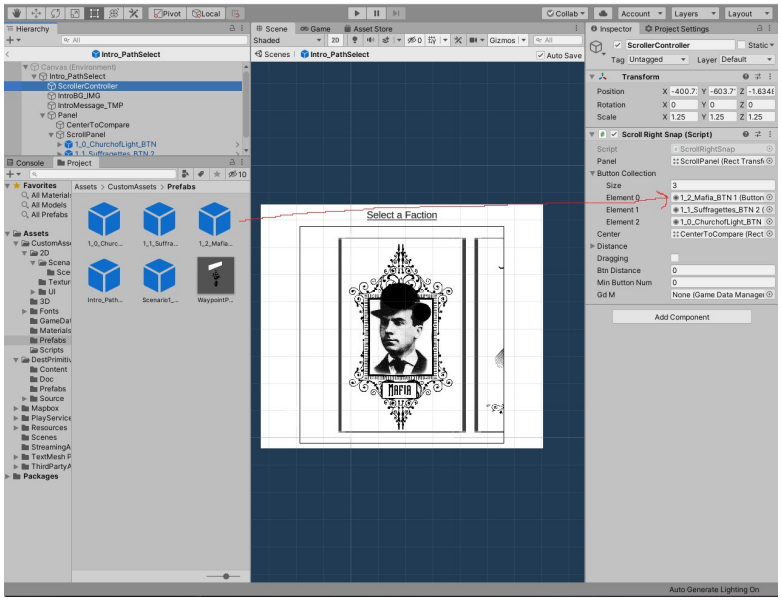
The last step is to add the new ImageCatalog prefab to the Main Game scene. Drag the prefab from the project window to the Hierarchy window. The ImageCatalog prefab should be parented to the StoryManager object within GAMEMANAGERS. Select the StoryManager object, and edit the ImageCatalogBank field, increase the size and drag the new catalog prefab from the Hierarchy window to the new element within ImageCatalogBank. The aspect ratios of each image are preserved when they are used to populate the NarrativeUI image component.



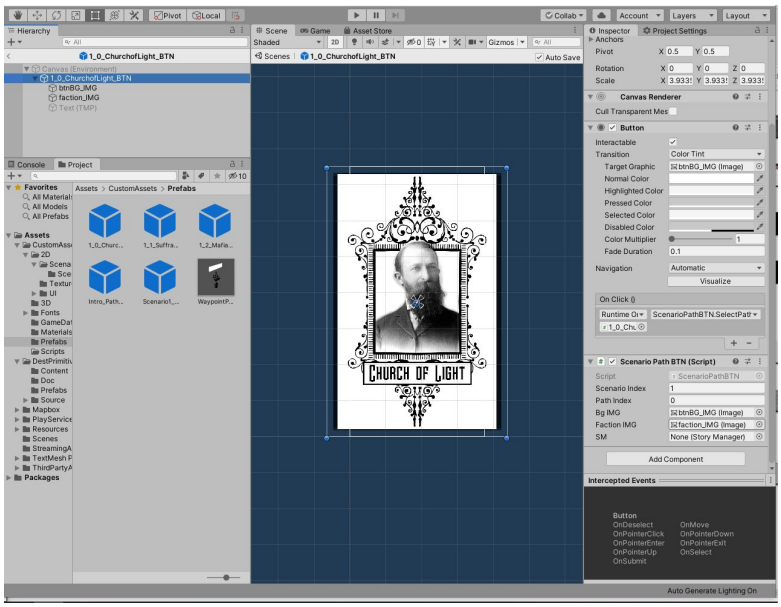
Adding Scenario Paths and Sequencing

After adding the new scenario's narrative data to StoryText.json, coordinate data to LocationCoordinates.txt, and images to a new ImageCatalog prefab, it is time to implement the different paths players can take within the scenario and the sequence of the story segments within those paths. This part of the process requires the most technical interactions with Unity and still relies on some hard-coded systems.

The first step is creating button prefabs that represent each path available in the new scenario. Start with duplicating one of the existing buttons in Assets > CustomAssets > Prefabs > 1_0_ButtonName_BTN.prefab. The naming convention of each button denotes the scenario it belongs to, the index the path represents, and the name of the path/faction. Replace the source image components of faction_IMG and btnBG_IMG to customize the main image and button background. A text element can be added if a created button image doesn't have the name of the path in it already. Select the main game object of the prefab in the Hierarchy window (1_0_ButtonName_BTN), and find the ScenarioPathBTN script component in the Inspector window. Edit the scenario and path index fields with values reflecting the new scenario. The BgIMG, and FactionIMG fields are for later development when creating path button prefabs is simplified. The SM (StoryManager) field is automatically filled by the ScenarioPathBTN script. Save the prefab and return to the CustomAssets prefab folder.



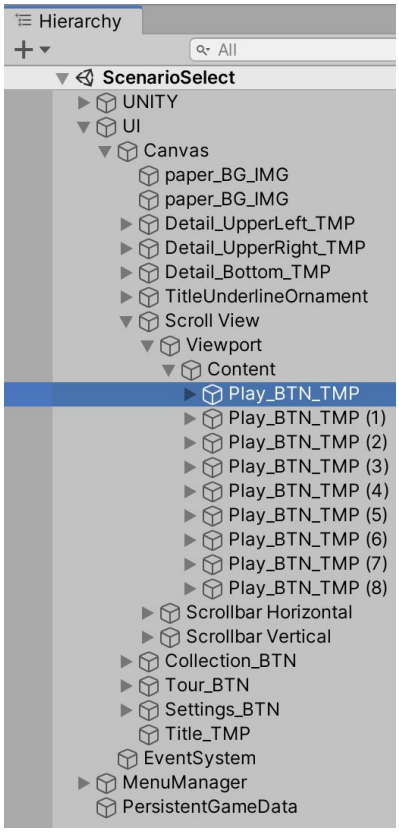
Open the prefab Intro_PathSelect, which contains the menu used for selecting a path within the current scenario. Select the ScrollerController gameobject and edit the ButtonCollection size field to the number of path option buttons being added. Drag the new button prefabs created to each new element in the ButtonCollection field.



Scenario Selection

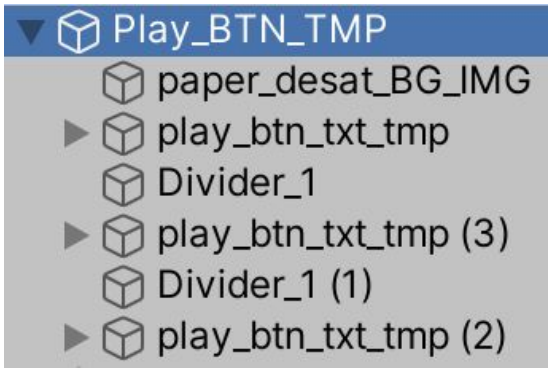
The final step in adding new scenario content to the Rochester Steampunk AR project is to add scenario select button to the main menu. This step will pass a scenario tag into the next scene, which allows StoryManager, WaypointManager, and SpawnOnMap to filter data relevant to a single scene.

Open the ScenarioSelect scene. Go to UI > Canvas > ScrollView > Viewport > Content > PlayBTN_TMP. The initial build has a repeating list of buttons for the same scenario. Delete all but two of those buttons (PlayBTN_TMP), the second button will be repurposed for the new scenario. If the buttons are already removed, simply duplicate one of the existing buttons.



PlayBTN_TMP contains the following components:

Play_BTN_TMP: TextMeshPro Button UI objects, the OnClick() event passes a scenario index tag to the persistent data object to carry on to the next scene and checks to make sure all necessary permissions are granted before launching the game scene.



paper_desat_BG_IMG: the background image for the button.

play_btn_txt_tmp: middle text field, used for listing factions.

Divider_1: first text divider graphic.

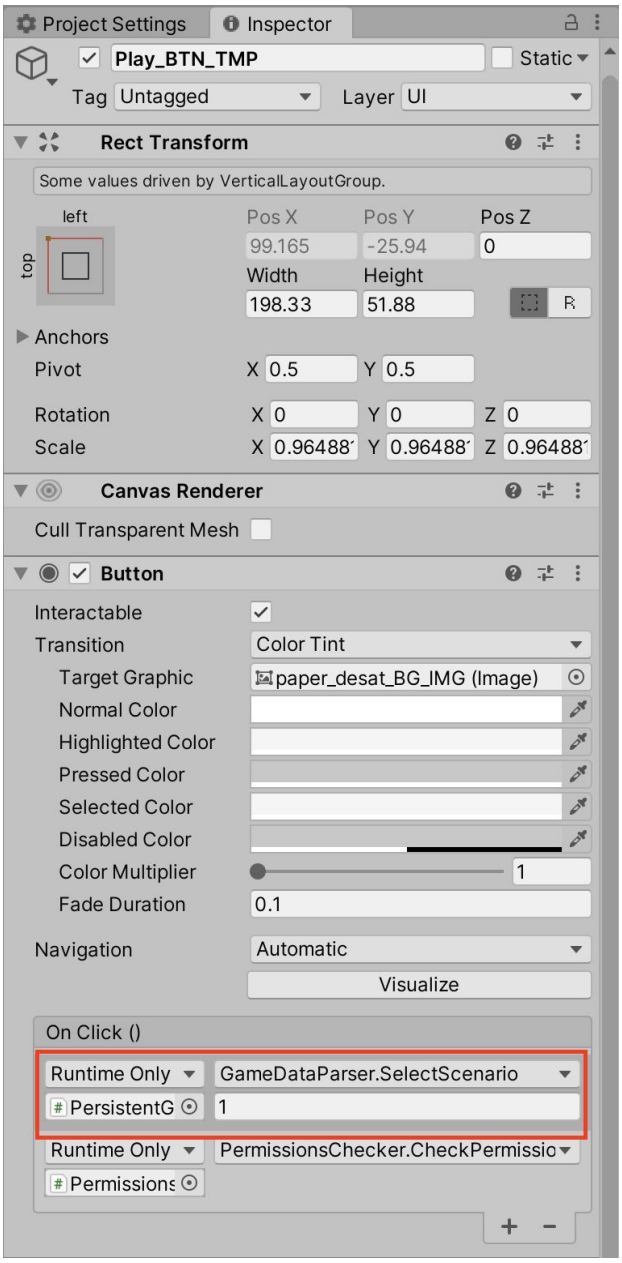
play_btn_txt_tmp(3): last text field, displays completion percentage.

Divider_1(1): second text divider graphic.

play_btn_txt_tmp(2): first text field, displays scenario name/number.

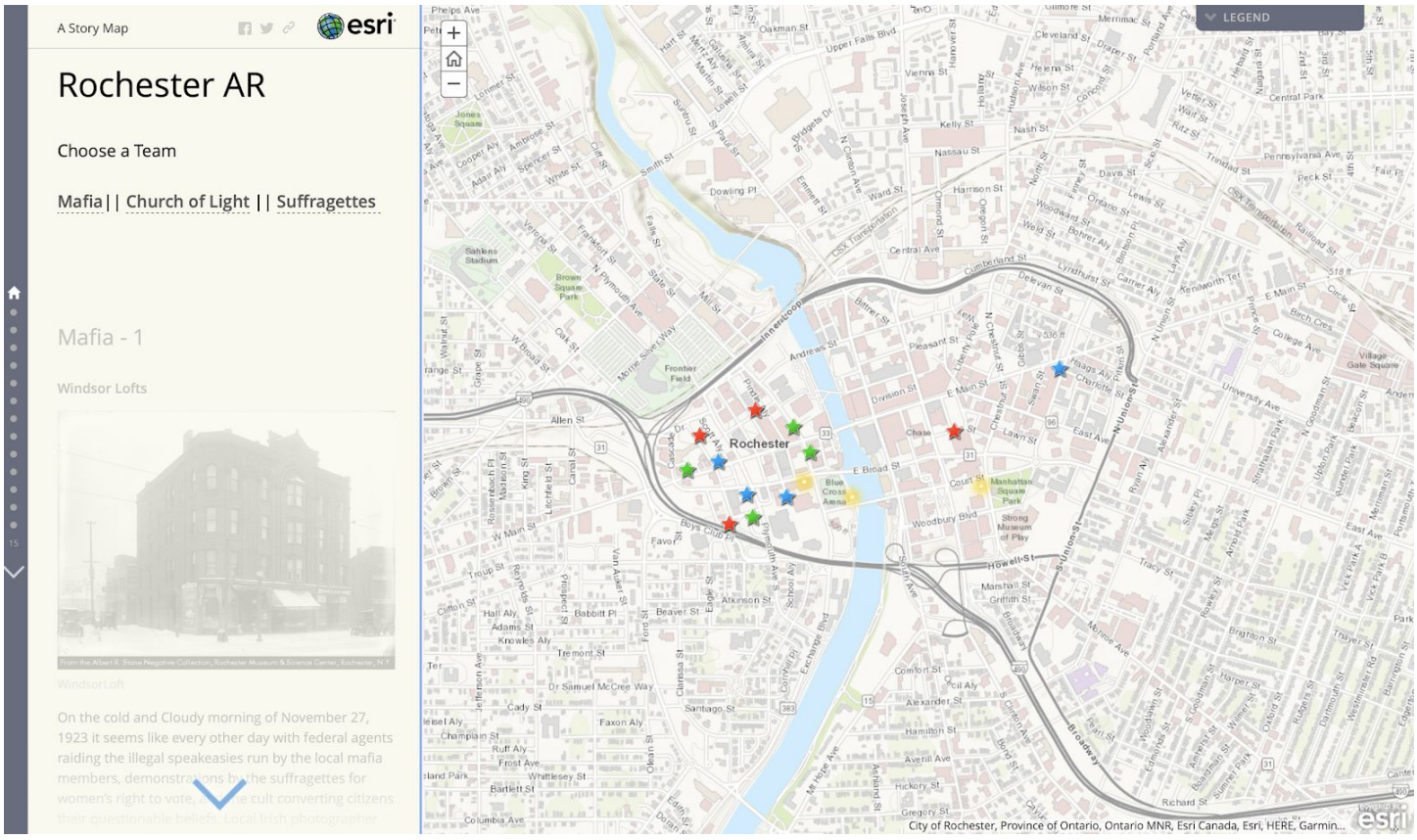
Each text and image field can be populated based on the theme of the new scenario. After updating the fields of the button, select the main button gameobject (Play_BTN_TMP), and go to the Inspector window. Find the OnClick() field of the Button component, and replace the value of the GameDataParser.ScenarioSelect field with the index of the new scenario.

After this step, a scenario index will be passed from the menu scene into the game scene, which will be used by SpawnOnMap, WaypointManager, and StoryManager to filter out game data not related to the new scene at runtime. Try creating a build or running the editor with Unity remote enabled.



IX - Notes on Creating Location-Based Experiences

Jessika Weber's thesis on making location-based games for tourism is an important tool to use when mapping the story sections of the narrative to real world locations. The thesis includes user-studies that cover what engages location-based game players, and what causes them to end the experience when playing a location-based app. Some elements, like the weather, are out of the control of the developer. The real world locations used for each narrative experience should be significant, interesting, and relatively close to one another.



This is a screenshot of the Esri prototype the Unity project is based on. The waypoints on this map are the same as the first scenario in the Unity app. Each waypoint is tied to a significant location, such as City Hall. Most of the waypoints are clustered in an area of downtown on the same side of the river. The distance between each waypoint is easily walkable, and the path to each waypoint is interesting on its own. Each user's walking threshold before they become disinterested in continuing the experience is different, but be sure to keep the distance between waypoints reasonable.

If certain waypoints have to be significantly farther from the cluster, which the prototype has, be sure to populate the path with secondary waypoints for the user to interact with on the way. Secondary waypoints can

also be used to guide users through the most interesting path between waypoints. An interesting path doesn't need to have a lot of app content, but should pass by scenic areas, historical buildings and landmarks, or areas of the town/city that have fun activities. In the instance of the user wanting to pause interaction with the app to experience the area around them, disengagement becomes a positive. Users are more likely to return to a location-based application that brings them to unique and interesting areas.

The implementation of custom datasets through Mapbox will allow developers to control the points of interest displayed on the in-game map. Elements like restaurants, stores, and graphics displaying land use (think green map sections representing parks) can be added.

X - Improving the Project as a Development Tool

Dynamic systems

When adding features to Rochester Steampunk AR the first version/iteration will tend to be hardcoded, meaning that it will work for a single set of input data. When thinking about implementing the feature to handle more than one scenario think of it as being a dynamic element or system in the project. The less interaction using the system requires from developers, the better. A good example of this is the way Rochester Steampunk AR handles spawning waypoint objects. Instead of making developers place X number of waypoint objects in the scene, a single waypoint prefab is used. The SpawnOnMap script has a reference to the prefab, which it uses at runtime to generate waypoint objects based on the number of locations in the current scenario. In Unity, making dynamic systems usually involves cutting down the number of editor-assigned inspector references and the amount of gameobjects that need to be placed in the scene.

One area that needs to be improved with the current state of the project is the method used to determine the number of steps in each path of a scenario. Developers add numbers to a collection within the StoryManager script, but this can't handle multiple scenarios in its current state. A fix for this could be a Scenario Data prefab that holds information about the scenario and is accessed by the StoryManager script.

Simplifying addition of content

Whenever a new type of content is added to this project, the act of adding and implementing that content should be made as simple as possible. One of the items listed for future development is adding 3D models for places of interest, which can include characters and buildings. The process for adding this content should be as simple as making prefabs and assigning them to a collection a script can run through to select the appropriate model to spawn as a waypoint on the map.

Current process can be simplified as well. When adding scenario selection buttons to the ScenarioSelect menu, the process could be simplified by replacing the act of copying and repurposing existing buttons with a prefab and inspector button system. The menu manager for the scene could be used to add buttons to the menu with the inspector, cutting down the amount of steps required.

Distributing workload

Another aspect to consider when developing Rochester Steampunk AR as a tool for student groups to use is the timeline and mix of backgrounds most groups at RIT have. Processes like handling the addition of narrative content should be able to be completed by students that want to focus on creative interactive

narratives, requiring as little interaction with Unity as possible. This allows more of their time to be spent creating quality content for the application than trying to figure out Unity UI elements and scripting practices. Students with programming backgrounds should be able to manage the Unity project, testing added content while adding and improving features in the app and tools in the Unity project.

XI - Build Chart and Future Features

	Original Project (Aris/Esri)	Unity Project/Build Version 1 - Spring 2020	Unity Project/Build Version 2 - Fall 2020	Unity Project/Build Version 3 - Spring 2021	Additional Features
App Features	- Narrative story segments Narrative images - Dialogue events/ NPC interactions - Decision paths within dialogue events - Maps + POI - Location tracking - Location-based interactions with narrative segments - Web player (accessible from most devices) - Random encounters Working Esri Demo	- Narrative story segments - Narrative images - Maps + POI - Location tracking - Location-based interactions with narrative segments - Familiar location-based game template - Custom map style Android build	- Dialogue events/ NPC interactions - Decision paths within dialogue events - 3D building extrusion from map - Improved Dialogue UI - Saved progress/game data - Error handling, messages	- 3D buildings & character models for POIs - Interactive content modules (videos, audio, games/puzzles) - Game audio	- Tour Mode (waypoints without narrative content at key locations in city) - Connection to online accounts (GooglePlay, social media, etc.) - AR camera implementation (NPC/Building modes in worldspace)
Development Tool Features	- Narrative catalog structure - Timeline/Process Manuals for Aris/Esri development Google drive folder	- Unity project - File-based game data (narrative + coordinates) - Mapbox implementation - Reusable Prefabs Unity project manual Google drive folder	- UI Themes (dynamic background images and fonts for UI elements) - Replace hardcoded areas I - Dialogue system - Different SpawnOnMap waypoint prefabs (narrative vs dialogue vs random encounter) - Updated Unity project manual	- Unity Inspector menus - Improve file handling - Replace hardcoded areas II - Mapbox Styles manual/walkthrough - Updated Unity project manual	