The Migrants' Chronicles: 1892 (README)

(16.10.2024, Contact: infothemigrantschronicles@gmail.com)

Welcome to **The Migrants' Chronicles: 1892**, an open-data educational game that immerses players in the migration journey of Luxembourgish migrants to the United States in 1892. This game is part of a series of serious games that combine historical accuracy with interactive storytelling to promote empathy and understanding of migrant experiences.

Tools Used in Development

- **Unity**: The primary engine used for creating and managing game scenes, characters, dialogues, levels, and handling the build process across platforms.
- **Articy**: A tool used for creating and exporting dialogues, along with handling variables and conditions within those dialogues.
- **Microsoft Excel**: Facilitates the localization of dialogues. Articy exports dialogue data to Excel files, where localized dialogues can be edited and managed.
- **Xcode (for iOS Builds)**: Used to generate iOS builds after Unity creates the Xcode project folder.
- **Git or Other Version Control Tools**: Essential for managing project changes. Certain files, such as generated dialogue files, are excluded from version control.
- **Graphics/Animation Tools (e.g., Photoshop, Illustrator)**: Used to create character sprites and UI elements, which are imported into Unity as textures or animations.

Table of Contents

General Structure

The player travels through different cities to get to New York.
Each city is its own Scene in Unity (subsequently called “levels”).

NewGameManager

The game manager is persistent between all levels. It gets instantiated as soon as the player starts in the first level after the main menu. It is a singleton which you can access globally with *NewGameManager.Instance*. It holds all the data that is persistent between levels, e.g. player inventory, health status, money etc.
The NewGameManager script contains all the data. In *Assets* > *Prefabs*, there is also a “GameManager” prefab, which holds the script and also persistent data that has been outsourced to its own components. These components still reside on the same prefab, and you can access them through the game manager.

HealthStatus

Contains the health status of all the playable characters and stores whether they are sick, exposed, hungry, seasick or homesick.
There are the classes “HealthState_[Hungry|Cholera|...]”, which represent the data for this sickness for one character.
The *ProtagonistHealthData* combines all sicknesses for one character.
The *HealthStatus* creates a protagonist health data for every character of the protagonists.

DebugConsole

In development builds, logs or errors will be displayed on the screen.

ItemManager

Has a list of all items that exist, which you can then access. Does not store what items the player has (-> PlayerInventory).

LocationManager

Every level / location has its own string ID, with which it is identified throughout the application. This ID should always be the same as the name of the Scene in Unity (in *Assets* > *Scene*). This ID can differ from the name that is shown in the UI by spelling (e.g. ID: “LeHavre” -> “Le Havre”) or by language (if the cities are called differently in Luxembourgish).
The LocationManager provides a list with all locations in the game, and stores for each location the ID (called “Display Name” here), the Technical Name (the ID in Articy), whether it is in Europe or America, and the localized name (that is shown to the user).

ShipManager

Handles whether the user is currently on the ship / visits a city during a stopover.

RouteManager

Routes are a route from one city to another with a specific transportation method. There can be multiple routes from one city to another city with different transportation methods. Not every connection from one city to another has all transportation methods available. Some routes are available as soon as you travel to the city, others are discovered during dialogs and should be hidden in the beginning.
The RouteManager stores all the routes that are currently available (routes that are available from the start and routes that have been discovered in dialogs so far).

DialogConditionProvider

Provides a list with all conditions that are currently set and links them to Articy. Conditions are explained more in detail in a later section.

QuestManager

There are main and side quests.
Main quests are given from the start and have to be handled somewhere in code (when they fail / succeed).
Side quests are given during dialogs and also succeed during dialogs. They can fail if you travel to a specific city or if you leave a specific city.
The QuestManager holds a list of all quests that have been started / failed / succeeded.

FamilyMainQuest

This should probably be refactored as soon as the other protagonists are implemented, but this controls the main quest of the family (get to New York before some date).

DiaryEntryManager

Diary entries are automatically generated. Diary entries are shown / created if you travel to a city and if you sleep there.
Auto-generated entries contain information about the current city, the health state, game information like where you slept, if items were stolen etc. Information about the diary entry and localized descriptions are stored as variables in this component.
Diary entries are regenerated when the language changes and further explained in a later section.

TransportationManager

Provides access to localized transportation methods to display to the user what transportation method he choose.

SaveGameManager

Triggers the save game functionality and stores the save game to a file.

GeoJSONManager

Provides information to generate the geojson file along with the PDF. Has the coordinates of every city and generates the route the player took.

CharacterManager

Has a list with all characters that exist in the game, which you can access and get the character by name / id.

LevelInstance

Each level is built the same way: Each level can have one or more scenes, you can talk to people and there are market stalls or shops to buy items.
The LevelInstance was built to facilitate the usage and encapsulates the shared functionality. You can easily add new scenes, dialog buttons, shops etc and the LevelInstance takes care of hiding / showing everything.
It is not persistent between levels, so each level has its own LevelInstance.
A scene in this project is a specific background with characters that you can interact with. A level can have multiple scenes and you can switch between scenes. Each scene has a background (for the sky / buildings in the background), middleground (characters, markets stalls, etc) and foreground (for objects that should be on top of characters, e.g. lantern poles, bushes) and is static (you cannot go further left / right or move in the scene). Each scene should have a Playable Character Spawn in the middleground. This is where the protagonist will be spawned at startup, depending on who you chose.
You can access the LevelInstance globally with *LevelInstance.Instance*. You can find further information in the class comment in *LevelInstance.cs*.

Prefab

The prefab "LevelInstance" holds the script and all the functionality that is shared across all levels.
The *Scenes* child object is the parent for all the scenes that you add to the level.
The *Global Volume* is used for blurring the background if you open dialogs, shops or the diary.
The *ForegroundScene* is used for the dialogs and further explained in a subsequent section.
All backgrounds (buildings etc) as well as the characters are sprites that are used / placed in the world (with SpriteRenderer). There are also UI elements that cannot be in the world (e.g. the status info, diary, the clock, diary buttons etc). For that, they have to be placed within the *Canvas*.
The *GameManager* is a reference to the previously mentioned game manager prefab, which you should change in the prefab itself.
The *AudioManager* contains the audio manager to play music and sound effects.
There are also different cameras here: The *Main Camera* for the sprites in the world (buildings, characters, etc), the *Foreground Camera* for the foreground scene (since this has to be rendered on top of the blur) and the *UI Camera* for the UI.
The *PDFCamera* and *PDFCanvas* are used to generate the images for the PDF (the journey image as well as the individual diary entries that are shown in the PDF).

Canvas

The canvas holds all UI elements in the scene.
The *SceneInteractables* is the parent for all scene interactables (dialog buttons, shop buttons, scene buttons etc). Each scene needs to have UI elements (buttons) to interact with the characters. Each scene has its own "SceneInteractables" prefab created under this parent and assigned to the scene (which takes care of showing / hiding it).
The *DialogSystem* holds the dialogs (the speech bubbles that are created for every dialog line).

The *Overlays* object is mostly used for shops. So each shop should be in the overlays.
The *Interface* is its own prefab and holds all the user interface elements (status info, clock button, diary etc).
The *BackButton* is used consistently everywhere where you should be able to go back or close the current elements (popups, diary, shops, dialogs etc).

ForegroundScene

The foreground scene is used in dialogs for the characters that should appear left and right of the dialog. They are displayed on top of the blur and rendered separately.
The *ForegroundScene* script takes care of instantiating the correct character when someone is talking. The correct size and position is calculated in this script and they are placed in the *Left* or *Right* child object accordingly.

Dialogs

Dialogs are created in Articy:draft and imported via the articy:draft Unity importer.

Export dialogs from Articy

1. In Articy, click on Export and select unity from the left technical exports.
2. Make sure *Enable localization support and create MS Excel file(s)* is checked.
3. Select *Plain text* as the *Markup for text formatting*. Otherwise it may result in strange formatting within Unity.
4. Select the export folder: Replace the *Dialogs.articyu3d* file in the Unity project, located in *Assets* > *Dialogs* > *Dialogs.articyu3d*.

Import dialogs in Unity

1. Whenever you focus Unity again or start Unity the next time, the dialogs are regenerated from the new Dialogs.articyu3d file. The importer generates all the dialogs and data from this file. These generated files should not be committed and can be regenerated from everyone else. After the import, dialogs can be selected to be played and templates, enums etc. created in Articy can be used in Unity.
2. If something does not import correctly, reimport the whole file in Unity: *Tools* > *articy:draft Importer* > *Advanced* > *Force full import of articy:draft export file*.
3. All other team members who work on the project have to reimport the whole file using the step above, after they have pulled the changed *Dialogs.articyu3d* file, so notify them if you update this file.

Using dialogs in Unity

1. Drag a new *DialogButton* prefab into the scene interactables of the scene.
2. Add the character, with whom you talk, to the *HideObjects* of the *DialogButton* script. This hides the character when you talk to him.
3. Fill in the values of the *PositionOnSprite* component to show the dialog button on top of your character (usually: set *Mode* to *SpriteContainer*, *SpriteContainer* to the character, and *NormalizedPosition* to (0.5, 1.1)).
4. Place a *Dialog* prefab as a child object of the dialog button.
5. In the *ArticyReference* component, select the dialog you want to play. Make sure that a dialog is selected (the icon are two speech bubbles, one yellow filled, one yellow transparent), and not a character, dialog fragment or something else.
6. You can play dialogs only under specific conditions with *Condition* of the *Dialog* component. If no condition is set, it evaluates to true and plays the dialog. You can add one or more conditions and chain them together using and *And* or *Or*. The dialog system iterates over all dialog child objects of the dialog button and plays the first one whose condition evaluates to true. You can use this to play different dialogs for different characters. Each playable protagonist will have different dialogs. You can have each protagonist set a condition when they're chosen (using *SetConditions* of the *PlayableCharacterData*) and then set this condition as the *Condition* of the dialog, so that the dialog only plays for this specific character.

7. By default, dialogs play until the end and don't repeat. So if you play through a dialog, the position is stored and you can return to this position. If the dialog is finished, you only see the history of this chat. Some dialogs should be restartable (e.g. Ticket Sellers). Typically they will be restarted from within dialogs (so from Articy). If a dialog should be restartable, there should be a condition / variable defined in Articy within the *DialogueEnd* variable set (should be unique for each character). Then you can set this condition in the *Dialog* component in *RestartCondition* (so e.g. *DialogueEnd.[NameOfTheCharacter]*). If this condition is set to true (usually from within dialogs), the dialog is restarted as soon as you reopen the dialog via the dialog button.
8. By default, all protagonists are displayed on the right (e.g. for the family: the mother and the children). For some dialogs (e.g. the introductory dialogs) you may want that only the main protagonist is displayed on the right (e.g. the mother), while the others (e.g. the children) are displayed on the left. You can change this in the *Dialog* component in *ProtagonistMode*. Use *Any* (the default) so that all protagonists are displayed on the right, and *OnlyMain,* if only the main protagonist should be displayed on the right.

Dialog localization

For further information, follow
https://www.articy.com/articy-importer/unity/html/howto_localization.htm
Localization of dialogs is handled differently than localization of elements within Unity itself. When you import the *Dialogs.articyu3d*, the *loc_Alle Objekte_en.xlsx* file is regenerated. It contains all the text that is used in the dialogs (dialog fragments) in English. You can add more languages by copying the file and replacing *en* with the language code of the new language. For luxembourgish, there is a file *loc_Alle Objekte_lb.xlsx*, where all the text is translated. All columns need to stay the same, except the *Value* column has to be translated. The importer does not provide a way to review the changes made to this file, so you have to find another way to reflect those changes in the other languages. The other languages (except English) are not updated or regenerated. Whenever you import a new dialog file, you have to apply the changes made to the english version to the other languages as well. What happens if the other languages don't provide a row used in the dialogs is untested.

Dialog entities

Articy lets you create objects like characters, items or locations, which can then easily be used in the dialogs. These objects usually contain a *Technical Name*, which is the ID of this entity and can be retrieved in Unity.
Since all data for these characters, items and location is stored in Unity, the entities exist in Articy solely for the purpose of using them in the dialogs. No data is stored in Articy for these objects. In Unity, the *Technical Name* of the objects used in dialogs can be retrieved and mapped to the data created in Unity, that contains all the information.

Dialog templates

Dialogs can trigger behaviour in Unity. For that, each dialog fragment can use a predefined template. A template in Articy is a list of features, and a feature is a list of properties, which can be strings, integers, booleans or enums.
If a dialog fragment contains a template, 2 functions in *DialogChat.cs* are relevant: *WantsToHandleTemplate*, where you can specify if the template is handled at all, and *HandleTemplate,* which is called to actually do something. For example, a route can be discovered, an item can be added, a quest started or money removed. This is handled as soon as the articy player pauses on this dialog fragment. Within the *HandleTemplate*, you can access the features and properties of the template and act accordingly.

Conditions

Conditions are variables that can be used within dialogs or in Unity to change the behavior of something based on a variable. There are conditions in Articy and Unity, and are partly linked.

Conditions in Articy

Conditions in Articy are called “Variables” and organized in different variable sets (e.g. “Health” for health related variables, “DialogueOptions” for variables used in dialogs etc). Variables can be strings, integers or booleans. They are always set and have a name that consists of the variable set and the name itself, so “[VariableSet].[Name]” (e.g. “DialogueOptions.AnnikLetter”) and a value.
Variables in Articy persist throughout the game and don’t get reset if you travel to a new city.

Conditions in Unity

We have our own conditions in *DialogConditionProvider.cs*. Refer to the class comment for further information.
Conditions in Unity are similar to the boolean variables in Articy. Other than in Articy, they are not always present, but are either set (added to the list of active conditions) or not, which corresponds to a bool (is in the list or not). Conditions are only a string, which you can choose however you want.
Conditions can be local and global. Local conditions only have validity on the level they were added, and are reset (removed from the list) if you travel to a new city. Global conditions exist throughout the game and continue to exist if you travel to a new city. If you set conditions, you can set them locally, unless you want them to persist.
The *DialogConditionProvider* provides access to special conditions that are not boolean conditions. For example, you can access how many days a character has been sick or hungry and compare that to an integer. Refer to the documentation for more details.

Combining both systems

Boolean variables and conditions in Unity are synced. If you add a condition, it checks if this condition exists in Articy. If yes, it is added to Articy. If not, it is added to the Unity conditions directly. DialogConditionProvider also listens to changes of the Articy variables, and adds /

changes new conditions to the list in Unity. Integer or string variables are not synced, since they cannot be stored in the existing system.
If the condition exists in Articy it does not matter if you want to add the condition locally, it is added to Articy anyway.
There exist conditions that are only defined and used in Articy, conditions that are defined in Articy but also used / set in Unity, and conditions that only exist in Unity and are being used only within Unity.
To interact with conditions by code, use the methods of the *DialogConditionProvider*, which handles everything in *AddCondition(), HasCondition()* and *RemoveCondition()*. Normally, you don’t have to interact with it by code, since everything that uses conditions exposes either a *SetConditions* to set new conditions (locally or globally) or *Conditions*, which can be used to conditionally make something visible / start a dialog etc.

Localization

Localization for strings in Unity (everything except the dialogs) are handled with the localization system in Unity.
You can access the localization tables in *Window > Asset Management > Localization Tables*. From there, you can edit every translation in the game and add new translations.
Localizable strings are organized in tables. We have a *DiaryTable*, which contains all the strings for the diary entry generation, *ItemsTable* for all the strings of every item, and *GeneralTable* for everything else. You can export each table on the 3 dots on the top right to csv and import them from csv.
Normally you don't need to interact with the tables directly, since you can use *LocalizedString* to reference one translated string. If you expose this as a variable, you can set the *TableCollection* to the table you want to use, and then click on *Add Table Entry* to add a new string. For *Entry Name* enter a unique identifier for this string. Then you can enter the *English(en)* and / or the *Luxembourgish (lb)* translation. To get the localized string by code, call *LocalizationManager.Instance.GetLocalizedString(localizedString)*.
As soon as you change or add new strings, you have to regenerate the addressables data with: *Window > Asset Management > Addressables > Groups* and then click on *Build > New Build > Default Build Script*. Otherwise your changes won't be reflected in the game.

Diary

The diary is a prefab used for both the ingame diary and the main menu (in *Assets* > *Prefabs* > *Diary*, the *Diary* prefab).
It has functionality and animations for switching pages to make it easy to add new pages.
The diary game object has multiple child objects, most of them having “Controller” in their name. These controller objects are modified in animations, so if you want to change sizes, anchors etc modify the child object immediately succeeding the controller parent object.
The *BackgroundController* object has the background image. This is animated and depending on the state of the animation, can be the closed book or opened book.
The *AfterBackground* is for objects that should render on top of the background, but below the content. In the ingame diary, it is used for rendering the quest log, in the main menu for the buttons that appear on the closed diary (start new game, continue game etc.).
The *MarkerController* contains the markers that appear on top of the diary for selecting the map page, diary page etc. These are currently hidden in the main menu.
The *PagesController* controls the animation of the background page when flipping pages. The content of the page is only the controls / images that the page consists of, without the actual background page (so with transparent background). The *PagesController* has the background page image, as well as controls the animation when flipping pages, so you don’t need that for every page that you add.
The *ContentController* has all the pages that you add to the diary. Each page that you add is a *DiaryContentPage* prefab, where you have a left and right page to add content to. You can also decide that it’s a double page (e.g. for the map, that spans over both pages). These *DiaryContentPage* are organized as children of *DiaryContentPages* objects. In the ingame diary, you have different *DiaryContentPages* for the map, diary entries, settings etc. Each *DiaryContentPages* can have multiple *DiaryContentPage* objects, which you can iterate over. For example, if you add diary entries by traveling and sleeping, you add new *DiaryContentPage* objects to the diary entries’ *DiaryContentPages* object.
The *AfterContent* has content that should be displayed on top of the content. This is mainly used for buttons with which you can flip pages (e.g. if you have multiple diary entries).
The *Binder* is used to render a middle line on top of double pages (the map).

The *IngameDiary* is used in game and has the map, diary entries, settings, inventory and the health status.
The *MainMenuDiary* is used in the main menu and has controls to start a new game and review the settings.
Both use the *Diary* prefab and therefore use the same mechanism and animations for flipping pages.

Diary Entries

Each diary entry consists of exactly 2 pages, on the left and on the right of the diary. For each page, you can select a prefab. The prefab determines the layout of the page, e.g. if there is only text, if there is an image at the top etc. Each prefab has a script attached that distributes the text and image data that you specify accordingly. Which text / image is used is specific to the prefab.

Each page can also have drawings in the background. Those images render behind the actual text data.
DiaryEntry and *DiaryPageData* define the visual elements of the diary entry, so which text should be displayed.
Diary entries are automatically generated by *DialogEntryManager* and contain information about where you are, how you slept, what your health is etc.
DiaryEntryInfo contains all the information required to generate a diary entry, so this is data that is collected from the game manager and is everything that you need to generate the same diary entry.
DiaryEntryData links both together and has the *DiaryEntryInfo* as well as references to the visual pages. This exists because the diary entries can be regenerated, i.e. if the user changes languages, the entries are regenerated using the new language (with the same *DiaryEntryInfo*) and update the texts in the UI.

Characters

Characters are divided into 2 parts.
In *Assets* > *Prefabs* > *Characters* are the base archetypes for the characters. These files will be modified / added by the artist / animator and are not used directly in the scenes.
In *Assets* > *Characters* are the characters that are really used in the scene. They have the prefabs of *Assets* > *Prefabs* > *Characters* as the child object of an empty game object. This is so that the animator can modify the characters without interfering with the developers.
In *Assets* > *Animations* > *Characters* are the animations for the characters. There are 2 animators used by the characters, the *ProtagonistAnimController* and *NPCAnimController*.
The *ProtagonistAnimController* handles the animations for the protagonists, which can have different animations based on the health state, additionally to the idle animation, as well as talk.
The *NPCAnimController* controls the animations for all NPCs, which only have an idle animation and can talk.
To create the animator used for the characters, create a *Animation Override Controller* and select the animator that you want to use (protagonist or NPC). Then enter your animations. These exist for the archetype of the character, and can be reused when you create variations of these archetypes.
In the actual prefab (in *Assets* > *Characters*), make sure to select the correct animator (override controller). Then add either the *ProtagonistAnimationController* or *NPCAnimationController* component to the parent game object. Here you can change the speed of the animations, if required.
If this is a character (prefab), that you can talk to (so it should appear in the foreground scene in dialogs), add the *CharacterDialogInfo* component to the parent game object. Here you should enter the Technical Name (of Articy), and whether the character looks to the left in the editor. This is to flip the character in dialogs if required. For children you can enter a scale factor to scale them down in dialogs and then an offset to place them at the bottom of the dialogs. This prefab should then be added to the *Character Manager* of the *GameManger* prefab, so it can be found and displayed during dialogs. Each character only should appear once in this list, so you currently cannot have variations of characters during dialogs.

Protagonists

At the start, you can choose between 3 protagonists, a family, a couple and a single man. Each protagonist can have multiple characters (e.g. the family: the mother and 2 children). Mostly they are also called protagonists in variable names etc.
Each of the 3 protagonists have their own *PlayableCharacterData* in *Assets* > *Characters*. This file describes how many protagonists (characters) there are and their names etc.
You can set conditions that should be added when this protagonist is chosen (that way you can play different dialogs).
The *Scene Prefabs* allows you to have variations of these protagonists in the scene. You can add a prefab (that has the protagonists in different constellations) with a name. In the *PlayableCharacterSpawn* of each scene you can specify a name that corresponds to the prefab here, which will then be instantiated.

The *ProtagonistData* should have an entry for every protagonist (e.g. the family: 3, for the mother and 2 children). For each protagonist you can then specify the display name (in *Full Name*), the ID (*Name*) and the Technical Name (of Articy). There should always be exactly 1 main protagonist (e.g. the mother).
You can also set the initial money and the main quest of this protagonist.

Transportation

Ship

The ship works differently than all normal cities. Instead of having multiples scenes, the ship only has one scene, which has multiple rooms. There is a prefab *Room* used for that. Each room is built the same way as a scene, so it has a background, middleground and foreground, and it has a *PlayableCharacterSpawn* by default.
You can select which classes are allowed to enter this room, and the room name.
At startup, a *RoomButton* prefab is instantiated for every accessible room that can be used to enter the room.

Transportation to levels

If you travel from one city to another, the normal flow is:

1. The *GoToLocation* function of the game manager is called.
2. It checks whether you can travel again today.
3. If side quests fail because you leave this city, popups are created and the flow only continues after you close the popups.
4. Additional checks if you can travel, and reducing the money.
5. Set the *nextLocation* and *nextMethod* property. This is used to later determine where you actually want to go.
6. Load the *LoadingScene* scene.
7. The *LoadingManager* component within the *LoadingScene* handles loading the next scene by looking at the *nextLocation* property.
8. It loads the scene asynchronously, and then waits until at least 2 seconds have passed.
9. After at least 2 seconds passed, it activates the new scene.
10. *OnBeforeSceneActivation* of the game manager is called, which creates a new journey entry and opens the new diary entry.

If you travel with the ship to America, the flow looks differently. You can start from a port in Europe, have a stopover somewhere in Europe, then pass time on the ship, arrive in Elis Island, and finally you arrive in New York City.

1. *GoToLocation* is called with "NewYorkCity" as the target and "Ship" as the method.
2. It checks whether you can travel again today.
3. If side quests fail because you leave this city, popups are created and the flow only continues after you close the popups.
4. The ship manager is signaled so it stores that you're now travelling on the ship.
5. *nextLocation* is set to "NewYorkCity" and *nextMethod* to "Ship".
6. The loading scene is loaded.
7. The *LoadingManager* loads the Ship level.
8. After the ship is loaded, *OnLoadedShip* is called, which creates a new journey entry and opens the new diary entry.
9. If you decide to visit the stopover the next day, the loading scene is loaded again.
    a. The *LoadingManager* loads the stopover location.

    b. *OnLoadedStopover* is called, which creates a new journey entry for the stopover.
    c. If you want to return to the ship, you can either press the ship button (that allows you to spend the rest of the day on the ship) or press the clock button (which ends the day on the ship).
    d. In both cases, the loading scene is loaded again.
    e. The *LoadingManager* loads the ship again.
    f. If you clicked the clock button, the end day popup is opened.
    g. If you clicked the ship button, nothing else happens.
10. If you arrive on Elis Island, the loading scene is loaded again.
11. The *LoadingManager* loads Elis Island.
12. *OnLoadedElisIsland* is called, which creates a new journey entry and resets the *nextLocation* property. The ship manager is notified to store that the player is not on the ship anymore. This is now treated like any other city.

General How-To's

Create a new level (city)

1. Create a new Basic (URP) scene in Unity.
2. Delete the *Main Camera* game object.
3. Instantiate the *LevelInstance* prefab.
4. Create a new scene for the level and assign the *DefaultScene* property.

Create new scenes

1. Instantiate a *Scene* prefab as a child object of the *Scenes* objects in the *LevelInstance*.
2. Give it a name in *SceneName* of the *Scene* component.
3. Place a *PlayableCharacterSpawn* in the *Middleground* of the scene.
4. If required, set a name of the prefab to spawn in the *PlayableCharacterSpawn*.
5. Assign the *PlayableCharacterSpawn* to the *CharacterSpawn* property of the *Scene* component.
6. Instantiate a *SceneInteractables* prefab in the *SceneInteractables* parent in *LevelInstance > Canvas > SceneInteractables*.
7. Assign the interactables to the *Interactables* property of the *Scene* component of the scene.

Position interactables on top of sprites

Characters and buildings are sprites in the world. Dialog buttons etc are UI elements. Since this game can be played on devices with different aspect ratios, the location of the UI elements won't match the location of the sprites. For this you can use the *PositionOnSprite* component, which is already added to most buttons and places the UI element on top of the selected sprite at startup.
You can switch between 4 different modes. Most of them use the *NormalizedPosition* property. (0/0) means at the bottom left, (1/1) at the top right. You can also specify values below 0 or greater than 1.
Sprite allows you to set a sprite and position the UI element at the specified position on top of a single sprite.
SpriteContainer is used if you want to place elements on top of characters. Characters consist of many individual sprites, so this mode calculates a bounding box of all child sprites of the character and places the UI element accordingly.
PlayableCharacter can position the element on top of the playable character that is spawned at startup, because this game object cannot be selected at editor time. Other than using the dynamically spawned game object, it is equivalent to *SpriteContainer*.
WorldObject allows you to select a WorldObject and place the UI element with offsets to the location of this object. *NormalizedPosition* is ignored. This mode is used by the room buttons on the ship.

Build for Android

1. Open the Build Settings in *File > Build Settings.*
2. If not already selected, select *Android* as the platform and click on “Switch Platform”.
3. If you want to export it directly to a device, you can select the “Run Device” here.
4. Enable the “Development Build” option if you want to test something (a console with error messages appears ingame), else disable it.
5. For “Compression Method”, use “LZ4” for development builds and “LZ4HC” for release builds.
6. You can “Build” (only build) or “Build and Run” (build and export the apk to the device). Both options export the .apk file to the location you specify.

Build for iOS

For building refer to: https://docs.unity3d.com/Manual/iphone-BuildProcess.html
After you have finished creating the build, you need to open the generated Xcode project to export the application to a device. Refer to:
https://johannadaher.com/2017/08/23/unity-so-exportiert-man-ios-apps/

Import textures

All imported textures are used as sprites or in the interface. In the Import Settings, make sure the following is selected:

1. Set “Texture Type” to “Sprite (2D and UI)”.
2. Set “Sprite Mode” to “Single”.
3. Enable “Alpha is Transparency”.
4. Disable “Generate Mip Maps”.

Bug Reports & Feedback

If you encounter any issues or have suggestions, please fill in the TMC bug report survey: https://forms.gle/1Tz7swmktuw7ffv49

License

Terms and Conditions:

By contributing to or using this project, you agree to abide by the terms outlined in the license. You can view the full terms and conditions on our GitHub repository.

Contact

For any questions, feedback, or contributions regarding **The Migrants' Chronicles: 1892**, feel free to reach out to us via the following channels:

- **GitHub Issues**: Submit issues or feature requests here.
- **Email**: You can contact the development team at infothemigrantschronicles@gmail.com

We look forward to your input and collaboration!