Programmer's Manual

for

TwinVisitor (Godot Engine Game)

Version 1.0 approved

Prepared by:
Vanja Venezia, Lucas Wilkerson, Daniel Arias, Jianna Angeles

December 5th, 2022

TABLE OF CONTENTS

1. Introduction

1.1 Purpose

The purpose of this document is to explain the program code and programming practices used in implementing TwinVisitor, with the goal of aiding developers both in debugging potential issues and in adding content or features to the game.

1.2 Scope

The purpose of this game is to create a narrative experience for a player evoking the aesthetics of low-poly games for early 3D-capable game consoles as well as the cyberpunk genre of fiction. The game exists primarily as a framework for creating such a narrative within, extending existing engine features of Godot as well as implementing unique features. Players can save and load their game, move the player, move the camera, switch playable characters, interact with objects and non-player characters, transition scenes, and manage an inventory of items gained through non-playable character interactions to progress the story.

1.3 Overview

The remainder of this document is sectioned into three succeeding components:

2. Getting Started – general setup of project and description of components
3. Coding Standards – defined by context
4. Implementation – overview of data flow and functional components

2. Getting Started

2.1 Prerequisites

TwinVisitor is built on top of the Godot Engine (godotengine.org), specifically v3.5.stable.mono.official [991bb6ac7], available from the Download page on the Godot Engine webpage. A present version (currently v3.5.1) that is fully compatible with the game project can be obtained from here. The TwinVisitor project is compatible with both Windows and Linux implementations of Godot. Note that C# was used as the scripting language for the project, thus the release of Godot with Mono support is necessary to properly interface with the scripts.

2.2 Importing the Repository

The source code for the project is available on the MushroomPeople Github page (https://github.com/MushroomPeople/TwinVisitor/). In the repository, there are two folders: Docs and Twin Visitor. Docs contains the Software Requirements Specification and Software Design Specification documents, including the software test plan and detailed information on game classes. Twin Visitor is the folder that holds the project files for the game, as well as a current Windows build.

To open the TwinVisitor project, the files must first be imported into the engine. Upon loading Godot, the Project Manager window appears. From the Project Manager, select "Import" and navigate to the "project.godot" file in the Twin Visitor folder. Once this is done, the project will be listed in the Project Manager and available to view and edit.

2.3 Scripts

- CopNPC.cs
  - Function: drives dialogue prompt and item instantiation relating to dialogue.
- DataUtils.cs
  - GameData
    - Function: serializable class designed to hold necessary data for saving and loading player progress. Notably:
      - playerATransform, playerBTransform
        - Global locations of player characters.
      - playerAActive, playerBActive
        - Which player character is active.
      - playerInventory
        - List of items in player inventory.
      - currentScene, currentSceneName
        - Relative path to game scene and its name.
      - playtime
        - String containing active playtime of game in hours, minutes, and seconds.
  - Save
    - Function: holds WriteData function, used to serialize a GameData object into JSON and write the JSON to a specified file.
  - Load
    - Function: holds GetData function, used to read JSON data from a specified file and deserialize it into a GameData object.
- DefaultItem.cs
  - Function: generic implementation of an item. On player interaction, a DialogBox is displayed containing the message “You got {item name}!” and the name of the object is added to the player’s inventory dictionary.

- DialogBox.cs
    - Function: generic implementation of a dialogue popup, requires a RichTextLabel child node. On player interaction, a popup box with preloaded text is displayed. On repeated interaction, if there is still text to display, the text is advanced. The popup box is closed when there is no more text to display. The text is held in an array of strings labeled “dialogue”, exported to the editor.
- GameControl.cs
    - Function: keeps track of player inventory and playtime, handles switching between playable characters, and handles adding items to and clearing inventory.
- InteractableNPC.cs
    - Function: generic implementation of a non-playable character that the player can interact with. Holds a DialogBox by default.
- InventoryButton.cs
    - Function: connects UI element on inventory screen to player inventory, facilitating player ability to equip items through the InventoryMenu.
- InventoryMenu.cs
    - Function: UI screen that pauses gameplay and displays all items available to player, allowing player to equip items via InventoryButtons.
- LoadApartment.cs
    - Function: alternative implementation of LoadScene that checks whether the player has a specific item equipped (“Keycard”), and only transitions to the specified scene if the player has the correct item equipped.
- LoadFromMainMenu.cs
    - Function: alternative implementation of LoadMenu that functions when accessed from the MainMenu, as LoadMenu expects to be transitioned to from the PauseMenu. Handles loading saved game progress and clearing user data. Loaded game data is injected into a new game instance.

- ➔ LoadMenu.cs
  - ◆ Function: UI screen that handles loading saved game progress and clearing user data. Loaded game data is injected into a new game instance. Includes a helper function for formatting playtime to store in a GameData object. Automatically populates save file display with information about the save data on instantiation, and updates the displayed information upon any action in the menu.
- ➔ LoadScene.cs
  - ◆ Function: handles scene transitions. On player interaction, the new scene is instantiated and the previous scene is freed from memory.
- ➔ MainMenu.cs
  - ◆ Function: UI screen that handles new game instantiation and transition to LoadFromMainMenu.
- ➔ PauseMenu.cs
  - ◆ Function: UI screen that stops the physics process of the game and allows the player to resume playing the game, quit playing and exit the game, or transition to the LoadMenu or SaveMenu.
- ➔ PlayerA.cs
  - ◆ Function: player controller, handles input for movement, interaction, and camera. Holds active status, if inactive, turns the inactive player character toward the active player character and follows movement at a set distance.
- ➔ PlayerB.cs
  - ◆ Function: copy of PlayerA with inverted default active status.
- ➔ SaveMenu.cs
  - ◆ Function: UI screen that handles saving of player data. On player selection of a save file, the current game state is used to instantiate a GameData object which can be passed to DataTools for writing to disk.
- ➔ SceneInfo.cs
  - ◆ Function: stores relative path to scene, scene names, and player spawn locations. Handles moving players to specified spawn points upon scene instantiation.

2.4 Design Rationale

Most of the organization of the game project is resultant from usage of the Godot Engine. The design was created with the intent of following structural recommendations from the Godot documentation. The node system Godot employs necessitates hierarchical organization of game components. Differences from good practices as demonstrated in the Godot documentation are generally due to time constraints in the creation of the software and the scope of the project. Deviations from official recommendations of the Godot Engine include the following: the UI screens are not encapsulated within a universal UI element, and instead exist as children of GameControl, like all in-game objects - there are few UI elements, so they do not clutter the game tree and remain clear in the editor; the inventory screen UI is hard-coded to reference specific objects, rather than abstracted and flexible - the game only uses one item in its current state, so this does not hinder functionality; the in-game environment scenes do not have any node organization, this is acceptable due to the few number of game scenes and scope of the project.

3. Coding Standards

3.1 C# / Mono

Where possible, adhere to the official Microsoft coding standards.

3.2 Godot

Imports are not named in any specific way. Some C# methods in scripts are named with GDScript formatting when automatically generated by the editor and have been left as such to ensure parity.

4. Implementation Details

4.1 User Interface

Menu screens in the game consist entirely of labeled buttons, except for the InventoryMenu where item viewports are used as buttons. In game, the only UI information conveyed to the player is the name of the present scene and text via DialogBoxes. A custom theme has been created and can be applied to all UI elements to keep font, element opacity, and colors and styling consistent between user interface elements in different contexts.

4.2 Input Handling

The Godot Engine abstracts input handling and handles it automatically according to predefined inputs and references to those inputs in code.

4.3 Game Flow

```
MAIN MENU -> NEW GAME (Game.tcsn)
          -> LOAD MENU (LoadFromMainMenu.tcsn)
          -> EXIT
NEW GAME -> Stage1-1.tcsn <-> Hallway.tcsn -> Apartment.tcsn
IN GAME -> PAUSE MENU (PauseMenu.tcsn)
        -> SAVE MENU (SaveMenu.tcsn)
        -> LOAD MENU (LoadMenu.tcsn)
        -> EXIT
LOAD MENU (either) -> NEW GAME + injected GameData
SAVE MENU (SaveMenu.tcsn) -> serialize game state to GameData
```