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

[2] Adapted from http://www.cs.concordia.ca/---ormandj/comp354/2003/Project/ieee-SDD.pdf

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. Functional Areas

2.1 Main Menu (MENU)

This functional area covers the Main Menu, which is the first thing the player is greeted with upon loading the game. Here, the title of the game is displayed as well as the options to either start a new game file from the beginning or load a previously-saved game file to continue playing from an earlier state of progress. The player can also choose to instead exit the game and close its processes from here.

The New Game button instantiates the Game.tcsn scene and removes the Main Menu from memory. The game processes and overall hierarchy of components is pre-defined in Game.tcsn, so once that is loaded into memory and the Main Menu is removed, the player can play the game normally from the beginning state.

The Load Game button displays a new screen continuing the visual theming of the Main Menu that presents the player with three game files to select from. If no games have been saved previously, these files will start the player at the beginning state of the game in the same fashion as the New Game button on the Main Menu. If games have been saved into any of the files, the file selection buttons will be populated with information about the state of the game when the player previously saved. Specifically, the file number, name of the game scene the player was in, and the total in-game playtime the player had put into the game before saving. Selection of any of these files will instantiate Game.tcsn similarly to a New Game, however the saved game state data is then injected into Game.tcsn after being loaded into memory. This allows the player to resume progress as if they had never left their original play session.

The final options available on this screen allow the player to clear saved data, over-writing the save files with New Game data, or to return to the Main Menu.

2.2 Pause Menu (PAUSE)

This functional area covers the Pause Menu screen of the game, which allows the player to temporarily stop the physics and other processes of the game in case they need to step away from it for a moment but do not wish to end their session. Additionally, the Pause Menu contains functionality to display a Save Game screen and a Load Game screen, besides allowing the player to continue playing or exit the game and terminate all game processes.

The Save Menu and Load Menu that spawn from the Pause Menu are both similar in function and appearance to the Load Menu that spawns from the Main Menu, with some key differences. The Pause Menu darkens the game screen while paused and displays the menu options over the darkened game screen, both the Load Menu and Save Menu follow this pattern, whereas the Main Menu's Load Menu continues the visual theme of the Main Menu, as there is no underlying game process until the user starts a New Game or Loads a file.

The Save Menu allows the player to select one of the three game save files and record the present game state to that file, including the locations of both playable characters, which stage the characters are in, which character is active, and which items the player has collected. This data is encapsulated in a serializable class called GameData which is then serialized to JSON and written to the file. More information about this class and the data handling can be found in **3.3 Scripts** under the **DataTools.cs** heading.

The Load Menu functions the same as the Main Menu load screen in terms of game state injection, refer to the previous section **2.1 Main Menu** for a more detailed description of this process. The primary difference is that the game is active underneath this Load Menu, so the player can preview the game state as it is loaded and can load multiple game states before deciding which to choose. Returning from the Load Menu and unpausing the game after loading a file are intentionally manual operations, in contrast to the Main Menu automatically starting the game after a save file is chosen.

2.3 Gameplay (GAME)

This functional area covers the actual game itself, so it is a more complicated implementation than the other two functional areas. There are many components involved in facilitating the core game loop.

The character controller is one aspect of this functional area, it is implemented with the scripts PlayerA.cs and PlayerB.cs. These scripts define input handling such that the user can move the active player character around in the virtual world of the game, trigger interactions between the player and non-player characters and objects, and move the camera / viewport. These operations are all linked to input handling, which is discussed in **5.3 Input Handling**.

Another aspect of the game is the inventory system. When talking to non-playable characters or interacting with objects that are meant to provide the player with items, the player can receive an Item. All items are implemented following the generic implementation entitled DefaultItem, existing as both a script and a Prefab. When a player interacts with an Item, a DialogBox is displayed with the message “You got {name of Item}!” and a dictionary, held by GameControl, is updated to include the name of the Item and a boolean value of true. Once (an) Item(s) is/are obtained, the player can access a UI screen entitled Inventory Menu, where small boxes representative of each item in the player’s inventory can be selected to change which Item is presently equipped, or equip an item if nothing is presently equipped.

Besides the character controller and inventory system, there is additional functionality necessary to implement the game. Non-playable characters can be interacted with, similarly to objects, though not all NPC interactions provide the player with an Item. Generally, when the player interacts with an NPC, a DialogBox is spawned containing the first line of dialogue for that particular interaction. Upon further presses of the interaction button, the text in the DialogBox is cleared and replaced with the next line of dialogue, until there is no more text to be displayed, at which point the DialogBox is closed and despawned until the player attempts to interact with the NPC again. NPC interactions are prototyped in the generic implementation script entitled InteractableNPC.cs.

The player must be able to transition between game scenes. An example from the project is that the player starts in Stage1-1.tcsn, known as “Streetlevel” in-game, and upon interacting with the glowing door of the building the police NPCs are standing outside of, the player is transitioned to the scene Hallway.tcsn, known as “Hallway” in-game. This is facilitated by the same interaction system used for Items and InteractableNPCs, however once the interaction is triggered, the functionality is different. To transition scenes, the new scene is instantiated and then attached to the global node entitled “CurrentScene”. After that, the oldest child of CurrentScene (always the previous scene the player was in) is freed from memory. When the new scene is instantiated, a script called SceneInfo.cs (attached to the root node of the instantiated scene) updates the on-screen scene name display and moves the player characters to predefined locations in the new scene.

Additionally, the scene transition can be gated behind a necessary Item; if this is implemented for a particular scene transition, then the currently equipped Item held by the player is checked before performing the scene transition, only doing so if the player has the correct Item equipped. An example of this in the game is the transition between “Hallway” and “Apartment”, the player can only enter the “Apartment” if the “Keycard” Item received from the police NPCs in “Streetlevel” is equipped.

These elements all combine to create a gameplay loop wherein the player can talk to in-game characters, receive Items, and move around the game world. The intention is to build puzzles around these features, though the game is presently barren for content as it is strictly a proof of concept.

Other basic aspects of this functional area include collision between the player and level geometry, lighting and texturing for visual appeal, displaying which Item is presently equipped, ensuring the camera is not obscured by level geometry, and other concerns that are typical of 3D video games.

There are also nonfunctional requirements for this project that are primarily associated with the GAME functional area, as it is the focus of the project. These nonfunctional requirements are that the game runs with a decent level of performance given the scope of the project and the time allotted to complete it, the atmosphere of the game evoking the “cyberpunk” genre as well as 90s and early 2000s 3D video games, software stability, and fun.

3. Getting Started

3.1 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.

3.2 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.

3.3 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.

3.4 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.

4. Coding Standards

4.1 C# / Mono

Where possible, adhere to the official Microsoft coding standards.

4.2 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.

5. Implementation Details

5.1 Repository Overview

➔ **Docs** (documentation)

- ◆ SDS.pdf
  - Software Design Specification
- ◆ SRS.pdf
  - Software Requirements Specification
- ◆ TwinVisitor Programmer's Manual.pdf
  - This document. Contains information for prospective future developers of the project, explaining the files, organization, and methodology of the project implementation. Technical focus.
- ◆ TwinVisitor User's Manual.pdf
  - A manual for end users, explaining how to interface with and play the game. Uses plain english and obscures technical information that is not necessary for players.
- ◆ (Other Documentation may be stored in this folder.

➔ **Twin Visitor** (game files)

- ◆ **Environments**
  - Stage1.tres
    - Custom World Environment preset used for the initial area of the game and kept throughout the rest of the experience.
  - (Other World Environment presets may be stored in this folder)

◆ **Fonts**

- ming.regular.ttf
  - Font used in-game for all text and UI elements.
- Ming_ReadMe.rtf
  - License information for Ming font.
- (Other Fonts may be stored in this folder)

◆ **Images**

- background.png *(& background.png.import)*
  - Visual asset used in Main Menu of game as well as its import information for Godot.
- twinvisitor_logo.png *(& twinvisitor_logo.png.import)*
  - Game logo visual asset used in Main Menu of game.
- twinvisitor_logo.xcf
  - GNU Image Manipulation Program (aka GIMP) project file containing the game's logo. Kept in repository in case any edits to logo are required.
- twinvisitor_logo_clean.png
  (& twinvisitor_logo_clean.png.import)
  - Unused visual asset. Simpler, less edited version of game logo above.
- (Other images that are used plainly in-game may go in this folder. Images used for texturing go in the Materials folder)

- **Materials**
  - **KenneyMaterials**
    - Materials used for texturing 3D environmental models in the game.
    - These textures are sourced from Kenney, who makes assets for video game development
  - Other files in this folder are all materials used for texturing 3D environmental models. Character models contain baked-in textures.
  - (Other materials used for texturing in-game 3D models may be placed in this folder)
- **Models**
  - **KenneyModels**
    - Models used for creating 3D environment scenes in the game
    - These models are sourced from Kenney, who makes assets for video game development
  - *.glb files
    - These are all 3D models used for creating game scenes, these are assets sourced from free-for-personal-use collections of game assets
  - *.ecsn files
    - These are all 3D models representing in-game characters, both player characters and non-player characters.

        - These assets were all created in Blender specifically for this game.
    - *.import files
        - Information generated by Godot describing particular import settings for the files of the same names.
    - *.material files
        - Materials erroneously placed in the folder, these have not been moved to the **Materials** folder as it would break the asset-linking in the project.
    - (Other 3D Models, specifically just models, NOT **Prefabs**, may be stored in this folder)

◆ **Prefabs**

- **Items**
    - *.tcsn files
        - ◆ These are Godot scenes containing individual items that extend the DefaultItem class.
    - Keycard.cs
        - ◆ Script defining the in-game Keycard item, erroneously placed in this folder and left to preserve project linking.
- *.tcsn files
    - These are all Godot “scenes” containing individual in-game objects complete with models and textures and preset node trees, ready to be instantiated in game scenes.

- (Other scenes ready to be instantiated in-game may be stored in this folder)

◆ **Scenes**

- *.tcsn files
    - These are all Godot scenes containing complete game areas or menus. The files here are differentiated from those in **Prefabs** by their purpose. Prefabs are individual items or decorations meant to be instantiated in tandem with other prefabs and models to create a game scene. **Scenes** are those game scenes that are built out of prefabs and models. They can also represent complete UI scenes that can then be loaded anywhere in the game, as well as the in-game scenes.
- (Other complete game scenes meant to be loaded as environments in-game *or* complete UI screens saved as scenes may be stored in this folder)

◆ **Scripts**

- The contents of this folder are discussed at length in section **3.3 Scripts** under the **3 Getting Started** heading. Generally though, these are all C# / Mono scripts used to define game logic and behaviors.
- (Other C# / Mono scripts can be stored in this folder)

◆ **Sounds**

- *.wav files
    - These are all sound files used for music, ambience, and indicating that the player has done something such as interacting with a non-player character or obtained an item.
- *.import files
    - Information generated by Godot describing particular import settings for the files of the same names.
- (Other sound effects and music may be stored in this folder)

◆ **Themes**

- Default.tres
    - This is a custom theme preset used to ensure consistency between all UI elements and menus in the game. It contains information about the font used and the styling of various elements such as buttons and pop-up boxes. This can be used as a template or directly extended.
- Stylebox.tres
    - This is a custom sub-element of Default.tres above, specifically used to store information about the dialogue pop-up boxes in the game. This is also contained within Default.tres.
- (Other theme presets may be stored in this folder)

- default_env.tres
    - World Environment preset. Necessary for project as a default fallback if individual scene World Environments cannot be loaded or are not found.
- icon.png *(& icon.png.import)*
    - Game icon used by Godot for window decoration and building *.exe. Presently the default Godot icon.
- project.godot
    - File generated by Godot to contain meta information about the game project, its file, and what is necessary to build and execute it. Automatically generated by Godot and not intended to be directly edited or interfaced with.
- Twin Visitor.csproj & Twin Visitor.sln
    - Project data, necessary for project but not meant to be directly edited or interfaced with.
- Twin Visitor.exe
    - Executable file to play a current build of the game

5.2 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.

5.3 Input Handling

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

5.4 Game Flow

MAIN MENU -> NEW GAME (Game.tcsn)

-> LOAD MENU (LoadFromMainMenu.tcsn)

-> EXIT

NEW GAME -> Stage1-1.tcsn <-> Hallway.tcsn -> Apartment.tcsn -> EXIT

IN GAME -> PAUSE MENU (PauseMenu.tcsn)

-> SAVE MENU (SaveMenu.tcsn)

-> LOAD MENU (LoadMenu.tcsn)

-> EXIT

5.5 Project Organization

The Game.tcsn scene serves as a structural template for the game. The root node is GameControl, which holds the GameControl.cs script. All other nodes during gameplay are children of the GameControl node. The first children listed in the project are PlayerA and PlayerB, representing persistent instances of the two player characters. Other children of GameControl include the various UI elements such as the PauseMenu and InventoryMenu, among others. Finally, there is a child node named CurrentScene which acts as the root node for instanced game scenes. CurrentScene exists to simplify scene instancing and destruction when transitioning stages and was necessary to accommodate inconsistency in individual scene organization.

6. Adding Content

6.1 New NPC Interaction

To add a new NPC Interaction to the game, first import the desired character Model or Prefab into the current game scene in the editor, then place the character model in their intended location in the scene. Next, instantiate a duplicate of the InteractableNPC Prefab in the editor, disable its visibility, and place it in the same location as the character model. These concepts are kept separate for flexibility and convenience, such that any character model can be switched out with another, without losing the dialogue interaction. To add dialogue, inspect the InteractableNPC in the editor. There is a Script Variable section of the inspector that will contain an array of strings labeled “dialogue”. The array can be edited directly from here to add dialogue. Once dialogue is added, the NPC interaction is complete and will be accessible in the game.

6.2 New Item

To add a new Item to the game, instantiate a duplicate of the DefaultItem Prefab. A script inheriting from DefaultItem can be created or a new class based on the DefaultItem script can be made, either method will follow the same process. If the item is part of an NPC interaction, it can be made invisible then all that is necessary is giving it a unique name and placing it in the game scene. One caveat of this however, is that there is no generic code written for scripted Item instantiation, so it must be manually implemented. An example is provided in the CopNPC interaction in Stage1-1.tcsn.

A model can be added as a child of the Item in cases other than NPC interaction, where representation of the item should exist in the game world. Then the item can be named and placed in the desired location in the game scene.

6.3 New Scene Transition

To add a new scene transition, instantiate a LoadScene Prefab. Inspect the LoadScene node in the editor. Find the desired scene in the file explorer, right click, and select “copy path”, which will return a relative path to the scene within the editor. Paste the path into the scene path field in the Script Variables section of the inspector. Underneath that is a field where a necessary Item can be specified as discussed in **2.3 Gameplay (GAME)**. Once this is complete, the LoadScene can be placed within the model that is intended to trigger the scene transition in the current game scene.

6.4 New Game Scene

To add a new game scene once level layout and geometry have been created, add the SceneInfo script to the root node of the game scene. Select the root node in the inspector and copy the path to the desired scene as described in the previous section, **6.3 New Scene Transition**. Paste the path into the scene path field in the Script Variables section, then add the in-game name of the scene in the adjacent field. To place player spawn points, create two empty spatial nodes and place them where the desired spawn locations are for PlayerA and PlayerB. Copy the Basis, aka Transformation Matrix, from each spatial node and paste the values into the matrices in the root node’s Script Variables in the inspector. After this, a LoadScene node must be placed in an existing game scene to allow the player to access the new scene. This process is detailed in the previous section, **6.3 New Scene Transition**.