Detailed Design

LateNite Design

JID Team 0111

Tristan Sallin, Jacqueline Chambers, Kacey Chung, Josh George, Araneesh Pratap

Client: Professor Suzy Watson-Phillips

https://github.com/TASallin/Project_Implementation

Table of Contents

List of Figures

Introduction

Background

LateNite is a video game product intended to provide incoming Georgia Tech freshmen with a preview of how they can expect their first semester in college to go academically and financially. It is a visual-novel styled game that focuses on choices and their consequences across four chapters of increasing complexity. It can be used during the orientation process to make such process more enjoyable and understandable by giving oncoming freshmen a simplified, virtual campus. We are creating LateNite in Unreal Engine 4, the most mainstream game engine, using an anime art style and a visual novel type of gameplay. Latenite will boost their time and money management skills using realistic scenarios and personalized feedback.

Document Summary

This report documents LateNite in four main parts.

The system architecture section describes how we use Unreal Engine's features to create a smooth gameplay experience, as well as the external tools we used for art assets such as Blender. It also details the flow of the game loop as experienced by a typical player.

The data storage design section describes how we import the Georgia Tech calendar to be more accurate during gameplay, and the password system that allows users to save their progress without proper save files.

The component design detail section provides details of the various screens of the game and the transitional state machine used to navigate between them. It also shows how the backend and UI of the game communicate with each other.

The UI design section shows the design behind our UI decisions for the game's menus, and how we create a consistent UI experience across the game's screens.

Terminology

Actor: A type of class in Unreal Engine that can be placed in the scene directly, like a 3D-object.

Bake: A playable version of a game in Unreal Engine that is compressed and encrypted.

Blueprint: The programming of language of Unreal Engine, which uses node-based programming.

Chapter: Represents a portion of the game whose data does not depend on other sections of the game.

Controller: Represents the code that takes user input such as button presses and updates the UI and backend with it.

CSV File: A comma separated values file that functions as a data table with rows and columns.

Function Library: A set of blueprints in Unreal Engine that can be called by any class, thus functioning like a package.

Game Loop: A term for the player's interactions with the main part of the application excluding the main menu and the code that runs it.

Map: A file in Unreal Engine that holds all of the actors and settings for a single screen the player views.

Row Structure: The names and data types for the rows of an array.

Singleton: A design pattern where runtime information is stored in a static variable.

UI: Stands for User Interface, the 2-dimensional overlay players can see on their screen. It includes text boxes, images, and interactable buttons.

Unreal Engine: The software used to develop this project, a popular game engine associated with Epic Games.

Widget: Unreal Engine's user interface class, taking forms such as buttons and text boxes.

Detailed Design - System Architecture

Introduction

In order to understand the structure of our game we have provided designs of the architecture of our system statically in Figure 2.1 and dynamically in Figure 2.2. These diagrams go above the smaller game design details to model the higher level components and dependencies of the game. In the static diagram we show how each system in our game communicates with each other, and in the dynamic diagram we show the flow of communication at runtime as the player goes through the game. The overall design pattern follows a model-view-controller style with data as a separately treated component. We will also focus on how Unreal Engine's unique properties affected our architecture decisions, mainly its UI system with on-click buttons and special GameInstance class.

Static System Architecture

The static architectural design shown here provides a functional view of the system. In this diagram we show how the components are interrelated.

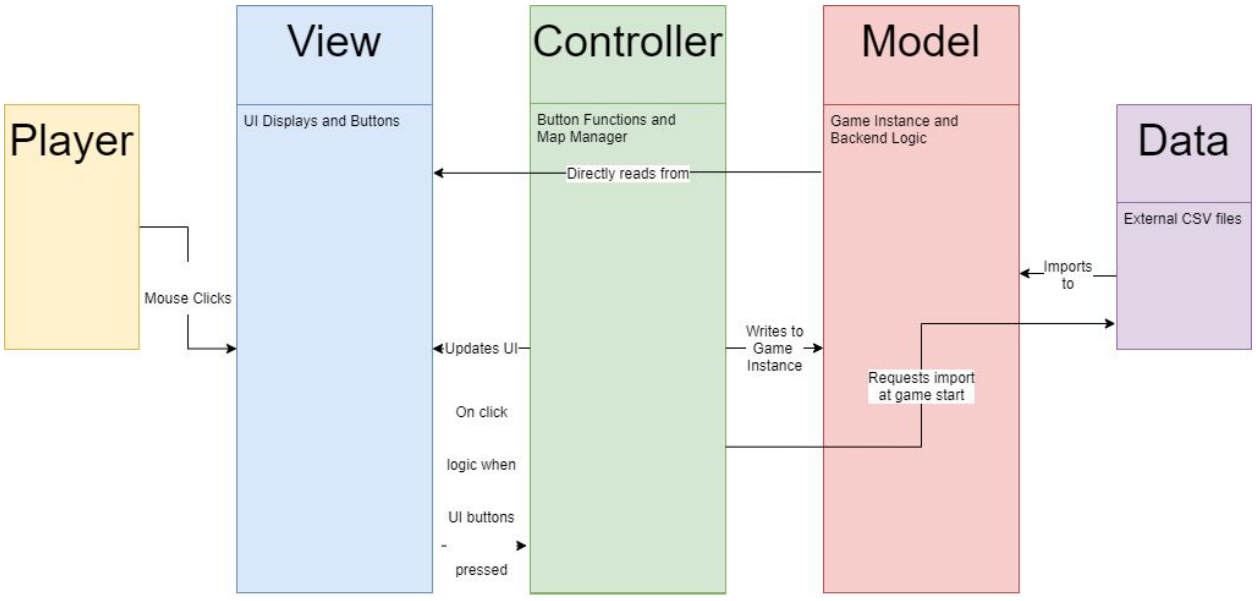

Figure 2.1: Static System Architecture Diagram

The view is fairly straightforward, using the dialogue system plugin to display text and Unreal Engine's widgets for other UI elements. The controller in this application is more abstract than concrete, consisting of mostly UI blueprints that govern the on-click functions of buttons. As such, it's main job is to communicate the player's mouse clicks to the model and update information accordingly while sometimes changing the UI without consulting the model. The map manager is a specially created class whose job is to load new maps of the game when directed by the model. There is one exception to the standard of information passing through the controller - UI elements can hold variables that are direct references to the backend, so any updates there update the view without using the controller.

The data is separate from the model in our diagram because of how uniquely it is treated by Unreal Engine. The calendar and event data are both csv files that are edited externally, which are then imported into unreal and converted into a data table class that is usable by the game's code. Within the model the game instance functions as Unreal Engine's version of a singleton, which stores dynamic information about the player. We chose to use a singleton despite its drawbacks for 2 main reasons. The first is that the scope of the game will not reach a point where it becomes difficult to edit the code because of the singleton. The second is that there is no save system so we needed someplace to hold data that would normally go in a save file. We decided to forego a save system since each chapter is short and is intended to stand alone, so players only need a way to start each chapter. Using a traditional save system would thus cause more security issues with login than it is worth.

Although there is no save feature for the game, it can be played across multiple sittings using the password feature. At the end of the first three chapters, players see a password that let's them skip the completed chapter(s) from the main menu. Although it is possible players could share the passwords online, we don't anticipate this being a serious issue because if the game is designed properly to be fun, players will not wish to skip chapters of it, and as the game is intended for freshmen orientation, the target audience will not have widespread connections to each other at the time they are playing.

Dynamic System Architecture

The dynamic architecture design shows the control flow within the system and the interactions between components of the system as information is exchanged. The figure

below depicts an expected path of a user who plays through the game from start to finish, with some omissions of self-contained paths such as pausing and unpausing the game.

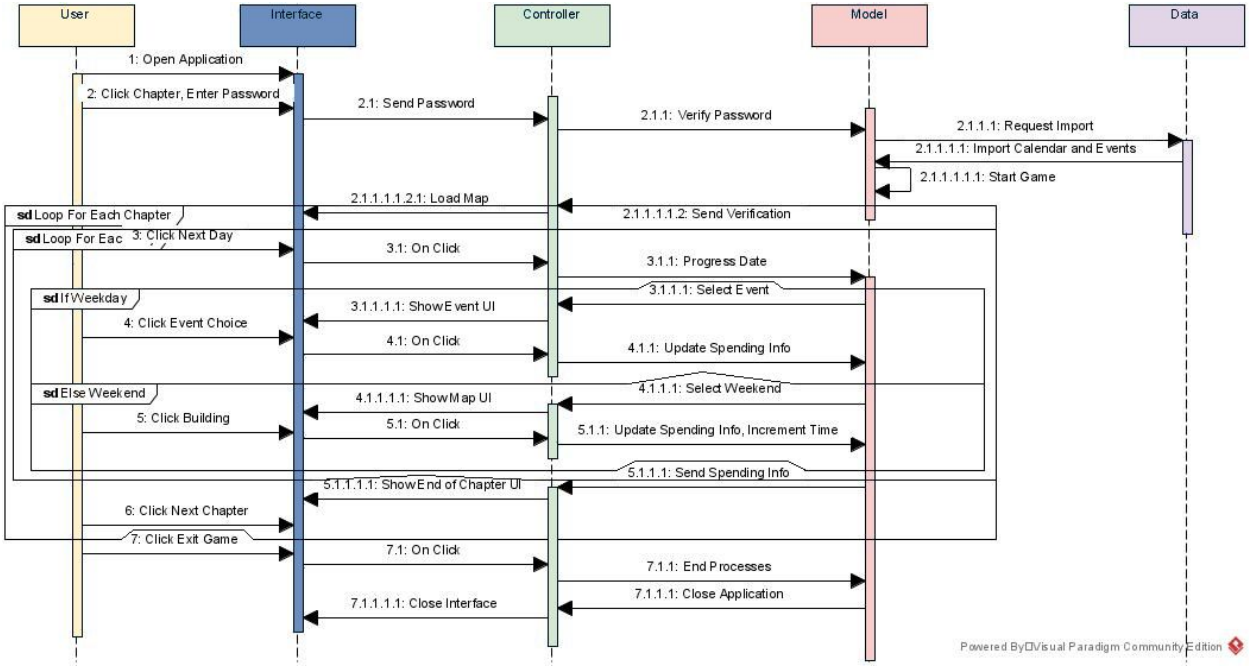

Figure 2.2: Dynamic System Architecture Diagram

The largest loop in Figure 2.2 that iterates through the chapters is what's commonly referred to as the game loop - the player's experience after they load the game from the main menu. By nature the game loop offers various choices to the player, and this diagram has streamlined the decision making to exclude confirming choices or viewing various optional menus. Most players will play all four chapters in one sitting, and starting from chapter 1 will bypass the need to confirm a password. Within a chapter, there will be about 30 days where the conditional in Figure 2.2 applies, although some of these will be skipped entirely with no gameplay, representing holidays. The rest of these days will be classified as weekdays where players will receive an event prompt and choose one of the given options or weekends where players will see a map of Georgia Tech and spend virtual time at buildings of their choice. The exit game process is portrayed after playing through all chapters, but can be initiated at any time during gameplay if the player wishes. The data is connected only briefly to the game as it is modified into a different form when imported to be used during gameplay, meaning the data files themselves are no longer used.

Detailed Design - Data Storage Design

Introduction

LateNite is a project that does not use a large amount of data, however it does make use of it and does so in unorthodox ways. In the sections below, we will explain the role data has in our project, how it is exchanged, and how we will keep it secure.

Database Use

In the figure below, we show the small amount of data that is used by our game. The application box represents the build of the game that has been baked, or compressed and encrypted by Unreal Engine. This makes any data within it inaccessible by users. The calendar .csv file can be optionally imported by the user, with the row structure shown in the diagram. This row structure is almost identical to the one used by the official GT calendar, with just the index column added to comply with how Unreal Engine handles data tables. If there is no index column, the import will fail. Since our external data is so small, we are using no official database design style.

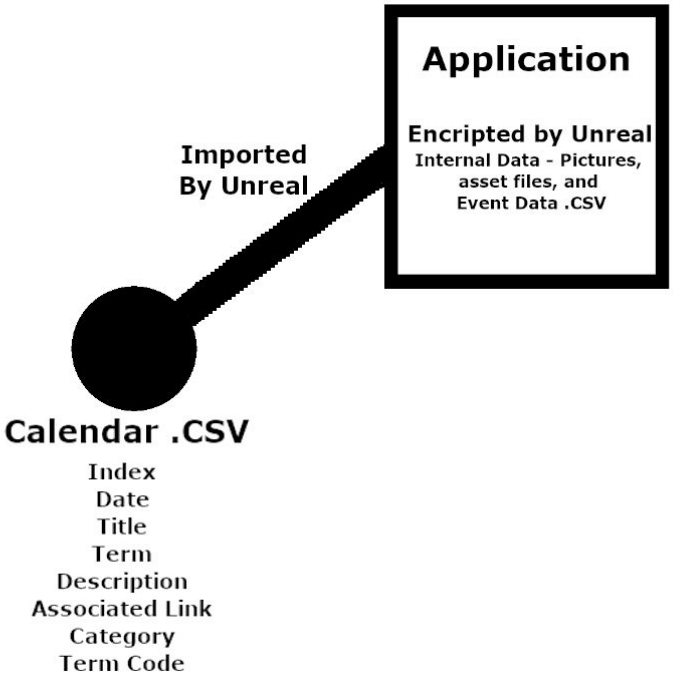

Figure 3.1: Data Storage Visual

File Use

The external calendar file will be placed in the same folder as the executable game file using the name 'gtCalendar.csv'.

All other files, including image files, are encrypted within the baked version of the game.

Data Exchange

The only way to exchange data across devices is to send the modified gtCalendar.csv file to the target device alongside the game application. For faculty and staff wishing to give students a specific version of the calendar this will entail sending a zip file containing their gtCalendar and the application together to any students that will be playing the game. The game's internal data resets each time it is opened, so the game can be sent to different devices by copying the files without any loss of fidelity or change in the player's experience.

Security Concerns

There are not many security concerns with Latenite compared to the average application. Users will not log in, save their data, or make financial purchases. If the gtCalendar is modified in some way that is unreadable by the game, it will use default data instead. Due to the encryption of the game Unreal Engine uses, it cannot be modified by users, so they can only send the entire game to other people. The one main concern is the password system, which was addressed in the system architecture in detail. In short, we believe that password sharing to access the chapters out of order will not be a significant problem.

Detailed Design - Component Design

Introduction

This section elaborates on the system architecture section and examines the systems of the game in greater detail. In the static section, we show how each screen the player can see interacts with each other and the backend classes of the game. In the dynamic section, we focus on the connections between each component highlighting which are part of the model, view, and controller.

Static

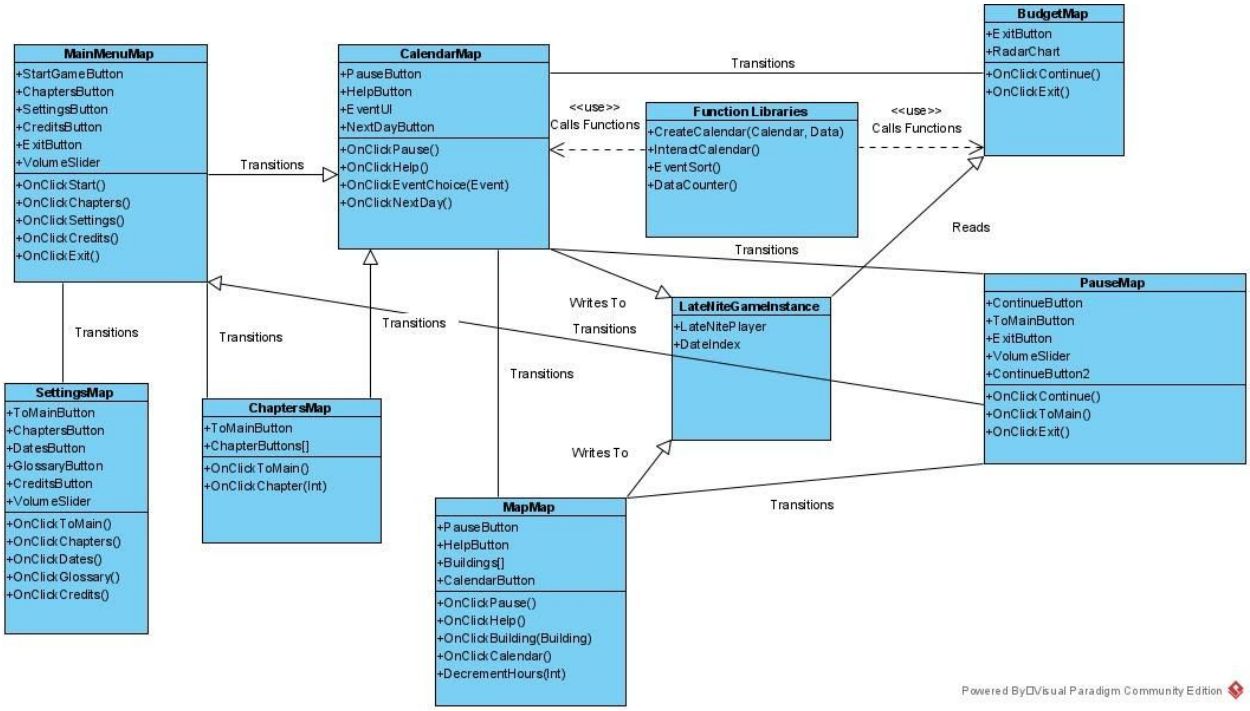

Figure 4.1: Static UML Class Diagram

The class diagram in the figure below represents the detailed components of the system, presented in static form. We decided to use the Unreal Engine maps to represent most of our classes. These maps are a hierarchy of game elements such as visible actors and UI widgets. For sake of simplicity, we have chosen to omit actors that do not affect the game

in an interesting way such as cameras or background images. If these were included they would serve no purpose other than add noise to the diagram and cause confusion when there is no need to understand how every small detail of a game works. The majority of the communication between maps are simple transitions from one to another caused when the player presses a button. One-way transitions are marked with an arrow. The other main classes we are showing are the LateNiteGameInstance singleton and the function library. The singleton contains information about the player that will be written to and read as shown in the diagram, allowing maps to communicate to each other in a more advanced way. The function library represents operations that can be called from any scene and holds no runtime variable data of its own.

Dynamic

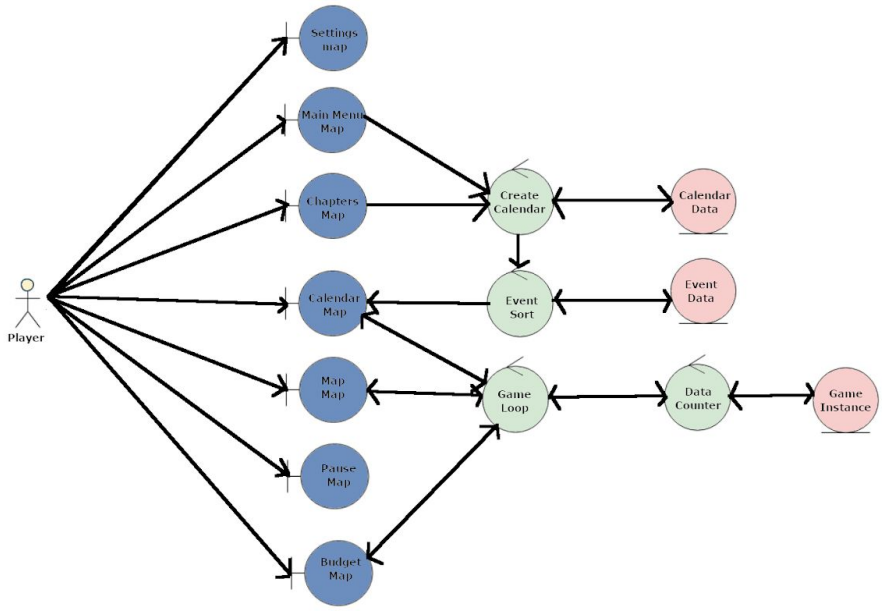

Figure 2: Dynamic Robustness Diagram

Our dynamic component design can be seen through our robustness diagram below. It shows the runtime interactions between the user and system and how each of our individual components interact. The UI components are Unreal Engine maps, which as explained above represent a single screen the player sees. A player user sees the main menu map when first opening the application, where they can change their settings before

initializing their game. In this process, the calendar and events are created using the backend data before directing the player to the calendar map. The lower half of the diagram shows the gameplay loop the player experiences, which is primarily a communication between the calendar map used for weekdays in the game, map map used on weekends in the game, budget map shown at the end of the chapter, and GameInstance singleton. A typical player might start on the main menu map, change the settings, start the game, then jump between the calendar map and map map several times before viewing the budget map and closing the application.

Detailed Design - UI Design

Introduction

In a video game the UI is not considered to be everything the player sees but instead is an overlay of widgets that is displayed over the scene hierarchy. There are two main types of UI used in Latenite. The first is displays that are built into a map itself, and the second is menus that can open and close within a map. In general, we are keeping our UI simple to streamline gameplay and provide the player with an easy to understand experience.

Main Menu

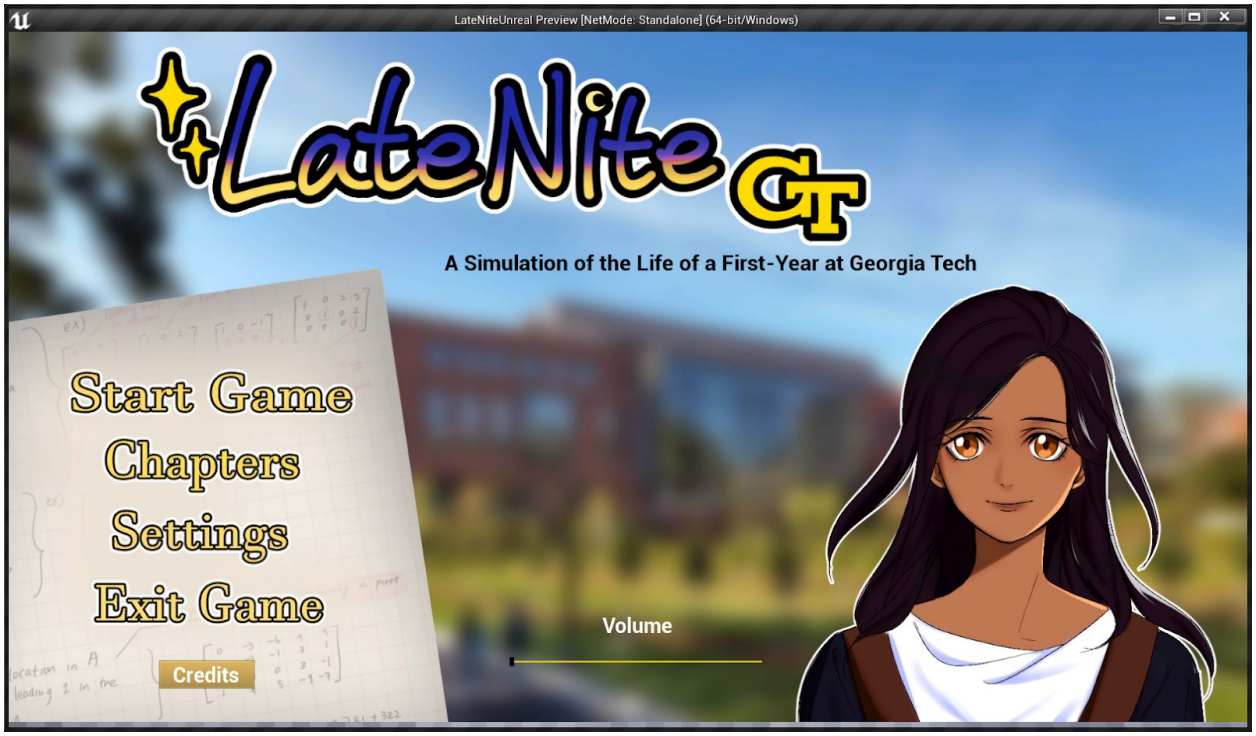

Figure 5.1 - Main Menu UI

The main menu is the first thing the player sees upon starting the game, so we made sure to put a relatively greater amount of time into its UI compared to other screens. We used

Georgia Tech colors for the buttons to reflect the institution and drew an image of the tutorial character to make the game feel welcoming. For the buttons, any game's main menu needs to have a start game and exit game button, and we included a chapters button as an alternative way to start the game, the settings button so players can immediately set their preferences without having to search for the settings, a credits button which is commonplace on modern games, and lastly a volume slider so players can change the volume of the game relative to other applications on their computer.

Settings Menu

Figure 5.2: Settings Menu UI

The settings menu retains some features of the main menu such as the background to make the transition between the two menus more seamless. The two important options here are the set dates button and the glossary button. Set dates will require a password that is given to administrators and allows them to set the start and end dates for the in game semester to match up with the academic calendar. The glossary will display various terms used in the game and their descriptions to help reduce player confusion.

Chapters Menu

Figure 5.3: Chapters Menu UI

The chapters menu is the latest piece of UI added to the game, and the four transparent areas are text boxes for the player to type in. If they enter the correct password, they will be taken to the chapter in question. Like the settings menu, we kept some aspects of the screen the same from the main menu to make the transition more seamless.

Calendar Map

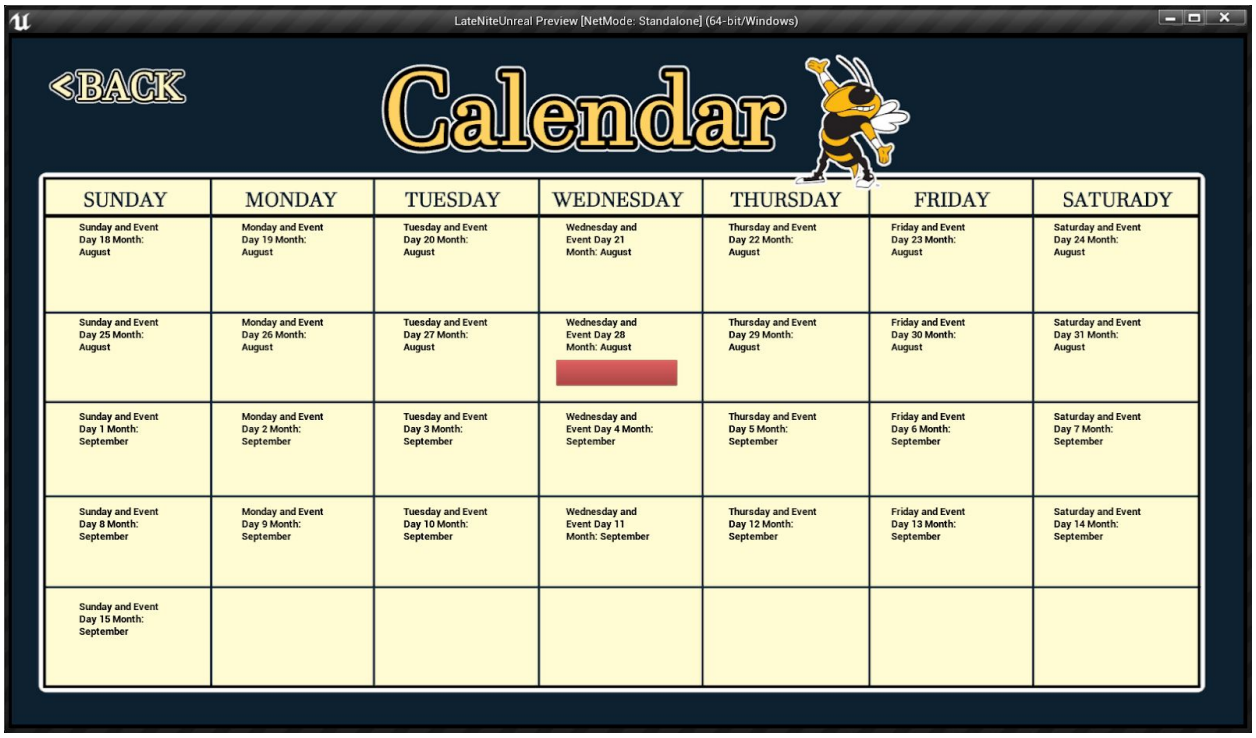

Figure 5.4: Calendar UI

The calendar map will be shown to the player frequently during the game-loop. This version in figure 5.3 is not the final version, which will be much more populated. The red bars are placeholder buttons for events that the player will click on once per in-game weekday. As days progress within a month, they will be marked with an X so the player will know what the current day is. The back button will be used for closing the calendar and moving to the map or budget maps. Finally, the calendar UI is stylized in GT colors and has buzz on it.

Event Pop-Up

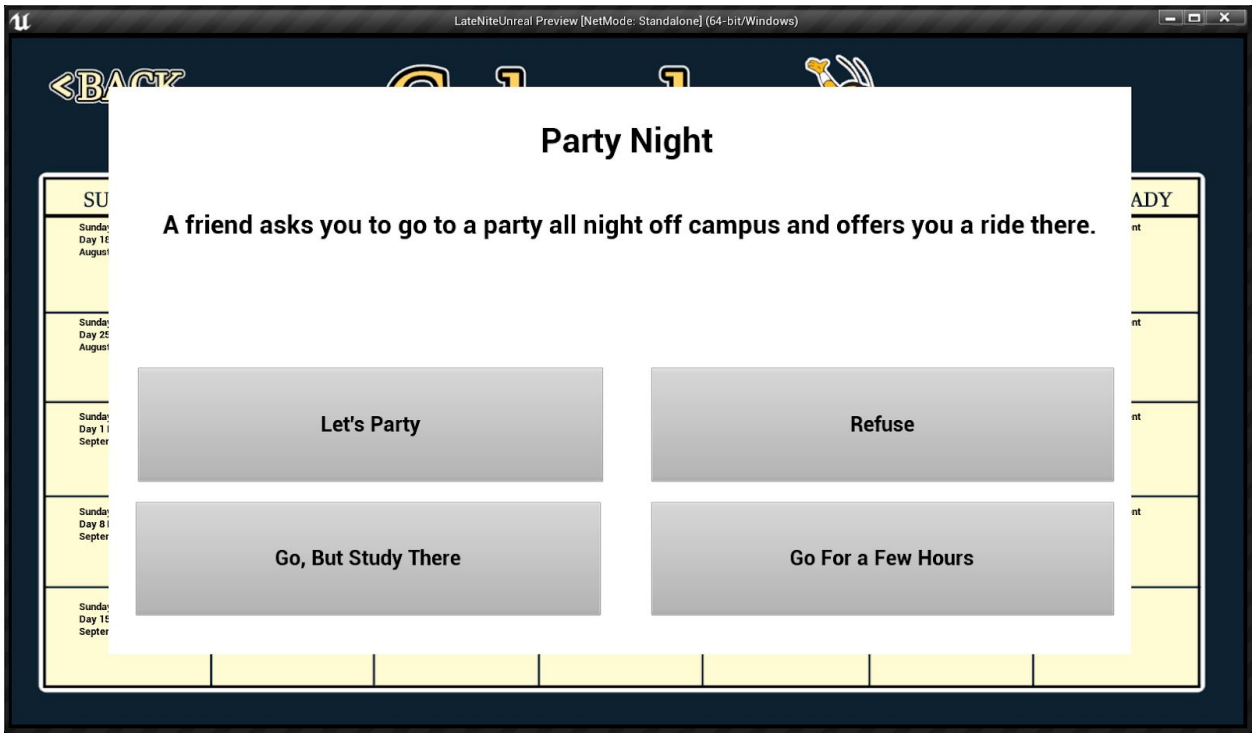

Figure 5.5: Event UI

The event UI appears as a pop up menu from the calendar screen when a day is clicked on, presenting a prompt and choices to the player. The layout is very basic and designed with other video games that have similar event systems in mind. Once the player clicks on one of the four buttons, they will be shown a text box that details the consequences of their choice before the pop up menu closes.

Map Map

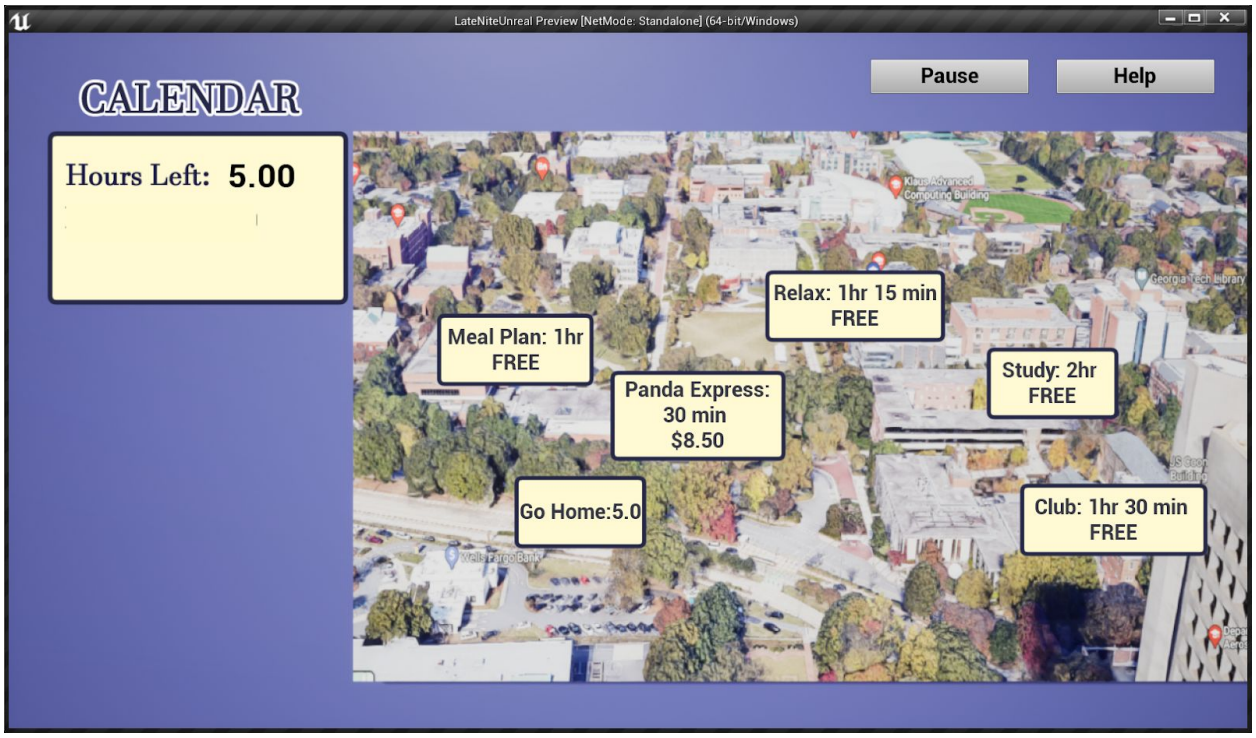

Figure 5.6: Map UI

The map UI is the most complex of any screen and offers the player the most freedom as well. Each of the buttons on the map of Georgia tech itself can be clicked to let the player perform the described activity. Depending on the button pressed, the three meters, hours, and budget can change. The energy meter increases if the event involves eating and decreases otherwise. The stress meter builds if too many activities that aren't recreational are picked. The knowledge meter increases as the player studies, but study activities cannot be chosen if the stress meter is full or the energy meter is empty. The budget and hours decrease as activities are picked and prevent the player from choosing more expensive activities than they can afford. When the hours reaches 0, the player is moved to the calendar map. Outside of the map itself, the calendar button lets players view a non-interactable version of the calendar UI, the pause button pulls up the pause menu, and the help button displays text that goes over what the player is supposed to do on this screen.

Budget Map

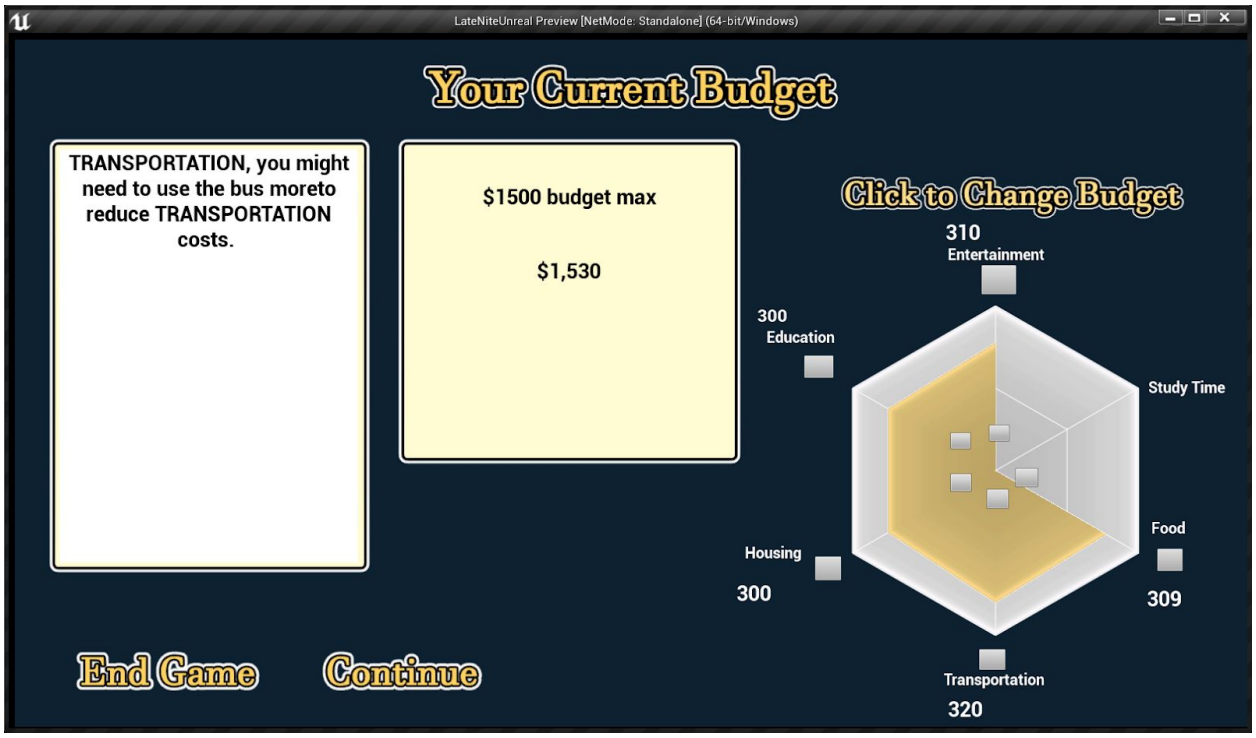

Figure 5.7: Budget UI

The budget UI is displayed at the end of the chapter and functions as a summary of the player's actions during that chapter. The white box on the left displays text based on the player's spending habits - it lets the player know if they are over budget or if one of their category costs like housing is unusual. The yellow box's text isn't implemented yet but it will display fun facts about student spending at Georgia Tech and colleges in general. The radar chart shows the player's spending in each category as well as their study habits. The boxes on the outer ring increase the category costs, and the ones in the inner ring decrease them to let the player tweak their spending plan. If the player is over budget, they will have to edit their radar chart to get within budget before pressing the continue button to move to the next chapter. The end game button returns the player to the main menu and can be pressed at any time.

Credits Screen

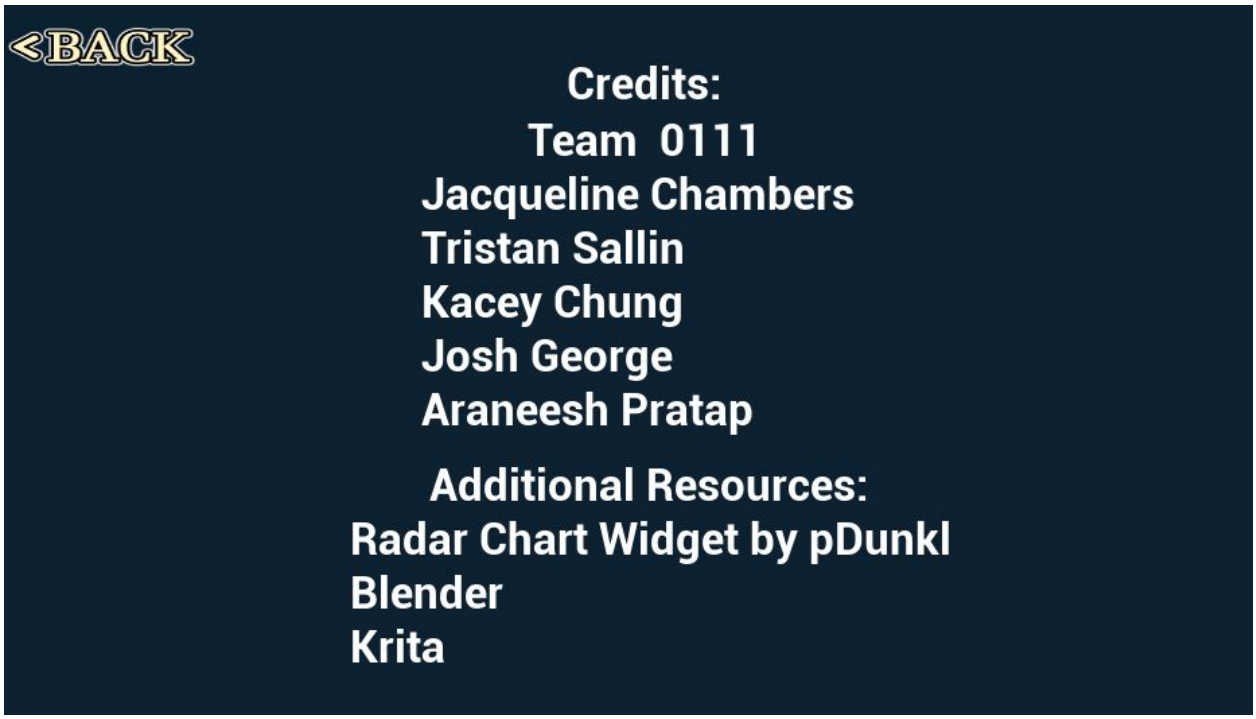

Figure 5.8: Credits UI

The credits screen is an important part of any video game and is accessible from most menus. It's minimalistic to keep the focus on the contributors to the project.

Detailed Design - Appendix A

Github Repository

https://github.com/TASallin/Project_Implementation

(Links to an external site.)

This is the version control repository our product is linked to.

Unreal Engine Documentation

https://docs.unrealengine.com/en-US/index.html

(Links to an external site.)

This external site gives documentation for the software our product is built in.

Radar Chart Plugin

https://unrealengine.com/marketplace/en-US/product/radar-chart-widget

(Links to an external site.)

This is the plugin we bought from the Unreal Engine store to help construct the radar chart used in the budget screen of the game.

Detailed Design Team Collaboration Form

Team member name	Role(s) performed for each draft (e.g., 1st, 2nd, final)
Jacqueline Chambers	Proofreader (all drafts)
Kacey Chung	Pre-Writer/Brainstormer (all drafts)
Josh George	Pre-Writer/Brainstormer (all drafts)
Araneesh Pratap	Proofreader (all drafts)
Tristan Sallin	Drafter, Delivery Coordinator (all drafts)