Conceptual Architecture Report

Team Name: Pingu Simulator
Members: Ryan Mahjour, Wesley Knowles, Zack Harley, David Wesley-James, Alisha Foxall, Michael Zadra
Student Numbers: 10142829, 10149982, 10135795, 10104425, 10129909, 10057216
Due Date: October 23, 2017
TA: Dayi Lin
Instructor: Ahmed E. Hasan

Table of Contents

Abstract

SuperTux is a free and open-source two-dimensional platform game that was released in 2003. The game was inspired by Nintendo's Super Mario Bros. series. It was originally created by Bill Kendrick and is currently maintained by the SuperTux Development Team.

The purpose of this report is to investigate SuperTux and propose a conceptual architecture for it. By reviewing documentation, the game itself, and its source code, we gathered information on which we based our architecture. Using this, we determined that SuperTux uses a hybrid object-oriented and layered architecture style.

SuperTux consists of many components that interact with each other to function cohesively. Some of these components consist of objects, such as maps, sprites, and menus. These objects are organized into a layered architecture, where they communicate with each other through their connections within the layered structure. This means that adding objects to the game does not need to affect other layers.

This report attempts to determine the conceptual architecture of SuperTux, explaining its sub-components and analyzing two sequence diagrams to demonstrate our understanding of the overall structure of the SuperTux system.

Derivation Process

The derivation process for the architecture of SuperTux included research of online documentation and resources, playing the game, having group discussions, and iteratively revising our derived architecture from feedback and findings. Our group started initial discussions after playing the game and taking a brief look at the repository of the source code for the game. We then looked for a general architecture that would match our findings.

Our group initially hypothesized that the architecture of SuperTux was solely a layered architecture. As there were no network components and the game seemed to have a simple hierarchy of game elements and objects. Our view of the game architecture changed to incorporate an object-oriented style, as it would better demonstrate the interaction between the components of the game. This lead us to a hybrid architecture of object-oriented and layered which would allow for the game objects to pass information to one and other and the game elements and components to pass information through the layers.

We then split the hybrid architecture into six subsystems: Interaction Layer, Input Handler, Game Elements, Data Manager, Resource Manager, and O/S Layer. We then analyzed each of these layers and their respective components to find out the relationships between them. Below, Figure 1 shows how these layers and components communicate with each other. Further details can also be seen in the sequence diagrams. Throughout the derivation process, we made constant revisions to our architecture model as we discovered more about the relationships between the components.

High Level Architecture

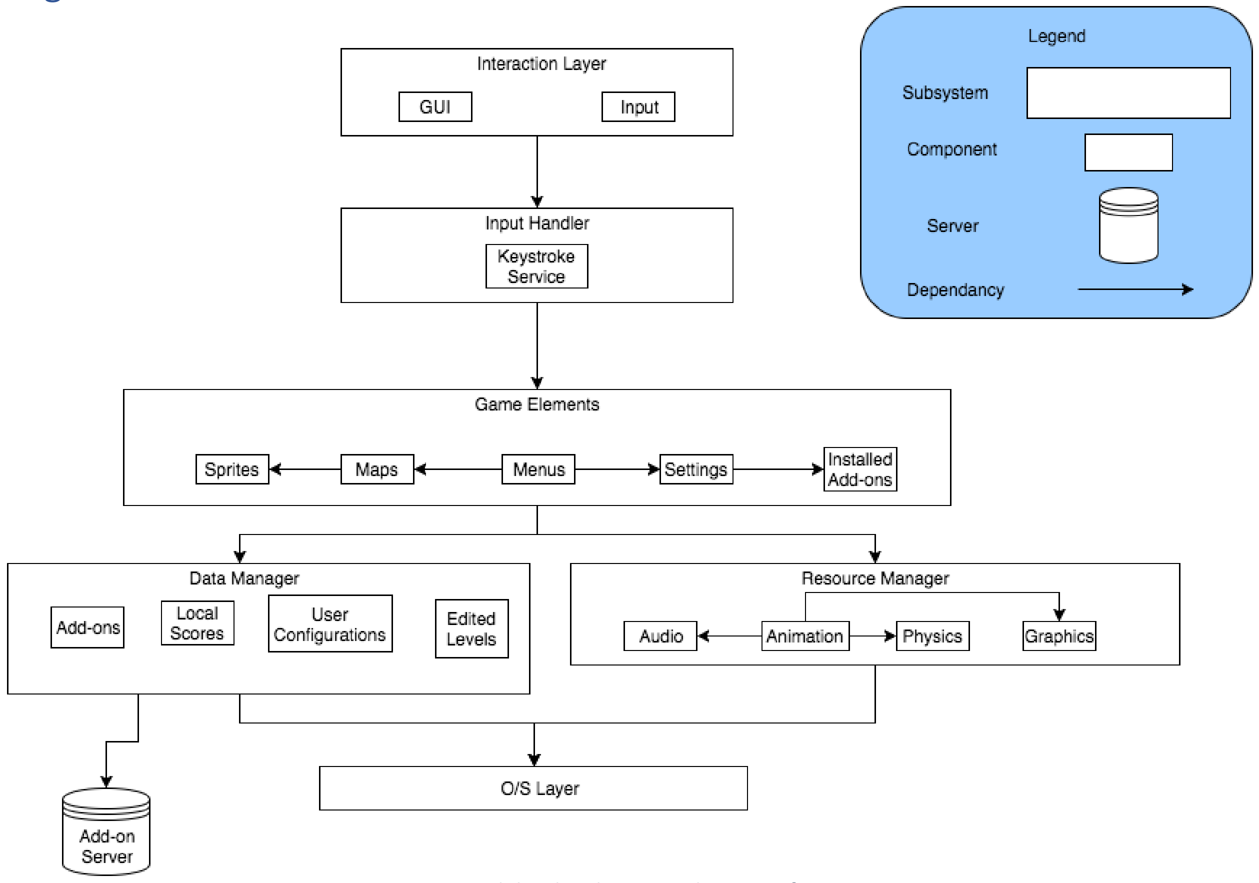

Figure 1: High level architecture diagram of SuperTux

A hybrid style architecture was derived for the game SuperTux, in which linearly controlled subsystems act as loosely coupled layers of a layered styled architecture, and inner components of the subsystems interact as objects in style of object-oriented architecture. A total of six subsystems were identified for the architecture which include the *Interaction Layer*, *Input Handler*, *Game Elements*, *Data Manager*, *Resource Manager*, and *O/S Layer*. Each layer invokes or obtains procedures and services from the layer below it in a hierarchical structure. An additional *add-on server* component was identified in the architecture, which serves as the online add-on repository accessed through a network call through the game to add additional features to the game.

The *Interaction Layer* includes the GUI and Input components, and is responsible for displaying and handling interactions on the user interface of the game. The *Input Handler* handles decoding user inputs from the Interaction Layer with the Keystroke Service component into specific commands and controls in the game. The *Game Elements* layer handles user commands invoked from the *Input Handler* layer, and calls instantiations of game element components to execute some game-related actions. The components include Sprites, Maps, Menus, Settings, and Installed Add-ons. The *Data Manager* layer handles locally storing game data, and persists user configurations of the game according to actions executed in the game invoked by the upper *Game Elements* layer. This layer also handles accessing and downloading add-ons from the SuperTux Add-on Server via network calls. The *Resource Manager* handles processing animations and audio resources via the graphics of the games according to calls invoked from the *Game Elements* layer. The final layer, the *O/S Layer,* is an abstracted layer which handles allocation of memory and storage for the game from the *Data Manager* layer, as well as manages system resources required by the *Resource Manager* used in the game.

Layered and Object-Oriented Architectural Style

The conceptual architecture for SuperTux is a hybrid architecture that combines both layered and object-oriented architecture styles. Figure 1 shows that there is low coupling between parts and the general flow of information and control is linear. This is a characteristic of a layered architecture.

The advantages of a layered architecture in SuperTux are system design, enhancement, and code reuse. The layering provides abstraction for easier understanding of components. This makes it easier for developers joining the open-source project to make contributions without having to understand the entire project. Furthermore, the layered architecture allows for code reuse. Many of the same elements are used in different parts of the game, and the layered architecture allows developers to create new components while the layers of the system remain the same.

Within each layer is an object-oriented approach to the system. This creates highly cohesive layers with many reusable objects inside them. In addition, the object-oriented style helps with the maintenance of the system. Due to there being many different developers working on the project, they can easily use existing interfaces to add changes without requiring a complete modification of a layer. However, developers still must be mindful of the coupling between components in the layers as any alterations to the components would affect behaviours of other components.

Subsystems

Interaction Layer

The interaction layer is the point of entry into the game for the user. It is roughly split into input and output, each of which has functionality for various input and output devices. Output goes to the screen and speakers, and possibly to a vibration motor on some devices if haptic feedback is part of the game. Input is more flexible, as the game can handle input from all kinds of controllers from keyboards to Guitar Hero guitars. The OS and device drivers help provide an abstraction for these devices, but it is likely that the input system contains a sublayer which maps inputs to functionality before handling that functionality.

Input Handler

The input handler subsystem deals with decoding user interactions and passing them on to the appropriate game element. In other words, the input handler is the middleman between the user and the game elements.

The input is given from the Interaction Layer. This raw input is decoded via the keystroke service. This service is specifically built to handle hardware interactions from a keyboard where the inputs from the keys would be mapped to their specific values. These values are then passed along to the corresponding game element, based on the context of the interaction.

Game Elements

The Game Elements Layer is the subsystem that contains objects that are viewed and interacted with by the player. They invoke procedures on the resource manager to call specific files, or execute certain physics. The Sprites component handles the properties, models, and physics of characters within SuperTux. It uses Animation, Graphics, and Physics within the Resource Manager in order to do so. Menus and Settings handle in-game functionality and navigation between different parts of SuperTux. They use Graphics within the Resource Manager, but mainly rely on the Data Manager for components such as Configurations. Add-Ons relate to additional maps that can be added to the game. The acquiring and storing of Add-Ons is within the Data Manager, but Add-Ons are also game elements in the sense that they are added in as elements of the game that players can use. Maps is responsible for the global map of SuperTux levels, as well as individual levels that can be played. Maps contain sounds and effects, contained within Audio in the Resource Manager. Additionally, maps require Graphics from the Resource Manager to display them to the user. Finally, Texts are pop-ups when characters speak, or to display tutorials or other text prompts. This requires the use of Graphics within the Resource Manager to display the text.

Data Manager

The Data Manager Layer handles the data storage and communication to the Add-on Server. This layer is composed of Edited Levels, User Configurations, Local Scores, and Add-ons. There are no dependencies between objects within this layer. The Edited Levels hold any locally created or edited levels. User Configurations keeps track of the user's personal settings such as language, resolution, etc. Local Scores holds all the local high scores for all maps played. Add-ons has two parts to it. Add-ons keeps a store of all the downloaded add-ons and it also communicates with the Add-on Server, allowing for further downloads.

Resource Manager

The Resource Manager Layer is responsible for handling most of what the user sees and hears, and is composed of three processes and one data store. The physics process handles the physics of all game objects, which includes moving tux around the map and handling all collisions, whether that be between tux and an enemy, blocks, or powerups. The animation data store depends on the Physics process. It takes the affected game objects and finds the appropriate animation. The Audio process has two facets to it. The first depends on the Animations, which takes an animation and plays the appropriate sound bite to go with it. The second actually depends on the Game Elements Layer, specifically the Maps and plays the background music for the level. The graphics process also depends on Animations, and handles all the rendering and drawing to the screen of all game objects and animations.

O/S Layer

The operating system abstraction layer is a subsystem used to make porting the software system to new hardware platforms. Due to the significant differences between operating (including things like filesystem, hardware support/drivers, memory allocation, etc.), this layer is necessary to make developing for multiple platforms.

This layer provides an interface to write programs that are cross-platform or, in the best-case scenario, platform-agnostic, meaning they can be installed on multiple or all platforms due to this abstraction. In the case of SuperTux, this layer allows the game to be run on multiple platforms, without involving a huge rewrite.

Use Case

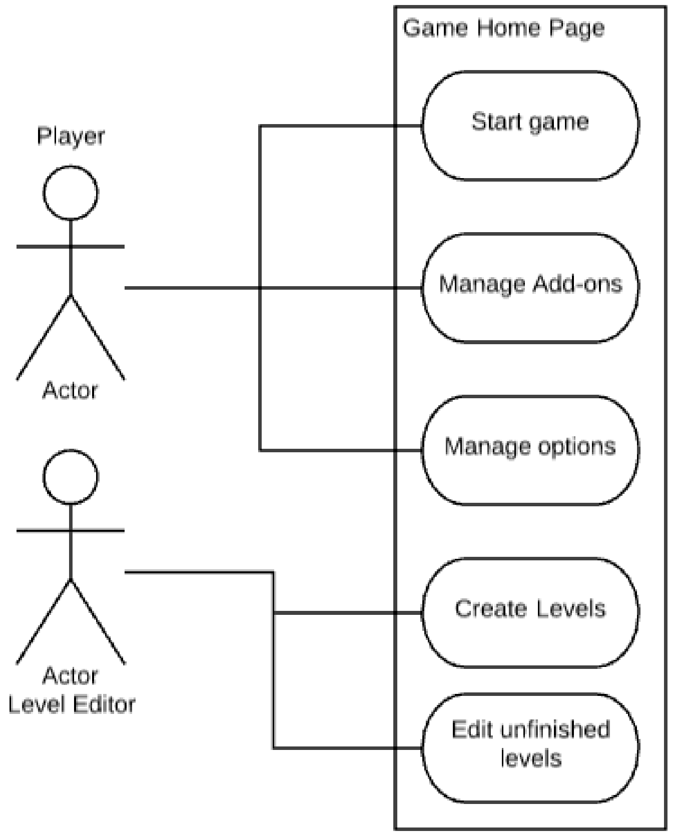

Figure 2: Use Case Diagram for the game menu

Figure 2 shows a basic use case diagram for this system. It includes two actors: the player and the level editor. The most common user is the player who can start the game, manage their add-ons and manage their game options. Of these use cases, the most common and most complicated is the Start Game use case, as it contains the entire gameplay.

Sequence Diagrams

Starting a Game

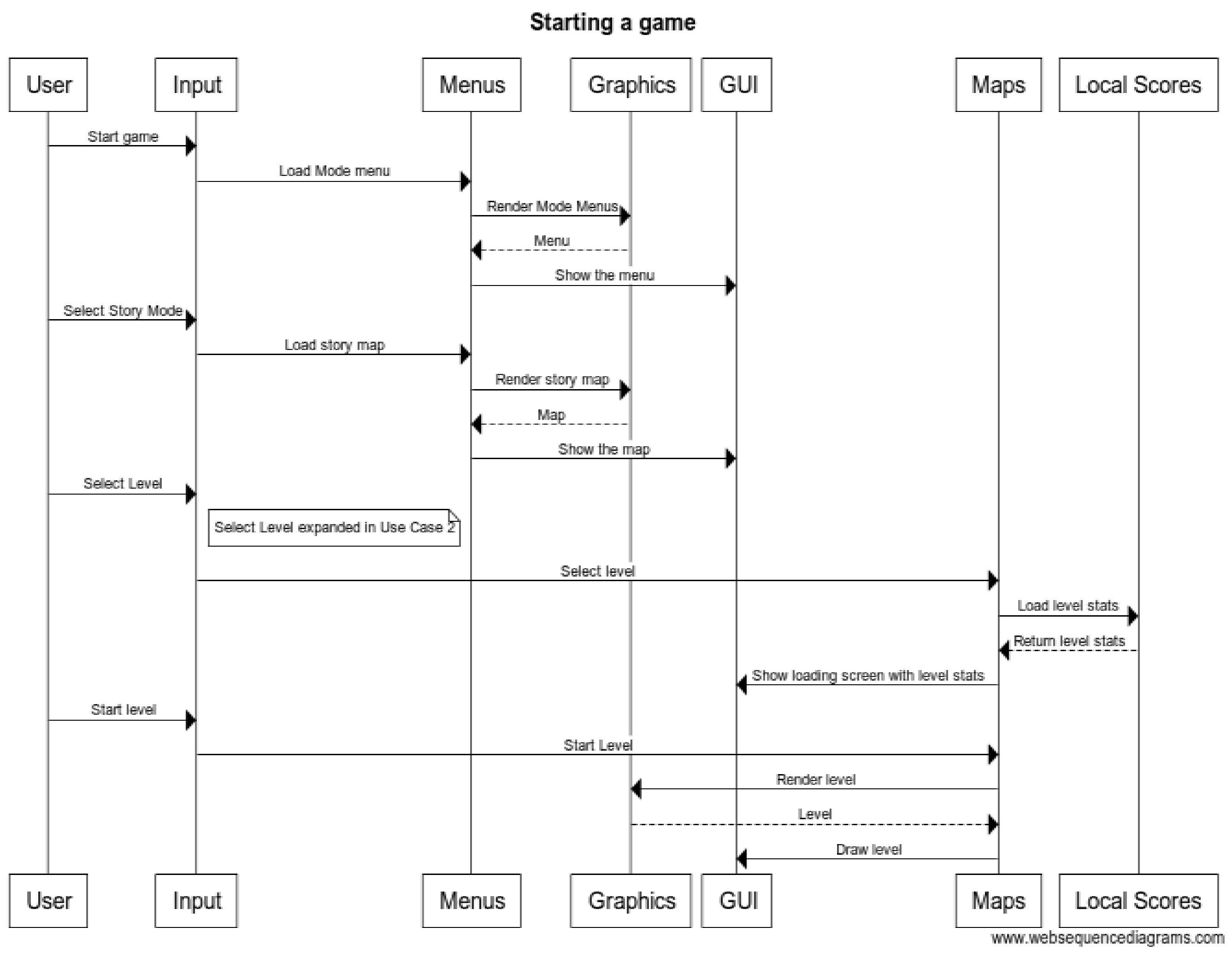

Figure 3: Sequence diagram for starting a game

Figure 3 shows a sequence diagram for a user starting a game. It describes the flow of a user choosing to start a game, selecting story mode, and selecting a level. The general elements are the Input, GUI, Menus, Graphics, Maps and Local Scores. Note that the Input actor shown in the diagram contains the Input and Input Handler subsystems from the high-level architecture to simplify the diagram. The general flow is that the Input receives input from the user, and passes it to either the Menus or Maps subsystem. These will then load any data they need from the Data Manager layer, get their contents rendered by the Graphics engine, and display them on screen.

Story Mode Map Selection

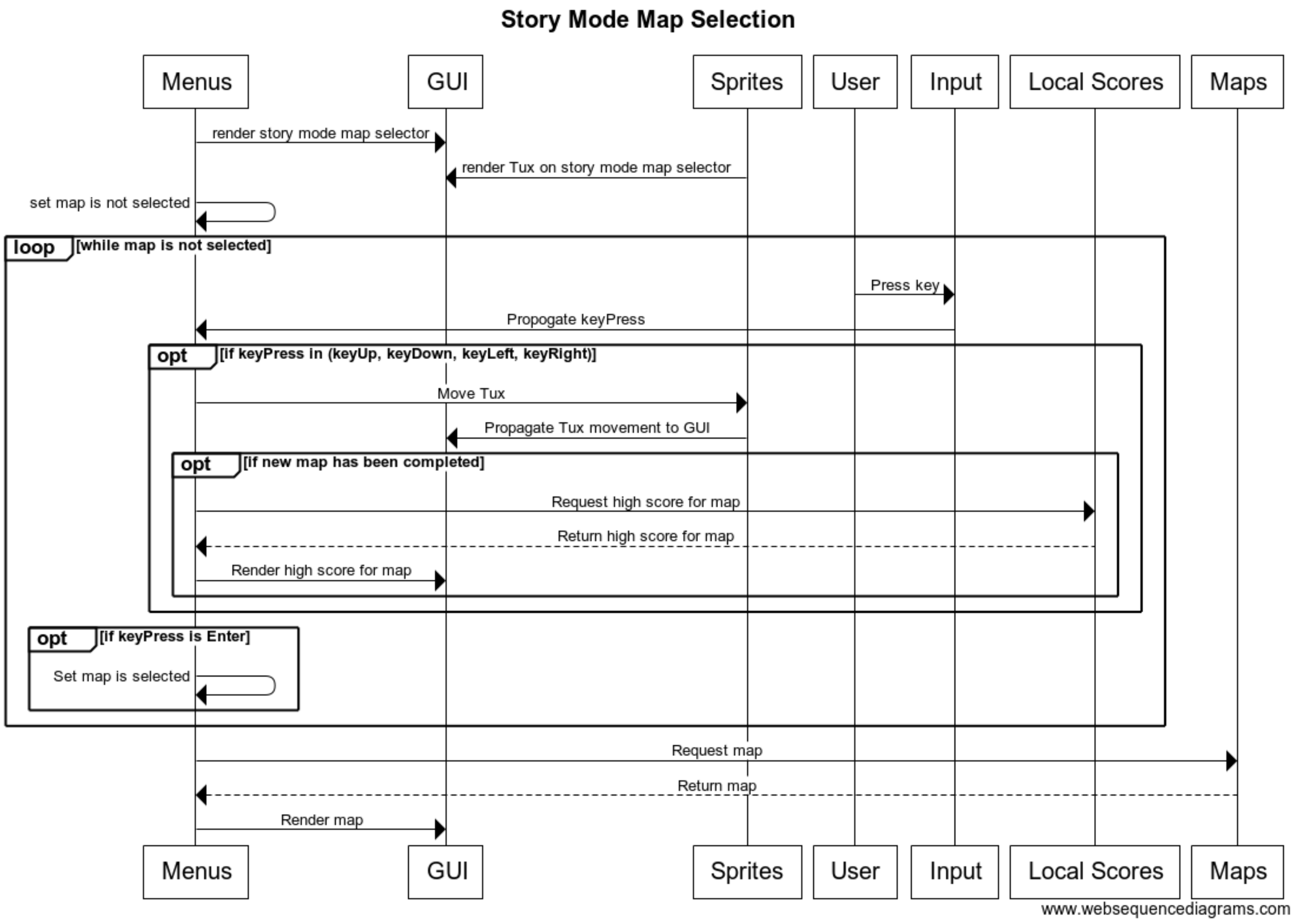
Story Mode Map Selection
Menus
GUI
Sprites
User
Input
Local Scores
Maps
render story mode map selector
render Tux on story mode map selector
set map is not selected
loop [while map is not selected]
Press key
Propogate keyPress
opt [if keyPress in (keyUp, keyDown, keyLeft, keyRight)]
Move Tux
Propagate Tux movement to GUI
opt [if new map has been completed]
Request high score for map
Return high score for map
Render high score for map
opt [if keyPress is Enter]
Set map is selected
Request map
Return map
Render map
www.websequencediagrams.com

Figure 4: Sequence diagram for story mode map selection

Error! Reference source not found. Figure 4 references the flow of interactions between the user and the SuperTux game system when going through the process of choosing a story mode level. After clicking the *Story Mode* option from the main menu, the story mode map selector menu is rendered to the GUI. Tux himself is also rendered to the GUI on a marker indicating a level.

At this point, the user can input keystrokes to interact with Tux and the map selector. To ensure that those interactions can happen until a map is selected, we must update the state for the story mode menu selector to reflect that a menu has not yet been selected; we are then able to loop while a map is not selected. In the loop, the input system waits on a keypress from the user. This keypress is then propagated from the input after being decoded. Once the menu receives the keypress, it has two options: move Tux or select a level.

To move Tux, the inputted key press must be in the set *{keyUp, keyRight, keyDown, keyLeft}*. The Menus direct the Sprites to update the position of Tux on the map selector, which propagates Tux's movement to the GUI. Once Tux has reached his new marker, the system determines if the new map has been completed before. If the new map has been completed, the Menus request the high score for this level from the Local Scores subsystem. Once the high score is received, the Menus render the high score for the map to the GUI.

The second scenario is much simpler. To select a menu, the user must press the *Enter* key. Once that keypress reaches Menus, we update the state of the story mode map selector to reflect that a map has been chosen. This causes the loop to exit.

Once the loop has exited, this means we have a level selected. The Menus subsystem requests the map from the Maps subsystems. Upon its receipt, Menus reflects the map selection in the GUI by rendering it.

Lessons Learned

Throughout the development of the conceptual architecture, a number of lessons were learned. First, we discovered that no one architecture is perfect, and an iterative method is necessary for determining the final architecture - in our case, a hybrid architecture. We learned to not look too deep into code and documentation when creating high level architectures, to keep ourselves focused on the overall structure. To learn more about subsystems, we learned that it is important to interact with the game as a user of the game to determine different aspects of the architecture. We also learned that high level architectures are much better described using visualizations like diagrams, rather than trying to describe them in words. Finally, we learned that all group members should understand the architecture in its entirety, to ensure that all members are on the same page when describing the architecture.