Concrete 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
Due Date: November 13th, 2017
TA: Dayi Lin
Instructor: Ahmed E. Hasan

Abstract

This report details our derived concrete architecture for SuperTux, an open-source multi-platform Super Mario-style game. It will also compare this architecture with our earlier conceptual architecture, and provide an updated version of the conceptual architecture. The concrete architecture was developed using a piece of software called Understand, which helped us trace and visualize dependencies between subsystems. The concrete architecture has a structure similar to the original conceptual architecture, but with many more dependencies. These dependencies shift the architectural style away from layered and towards object-oriented, though we still see remnants of the originally intended layered style.

The architecture was analyzed using reflexion analysis, which helped us identify differences between the two architectures. Two subsystems, Game Elements and Resource, were analyzed in more detail using Understand and our reflexion analysis. We also explore two sequence diagrams in order to better understand the low-level control and data flows in the system, and detail the lessons that were learned during this analysis.

Derivation Process

The derivation process for the concrete architecture of SuperTux began by updating our conceptual architecture. We compared our old conceptual architecture to the new one and created a basic five level layered system: Interaction Layer, Input Handler, Resource Manager, Game Elements, Add-on Manager. After this, we started to derive our concrete architecture by following these basic five subsystems using the Understand tool.

An effort was made to keep most files that were grouped together in the source code by the developers, together within our layers. We mapped each directory into the layers we thought it matched best, based on our conceptual architecture. We then looked at the internal dependencies of each layer. If any unexpected dependencies were found we would examine the dependency and determine if either our architecture diagram needed to be updated or if the file need to be relocated to a more suitable location within the architecture. This process was repeated until we came up with an architecture that was most logical with the existing dependencies.

Concrete Architecture

Upon analyzing the SuperTux source code folder and all its component folders, the team was able to derive a concrete architecture for the game by generating a dependency map for the source code via the Understand Software. Understand is a customizable integrated development environment that enables static code analysis through an array of visuals, documentation, and metric tools [1]. The group distributed the component folders amongst 5 subsystems directly related to the functionality of the source code within the components.

These subsystems include the Interaction Layer, Input Handler, Resource Manager, Game Elements, and Addon Manager. The generated dependencies for the concrete architecture by Understand can be seen in Figure 1. The high level concrete architecture diagram can be seen in Figure 2.

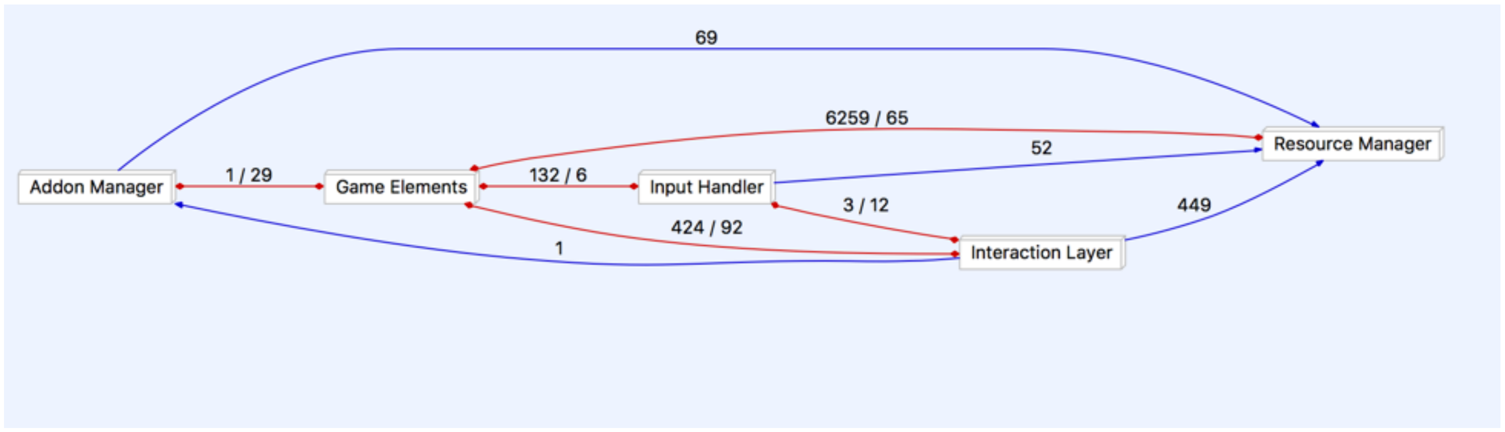


Figure 1 Understand Concrete Architecture Subsystem Dependencies

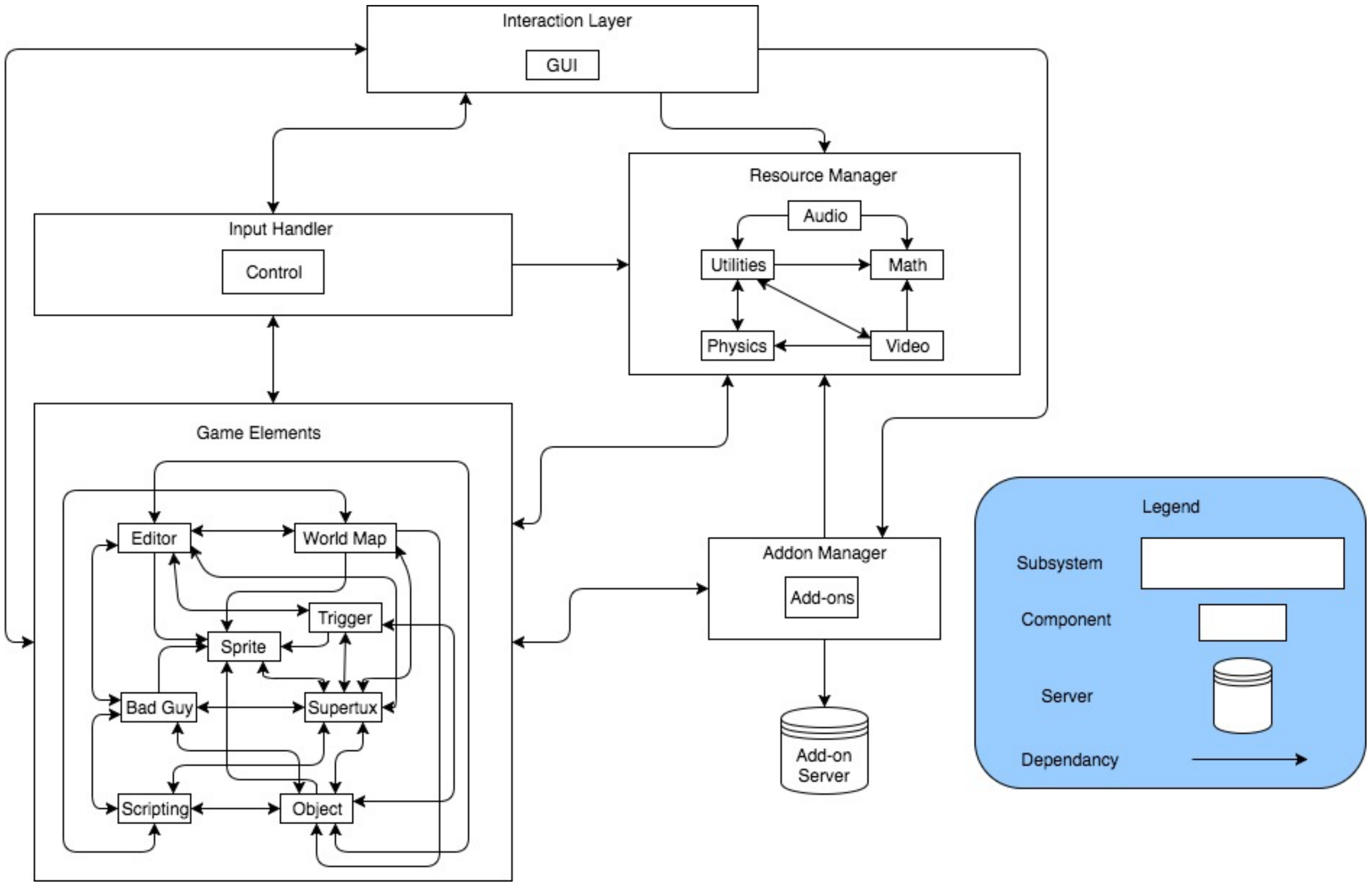


Figure 2 High Level Concrete Architecture Diagram

It can be seen in Figure 2 that the architecture is much more loosely layered than originally anticipated, and incorporates more object-oriented architecture style qualities due to the majority of its subsystem and component dependencies being bi-directional. This interdependence within the system, along with multiple hierarchal unidirectional dependencies still existing in the architecture, would define the architecture to be a loosely-layer object-oriented hybrid architecture [2].

The Interaction Layer subsystem includes a single GUI component, and handles displaying, generating, and handling interactions with GUI related objects such as menus or buttons. The Input Handler subsystem includes a single Control component, which handles decoding keyboard strokes, as well as decoding controller invocations should the player be using a controller or joystick. The Game Element subsystem includes the Editor, World Map, Sprite, Trigger, Bad Guy, Supertux, Scripting, and Object components, and handles the majority of the in-game functions such as controlling SupterTux, loading maps, initiating sprites on maps, loading user configurations and more (detailed explanation in subsystem explanation section of report).The Addon Manager subsystem has a single component called Addons, and handles accessing and downloading add-ons from the SuperTux Add-on Server via network calls. It provides these addons as a service to the game. The Resource Manager subsystem provides resources for playing audio files in-game, defining physics rules, supporting graphics generation (such as shapes), and calculating any required mathematical functions required in-game.

Subsystems

Interaction Layer

The Interaction layer is a simple subsystem that handles drawing things to the screen. It only has one component - the gui. This component seems like it should handle drawing everything to the screen, but it doesn't. In another example of poor structure by the development team, most of the screen rendering is spread between other subsystems such as the resource manager and supertux.

Input Handler

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

The input is given from the Interaction Layer. This raw input is decoded via the Control component. This component is specifically built to handle hardware interactions from a keyboard (or other input device) where the inputs from the keys would be mapped to their specific values. These values are then passed along to the corresponding game subsystem, based on the context of the interaction.

Game Elements

The Game Elements subsystem is by far the most complicated and important subsystem in the architecture. The role of the Game Elements subsystem contains almost all functionality of SuperTux. This can be seen by the vast number of components within the subsystem, including the Editor, World Map, Sprite, Trigger, Bad Guy, Supertux, Scripting, and Object components. The names of these components describe their functionalities, which range from the in-game maps, the sprites and characters in game, map editing, scripting, and more. The Game Elements subsystem is co-dependent on every single other subsystem in the architecture, including the Interaction Layer, Input Handler, Resource Manager, and Addon Manager.

Add-On Manager

The Add-on Manager subsystem is one of the multiple very simple subsystems with only one component within it. The role of the Add-on Manager is to store downloaded add-ons, communicate with the Add-on Server, and facilitate downloads from the server. Add-on Manager depends on the Resource Manager layer and the Game Elements layer, and is depended on by the Interaction layer and again the Game Elements layer.

Resource Manager

The Resource Manager subsystem is responsible for handling updates the Game Elements layer that cause changes to or require logic from the Audio, Physics, Math, Video, and Utilities components. These components are responsible for things like sound effects, music, calculating Game Element trajectories based on speed and actions, and updating the position of Game Elements.

While Game Elements is responsible for initializing all of the components that enable you to play SuperTux, the Resource Manager is in charge of updating the Game Elements to make gameplay possible. In other words, Game Elements are the things in the game, while the Resource Manager components are the actions being performed by or on the things.

Styles and Design Patterns

In our updated conceptual architecture for SuperTux, you can see that the one-way dependencies indicate a layered architecture; although, the dependencies within the subsystem show characteristics of an object-oriented approach.

Our concrete architecture is similar to our conceptual architecture in that it has a hybrid architecture with elements of both layered and object-oriented architecture styles. When looking at the concrete architecture diagram, you can see the large number of two way dependencies, which is an indicator of an object-oriented architecture. When looking at the organization of the subsystems, it can be seen that there was an attempt to follow a layered approach in that inputs come from the Interaction Layer, are handled by the Input Handler, then manipulate the other subsystems.

The advantages of a layered architecture in SuperTux are system design, enhancement, and code reuse. The layering provides abstraction for easier understanding of components. It can be seen that as development progressed, the abstraction has lessened and tighter coupling has crept in which has led to some of the unexpected dependencies that we discovered in our reflexion analysis.

The strong presence of object-oriented architecture has several benefits. This architecture creates highly cohesive layers that with many reusable components within them. The object-oriented also style helps with the maintenance of the system. As there are many different developers who have contributed and continue to contribute to SuperTux, the object-oriented manner of the architecture allows for interfaces which decrease difficulty when contributing changes without requiring a complete modification of a subsystem; however, developers must still be mindful of the coupling between components in the subsystems as any alterations to the components could cause unintended side-effects on other components.

Concrete and Conceptual Architecture Comparison

Modified Conceptual Architecture

The modified conceptual architecture for this group can be seen below in Figure 3. It was changed to more closely resemble the concrete architecture. Some notable changes between the old and new architecture include renaming the subsystem Data Manager to Addon Manager, as well as renaming multiple components in the Game Elements and Resource Manager. The old conceptual architecture can be seen in the Appendix (Figure 10).

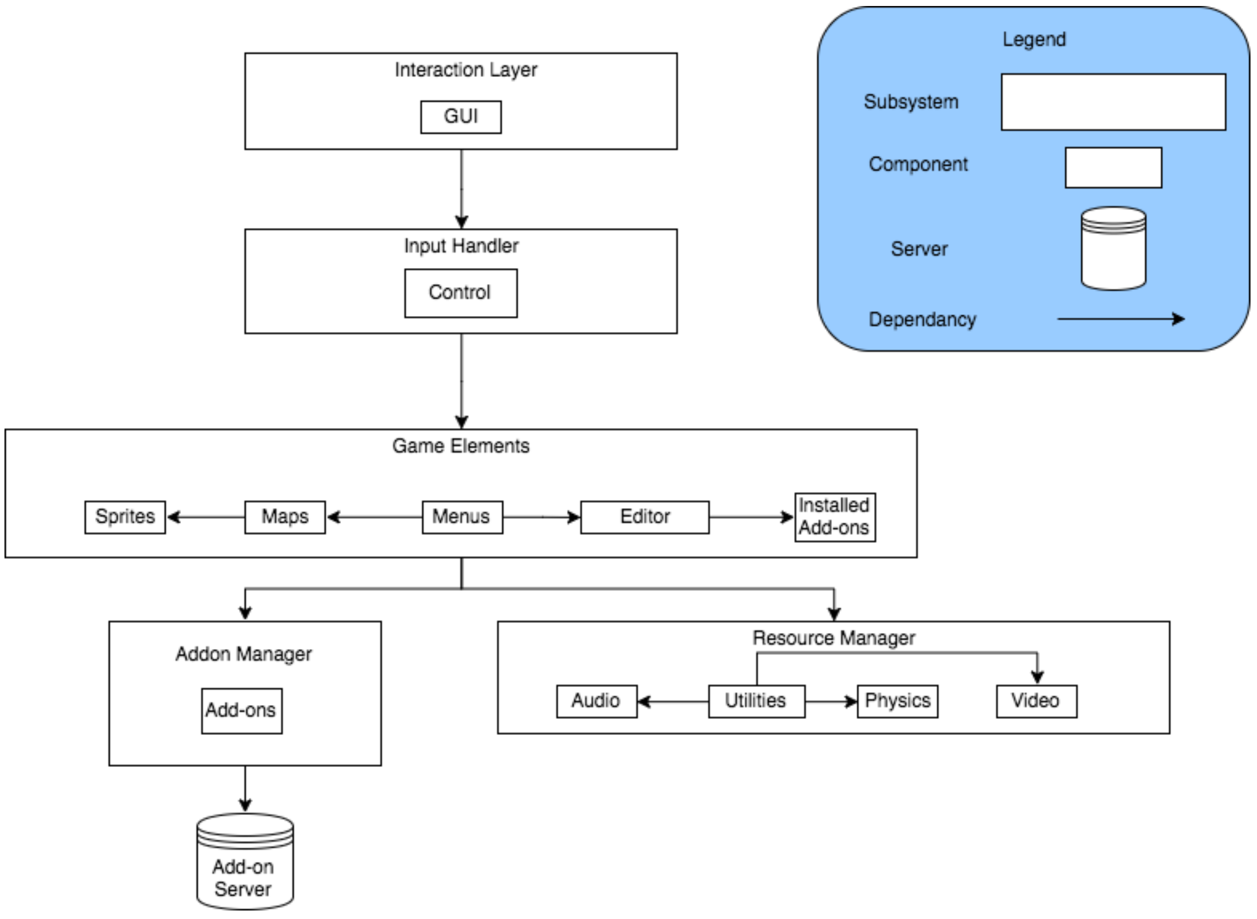


Figure 3 Modified Conceptual Architecture

Low-Level Subsystem Comparison

The Reflexion Analysis is used to measure and discuss the differences between the conceptual architecture and concrete architecture. This allows us to discover the discrepancies in dependencies between the two architectures, and investigate the reasoning for those differences. We will dive into two subsystems and analyze the differences between the conceptual and concrete architectures, as well as the divergences that exist between them.

Resource Manager

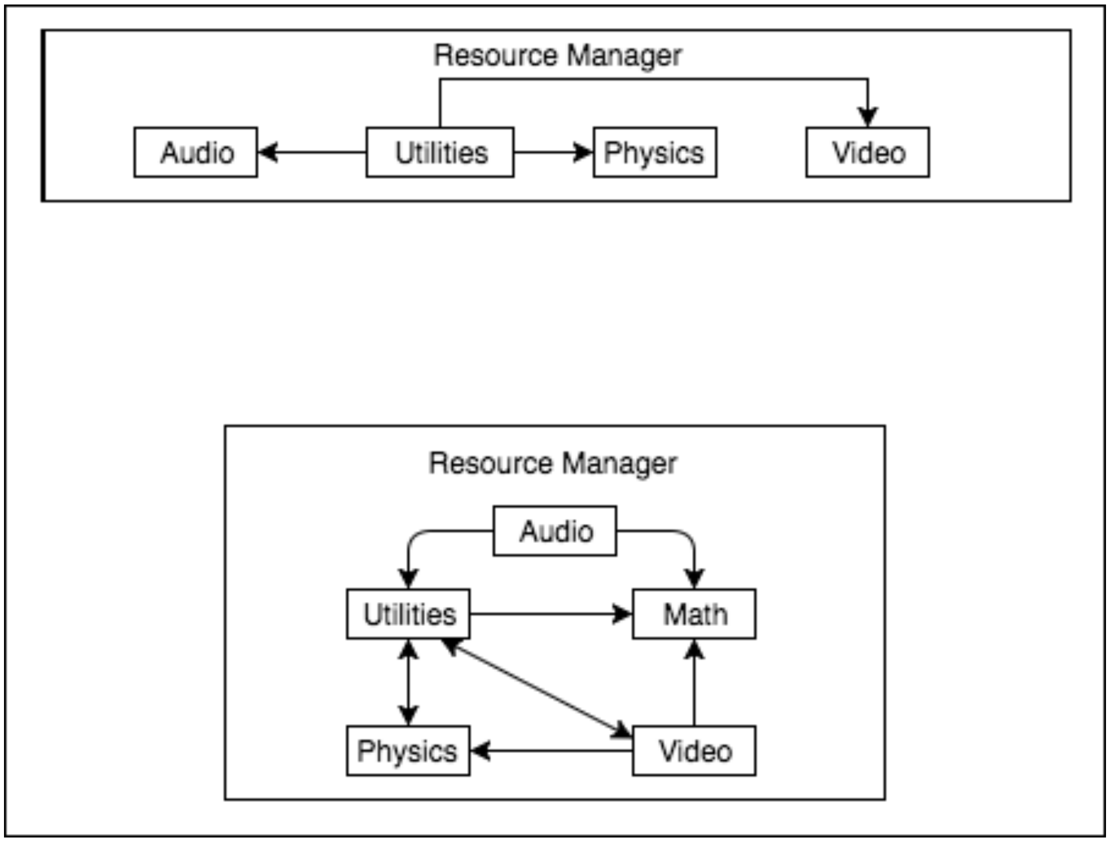


Figure 4 Resource Manager Differences

There were multiple differences between our conceptual and concrete architecture for the Resource Manager layer. In the concrete there are five objects, whereas our conceptual only had four. We originally had an Animation component, which turned out not to be there, and we missed the Utilities and Math components. The concrete architecture had much higher coupling than we were anticipating, which includes many bi-directional dependencies. In our conceptual diagram, there were strictly uni-directional dependencies. There are several unique dependencies to take note of in the concrete diagram. Every component depends on Utilities, and similarly Utilities depends on almost every other component, thus making it the core of the Resource Manager subsystem. The Math component has no dependencies, yet three other components depend on it, also making it take an important role in the subsystem. Finally, unintuitively the Physics and Math components have no dependency between them. Within resource manager, Audio depends on Math to invoke its vector methods, in order to process the sound_manager procedure. It's not clear why a mathematical vector would be required in order to support the sound_manager. Our conceptual architecture did not have this dependency, as it is not something that is intuitive in the architecture. Strange dependencies like these are the reason for most of the divergences between the conceptual and concrete architectures.

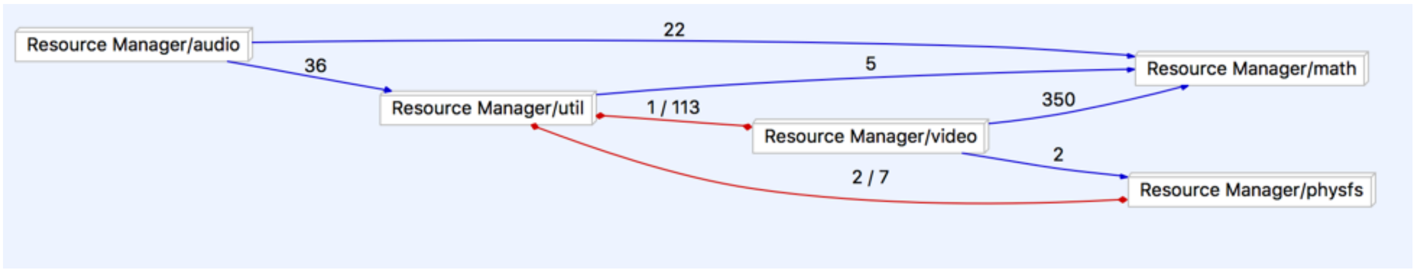
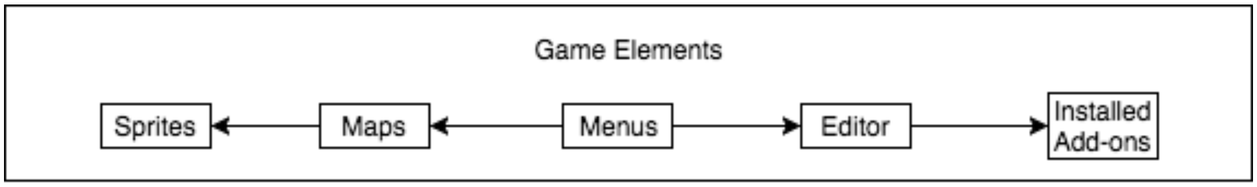


Figure 5 Understand Dependency Graph for Resource Manager

Game Elements



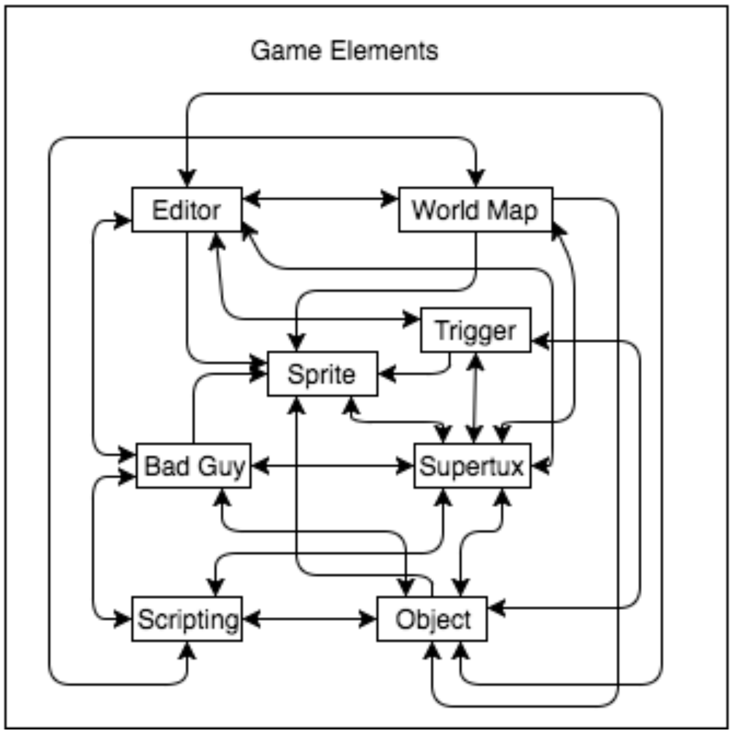


Figure 6 Game Element Subsystem Differences

With a quick glance at the diagram with our conceptual and concrete architecture for the Game Elements subsystem one can easily see the two are considerably different. The only component that is in both architectures is the Sprite component, yet even then we did not expect the dependencies that showed. We hypothesized that only the Maps component would depend on Sprites, instead there are a total of six components that depend on Sprites and it depends on the Supertux component. The Supertux component is very similar to a God class, as it does so much (it has seven bi-directional dependencies!). The Maps component, which as it is in our conceptual architecture is used to store each individual game level, is actually stored within the Supertux component. There are multiple other components in this subsystem that we found to be confusingly named. The Trigger, Object, and Scripting components are not named in such a way as to make their purpose clear. Specifically, upon closer inspection inside the Scripting component we found that it seems to be a place that random functionality was put, creating unforeseen dependencies between it and other components. An unexpected dependency here can be seen between Scripting and Object, caused by strange organization of code. Within Object, there is level_time which represents the user's time to complete a level. The level_time object includes the squirrel_util file, which is within the Scripting object. This squirrel_util file handles storing and persisting user values, such as their level_time for respective maps. Again, our conceptual architecture did not have this dependency, as it is not something that is intuitive in the architecture. There are many dependencies that are as unintuitive as the above example, and are the reason for most of the divergences between the conceptual and concrete architectures.

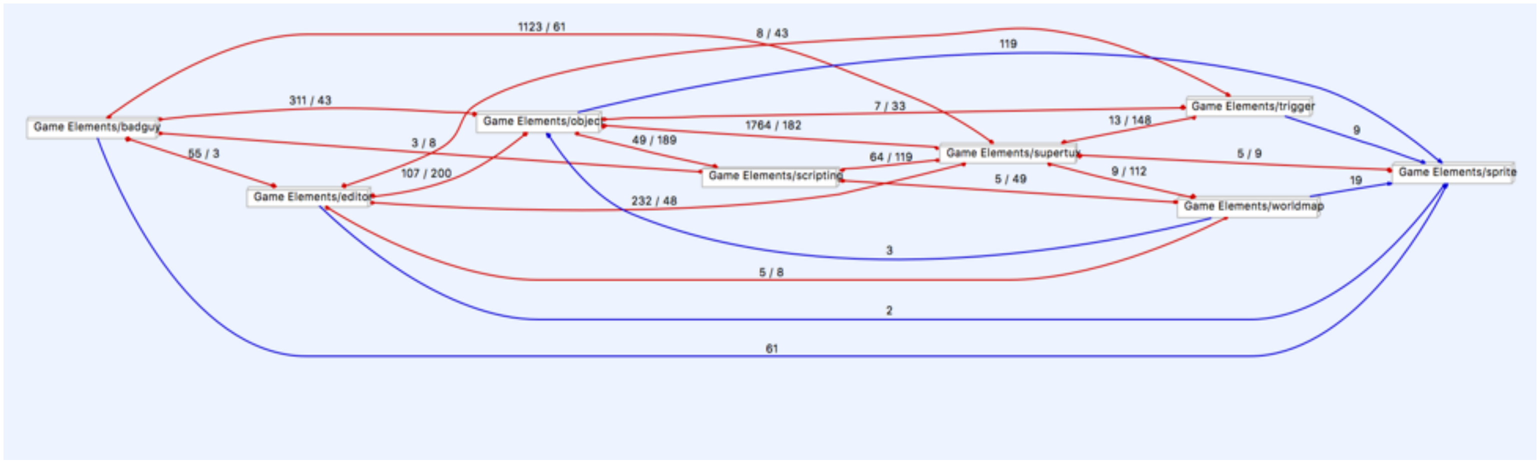


Figure 7 Understand Dependency Graph for Game Elements Components

Sequence Diagram

To gain insight into the concrete architecture, we traced its the execution through two use cases: story-mode map selection and exiting a level. We used the SuperTux source code [3], exploring the different classes to find how data and control flowed throughout the scenarios. Each use case is shown below as a sequence diagram accompanied by an explanation.

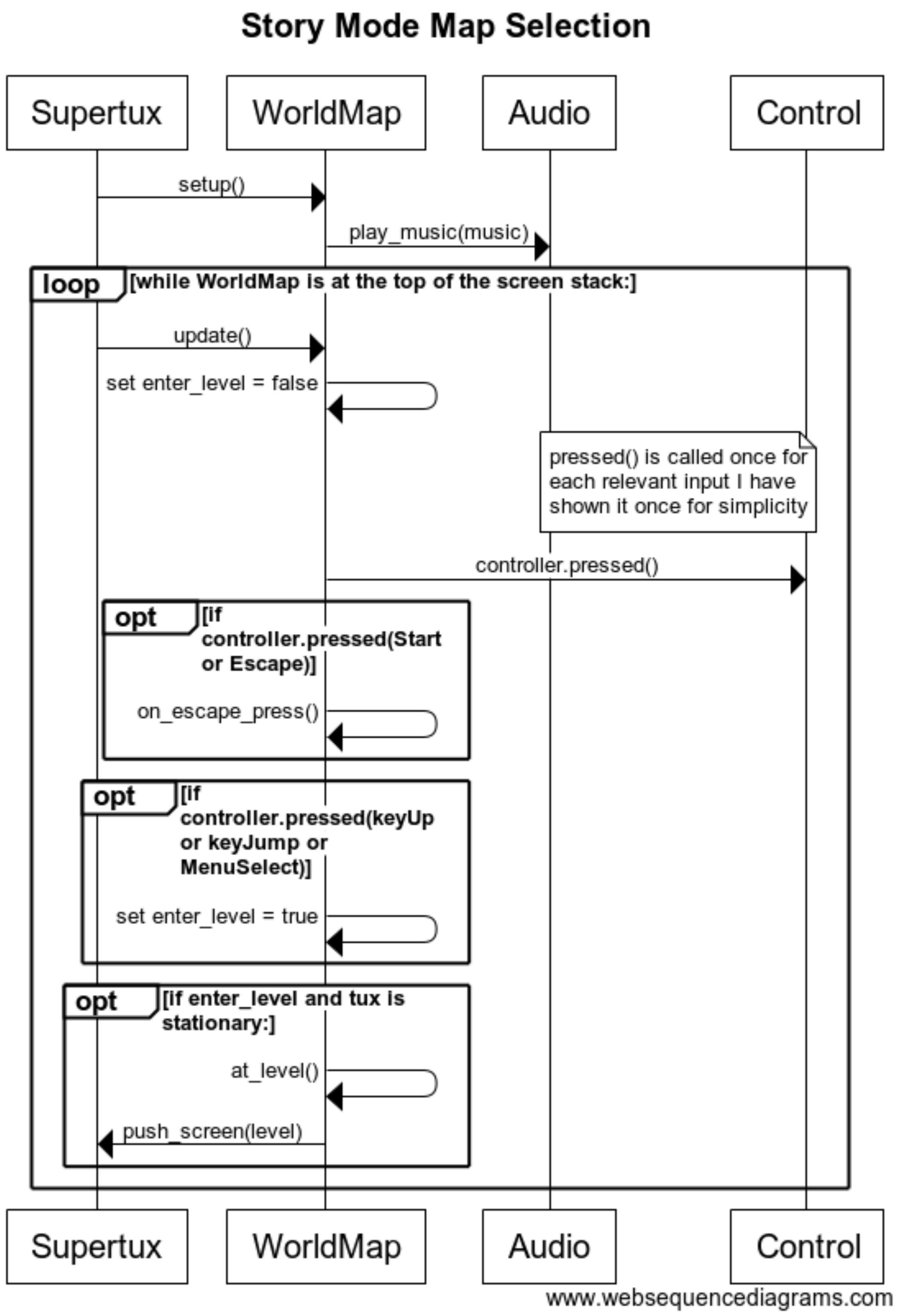

Story Mode Map Selection
Supertux
WorldMap
Audio
Control
setup()
play_music(music)
loop [while WorldMap is at the top of the screen stack:]
update()
set enter_level = false
pressed() is called once for each relevant input I have shown it once for simplicity
controller.pressed()
opt [if controller.pressed(Start or Escape)]
on_escape_press()
opt [if controller.pressed(keyUp or keyJump or MenuSelect)]
set enter_level = true
opt [if enter_level and tux is stationary:]
at_level()
push_screen(level)
Supertux
WorldMap
Audio
Control
www.websequencediagrams.com

Figure 8 Sequence Diagram for selecting a level on World Map

Figure 8 shows a sequence diagram for selecting a level on a world map. It mostly involves the Supertux and Worldmap components, with Audio being used to play the map's music. The flow of control in Supertux is a bit strange, so I have chosen to focus on the parts relevant to this scenario. Within the Supertux component is a class called the ScreenManager, which keeps track of which screen the game is on. It also runs the game loop, which we see above. The actual condition of the loop is that there is still a screen to show, but this scenario will only continue as long as the WorldMap is the current screen. Thus, the push_screen() call from WorldMap to Supertux will change the top screen to the selected level, ending the scenario.

Another thing to point out is that there is no control here for moving Tux around the map. This system mostly has components handle their own inputs, so WorldMap maintains a Tux object which manages its own direction and position on the map.

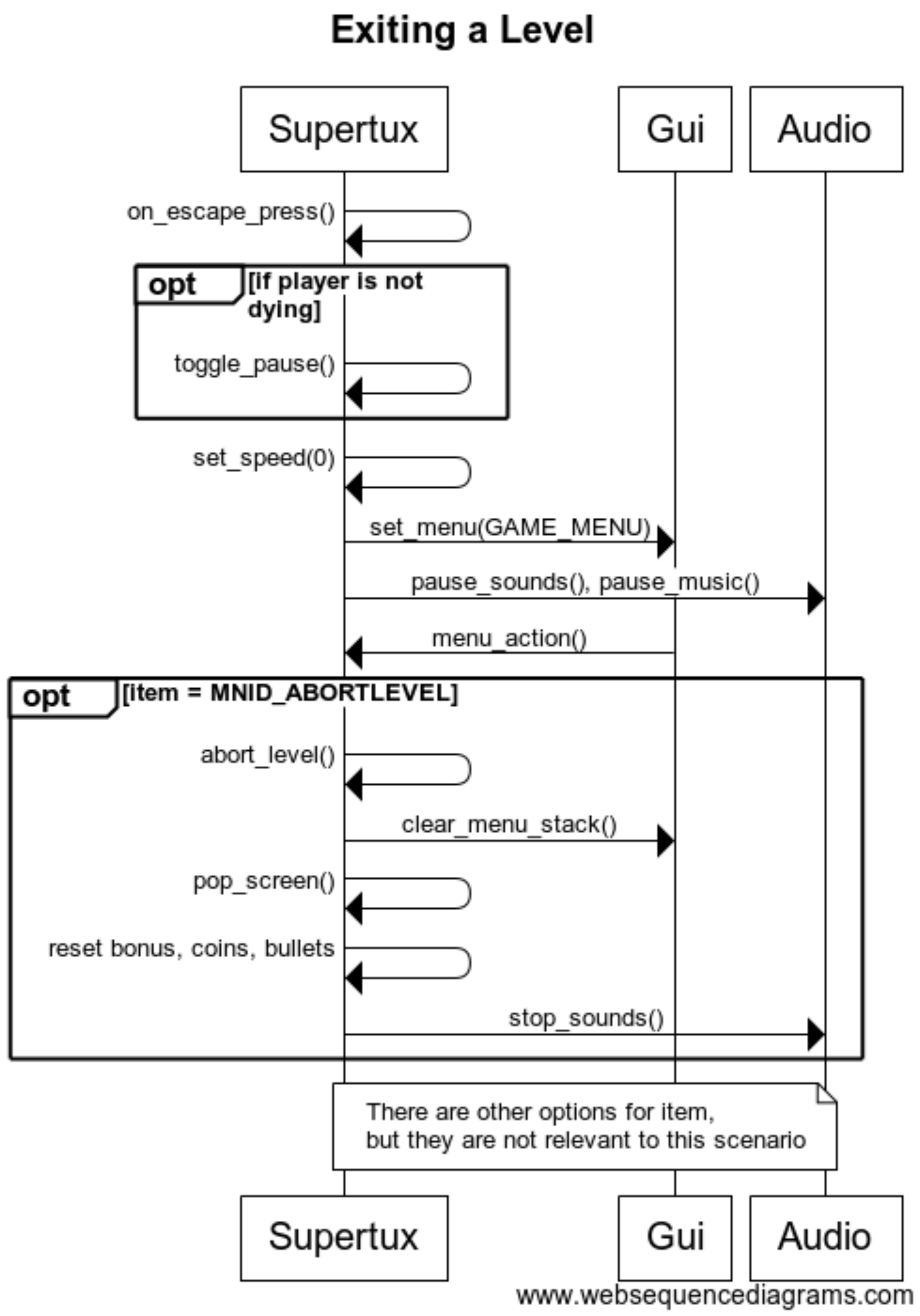


Figure 9 Sequence Diagram for user exiting level

Figure 9 shows the sequence of steps when a user exits a level. It starts when the GameSession recognizes an escape input. It then checks that the player is not dying, because if they are it won’t let them pause and avoid the death. It then toggles the pause state, which stops the screen from running, stops audio, and pushes a game_menu.

Here we see another major architectural issue. The Gui component has a class called MenuManager which tracks the MenuStack and handles menu inputs. However, the different kinds of menus live within Supertux, so control is passed to the Gui then right back to Supertux. If Supertux gets a menu_action that tells it to abort the level it clears the menu, pops the current level from the screen stack, resets local variables for bonus points, coins etc, and stops the music. Control is then returned to whatever screen was below the level in the stack, which is probably the WorldMap.

Concurrency

SuperTux is multi-threaded. While the game is running, processes such as audio and graphics are executed concurrently. These processes need to be processed at the same time to avoid audio and visual lags. Physics also acts in parallel with the SuperTux object. For example, the physics needs to know how fast SuperTux is moving to be able to calculate the location of where Supertux will land when it jumps.

Lessons Learned

Throughout the development of the concrete architecture, several lessons were learned. First, the concrete architecture had much higher coupling than we would have expected, especially compared to our conceptual architecture. We also learned that software design requires some compromises and planning. The high coupling associated with SuperTux was likely caused by poor planning. Within our group, we learned that it is important to state the architecture beforehand, to ensure all group members are on the same page. Using Understand, we discovered that it is a very powerful tool to understand the code base, but it has a steep learning curve. Finally, we determined that it is hard to derive an architecture by simply observing how a game runs without looking at source code.

References

[1] SciTools, "Understand," [Online]. Available: https://scitools.com/. [Accessed November 2017].

[2] A. E. Hassan, "Software Architecture: Intro and Styles," October 2017. [Online]. Available: http://cs.queensu.ca/~ahmed/home/teaching/CISC326/F17/slides/CISC326_04_ArchitectureStyles.pdf. [Accessed November 2017].

[3] SuperTux Development Team, "SuperTux Github Source Code," [Online]. Available: https://github.com/SuperTux/supertux. [Accessed November 2017].

Appendix

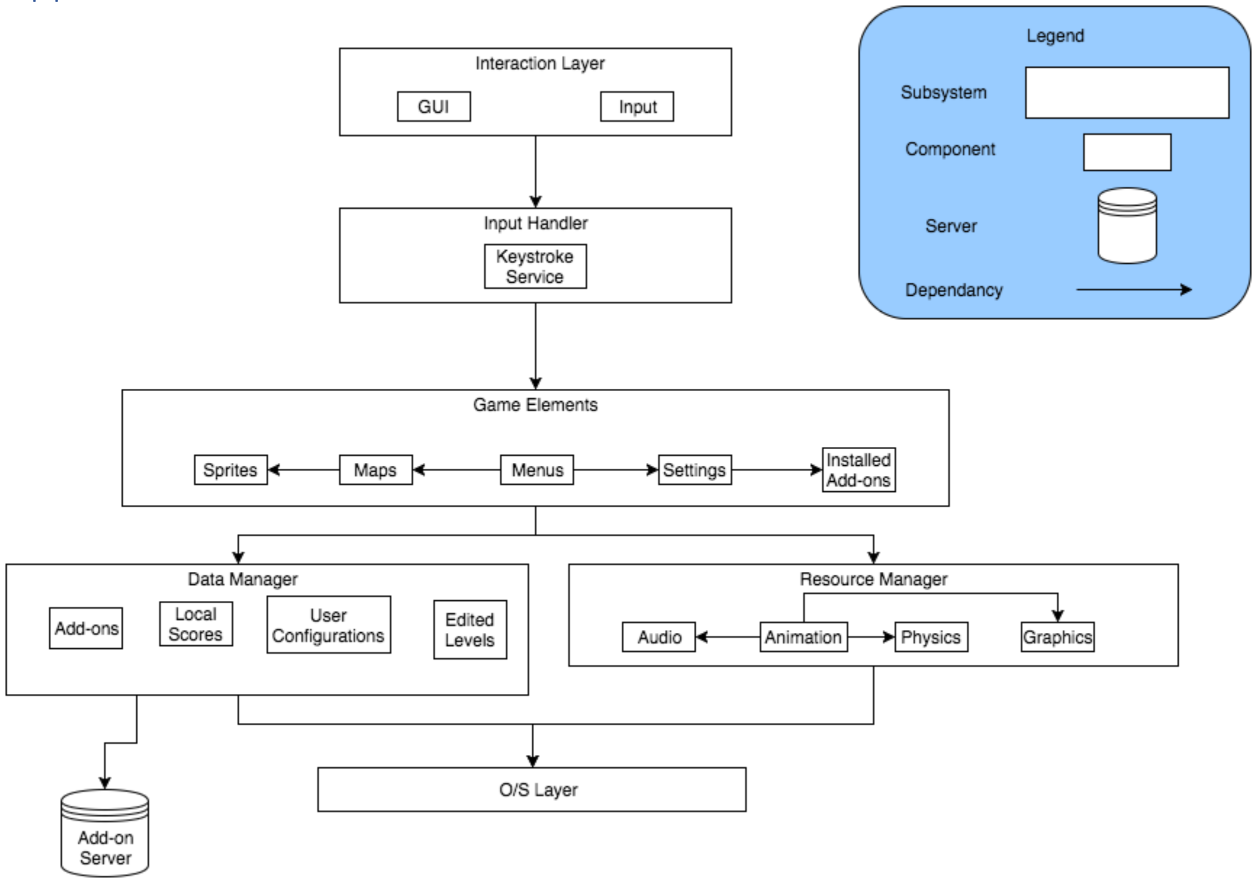


Figure 10 Old Conceptual Architecture