Enhancement Proposal 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: December 5, 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.

In this report, we will be discussing and analyzing the feasibility of adding two-player local multiplayer to SuperTux. We will do this by explaining this enhancement, and performing a SAAM analysis for two different methods of implementing it. The two methods involve either enhancing the SuperTux component within Game Elements, or creating an entirely new Multiplayer subsystem. In the end, we chose that enhancing the SuperTux component would be preferable, and the reasoning will be discussed in detail. We will then show how we would change the architecture itself, and the effects of these changes. The architecture styles and design patterns will be examined, and finally the potential risks that exist from enhancing SuperTux will be discussed.

Proposed Feature and Motivation

Currently SuperTux is strictly a single player game, but by adding 2-player multiplayer we add in the opportunity for the player to be able to share the game with a friend, expanding the SuperTux experience to include social aspects. Moreover, there is the added benefit for some less experienced players to get a helping hand with the tougher levels that they are unable to complete alone. With our proposed integration approach, we can easily scale the number of players, offering the potential for some other game modes if we see success with 2-player mode. One argument against adding in 2-player multiplayer is that the game was created and designed for one player. We understand this and are still giving the player the option to play alone with no changes to the gameplay they know and love.

Enhancement Architectures

In implementing local two-player multiplayer for SuperTux, two enhancements of the conceptual architecture are presented below.

Implementation 1 – Abstract "SuperTux" Design Pattern

Implementation 1 would manage two-player local multiplayer through enhancing the SuperTux component within Game Elements, as seen in Figure 1, to implement an Abstract "SuperTux" Factory Design Pattern. It would involve modifying the GUI component within the Interaction Layer subsystem to add a "Local Multiplayer" option to the menu elements. Furthermore, the Control component within the Input Handler subsystem would be modified to be able to handle another set of input controls that are decoded and mapped to a newly instantiated SuperTux object. The SuperTux component within Game Elements implements an Abstract Factory Design Pattern in order to create families of related Tux objects without specifying their concrete classes. All the changed components within the architecture are highlighted as red in Figure 1.

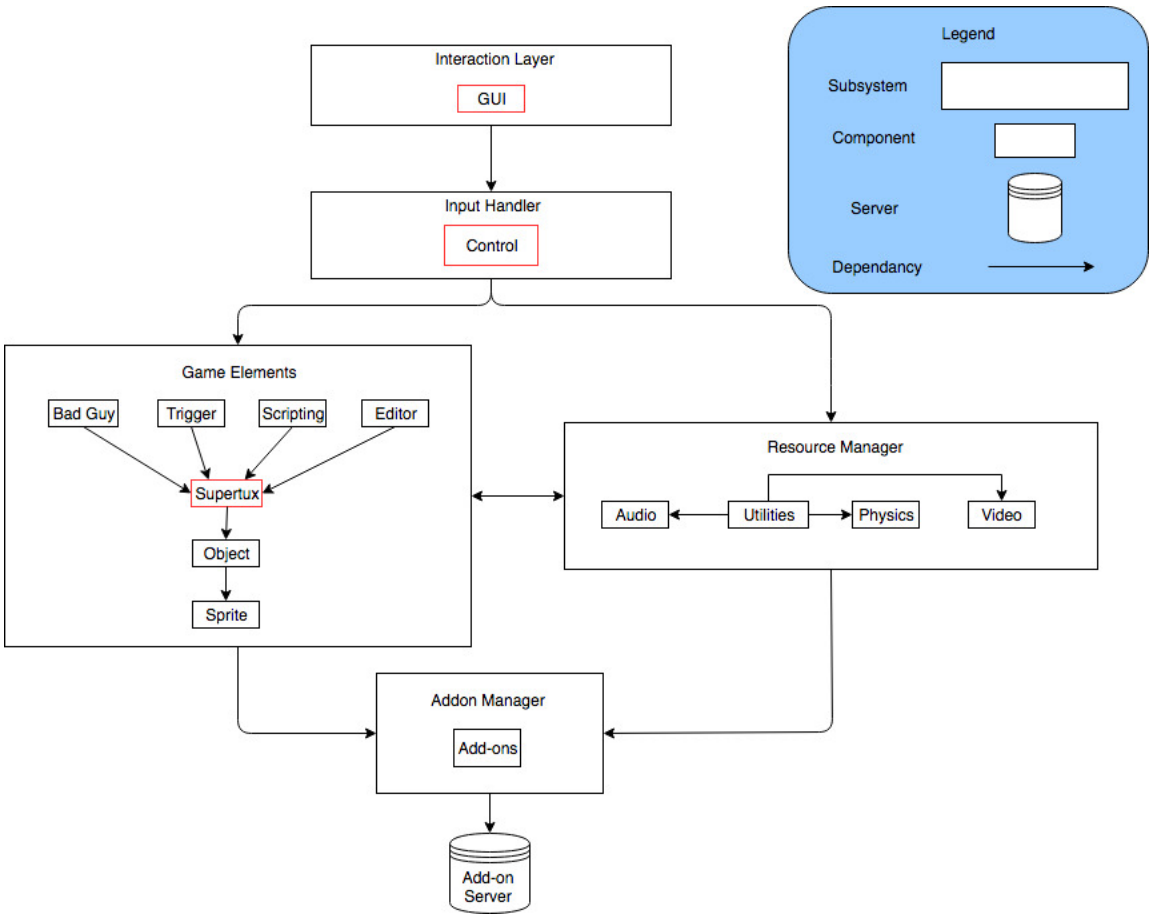

Figure 1 Enhanced Conceptual Architecture for Implementation 1

Implementation 2 – Multiplayer Manager Subsystem

Implementation 2 would manage two-player local multiplayer through the implementation of a new Multiplayer Manager subsystem, as seen in Figure 2. For this implementation, the GUI component would be modified within the Interaction Layer subsystem to add a “Local Multiplayer” option to the menu elements just like implementation 1. The Control component in Input Handler subsystem would be modified to be able to handle 1 extra set of input controls that are decoded and mapped to the 2 respective player components within the new Multiplayer Manager subsystem. The Multiplayer Manager subsystem intercepts user input from the Input Handler, and tracks the behavior of each player’s SuperTux. Thus, the SuperTux component in Game Elements would be modified to be able to handle 2 SuperTux instances on the map, reference the Multiplayer subsystem, and poll it in order to update the two SuperTux instances in the game.

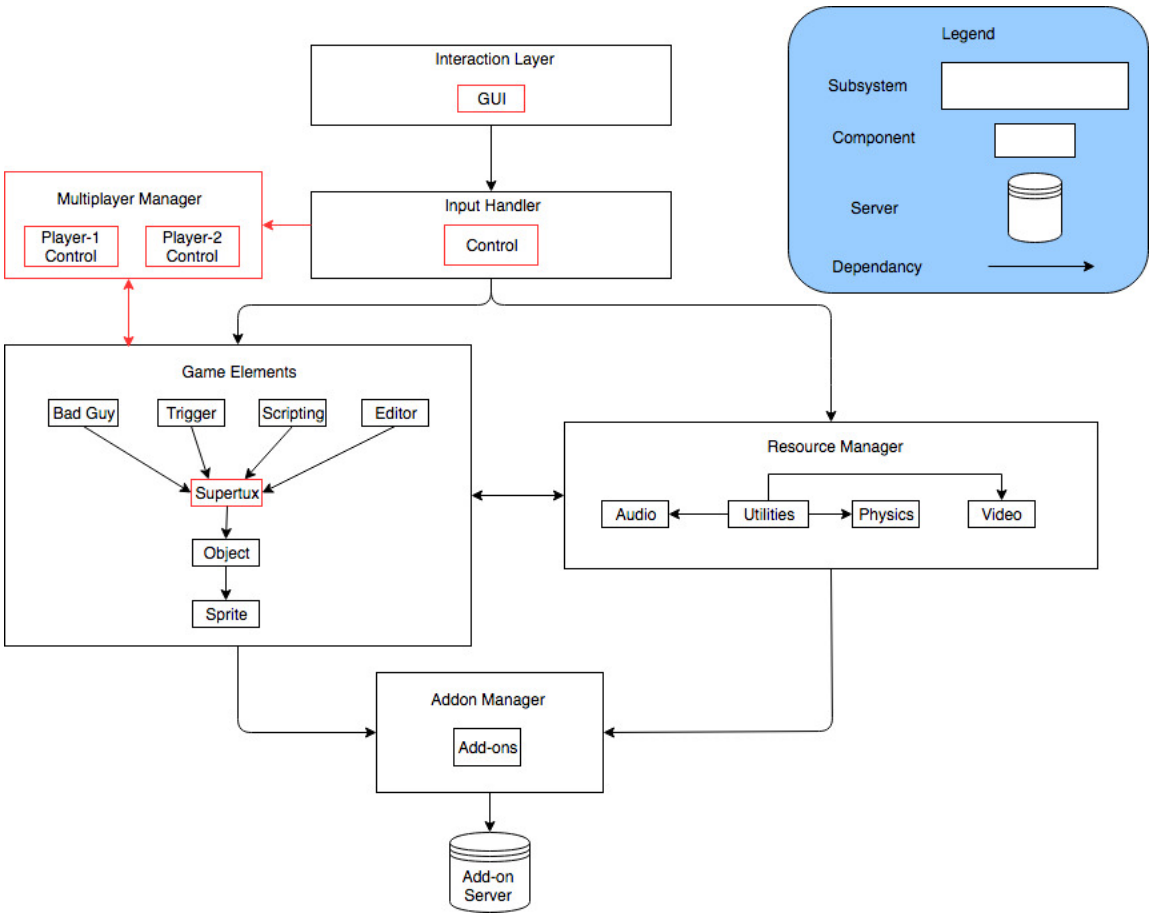

Figure 2 Enhanced Conceptual Architecture for Implementation 2

The major differences in interactions between implementation 1 and 2, are the fact that implementation 1 only modifies existing components in our conceptual architecture, utilizing the already existing dependencies between the subsystems to operate. On the other hand, implementation 2 requires the addition of a new subsystem, as well as 2 new dependency additions to the subsystems Game Elements and Input Handler. For both implementations, the architecture style is not significantly changed, and maintains a hierarchal structure as well as object-oriented dependencies and interactions. Thus, it's still considered a hybrid Layered & Object-Oriented architecture design.

The group decided that implementation 1 is the optimal approach in implementing the enhancement, as it would help improve maintainability, as well as the cohesion and coupling of the entire system. This is further discussed in the SAAM analysis of this report.

Architectural Styles and Design Patterns

Our proposed feature would require us to enhance the Game Elements subsystem and add a SuperTux factory. To do this we would utilize the Abstract Factory design pattern. This design pattern is very useful because it provides an interface for creating families of related or

dependent objects without specifying their concrete classes. This is ideal for creating more than one SuperTux character on the map.

Implementing this design pattern would not affect our architectural style and it would remain layered/objected-oriented. Furthermore, the current implementation of SuperTux utilizes the Abstract Factory pattern in a file titled object_factory.cpp. This shows that this design pattern can be easily implemented within the system. It also allows the team creating the multiplayer feature to base their work of the working abstract factory implemented within SuperTux.

SAAM Analysis

Table 1: SAAM Analysis for both implementations

Stakeholder (NFR)	**SuperTux Factory**	**Multiplayer Subsystem**
Developer (*Scalability*)	*High*. With an abstract SuperTux factory, scaling the multiplayer capacity is as easy as calling the SuperTux factory. This means that we can scale up the multiplayer capabilities to as many players as the host computer can handle. The only real limitation of the scalability of this implementation is the capabilities of the hardware on the PC running SuperTux.	*Low*. With a dedicated Multiplayer Subsystem, there is a component for each player. This means that scaling the system means creating a new multiplayer component for the new player. This adds a lot of overhead and increases the difficulty of scaling the system.
Developer (*Evolvability*)	*High*. Changes to the SuperTux factory affect all SuperTux instances. Because SuperTux instances are created from the factory, new versions of the game would use the updated SuperTux factory meaning that all players will have the same changes from the update.	*Low*. Changes to each player need to be implemented individually. In other words, changes to player one need to be separately implemented for player two. Assuming consistent time to implement features for each player, this approach increases development time proportional to the number of players supported (i.e. 2 players, 2x dev time).
Player (*Usability*)	*High*. Multiplayer is enabled and can be played by two users. The main goal of this feature is to allow two players to play on the screen at the same time. This approach accomplishes that.	*High*. Multiplayer is enabled and can be played by two users. The main goal of this feature is to allow two players to play on the screen at the same time. This approach accomplishes that.

Effects of Enhancement on the System

The chosen abstract SuperTux factory implementation will affect the maintainability, evolvability, testability, and performance of the system. The maintainability of the system is arguably decreased with the implementation of multiplayer. Anytime code is added to the codebase (and none is removed) it can be said that maintainability is decreased. When looking at maintainability from a developer's perspective, it is important to weigh the value added by the feature vs the time cost of maintaining it.

As discussed in the SAAM Analysis, the SuperTux factory allows all changes to the SuperTux factory to be reflected in all instances of SuperTux that it derives. This means that changes to any player are reflected for all players in multiplayer, meaning this implementation allows for high levels of evolvability.

To be able to properly test this implementation of multiplayer, we will utilize unit testing at the code level, integration testing at the subsystem level, system level testing for ensuring that all subsystems interact properly, and regression testing to avoid re-implementing previously encountered bugs and errors. This will allow us to ensure that we maintain the functionality of the newly written code when fixing bugs or adding features to the rest of the system. The implementation of these types of testing will increase the overall testability of the project as SuperTux currently has not testing framework.

Lastly, performance of the system could be negatively affected by the new multiplayer feature. This is mainly due to increasingly concurrent nature of the multiplayer feature set. When running multiplayer, you must calculate the positioning of each player's Tux and rerender it on the screen. This increased processing could negatively affect the performance of the system.

Sequence Diagrams

Starting a Level in Multiplayer

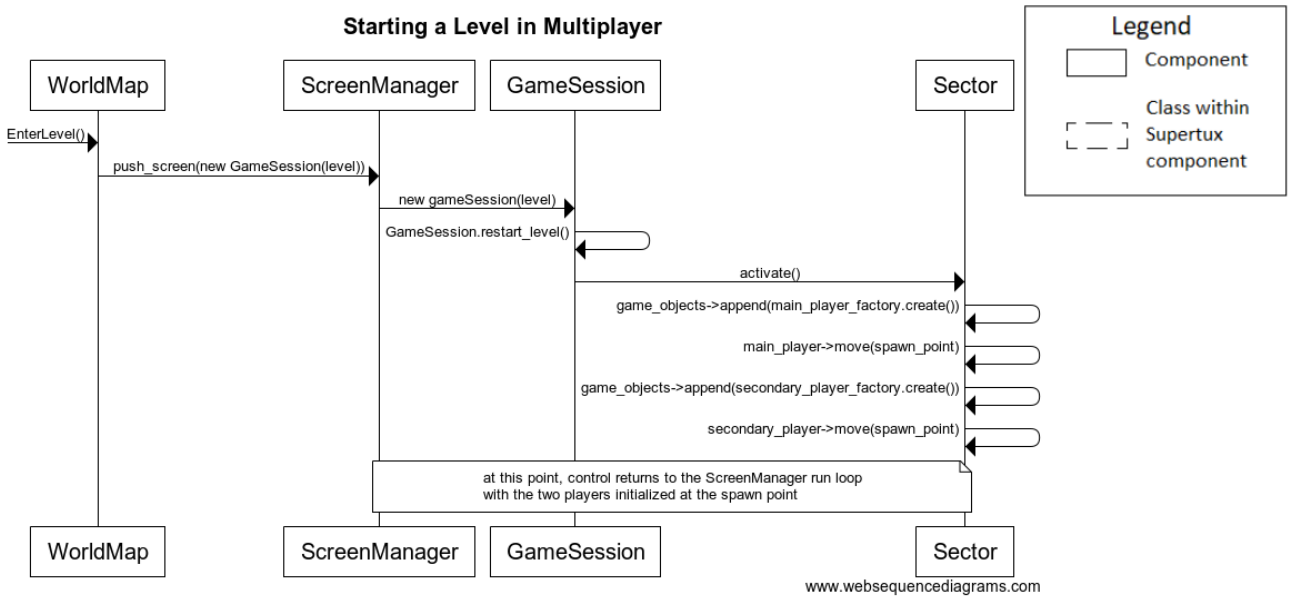

Figure 3: Sequence diagram for starting a level in multiplayer

Figure 3 shows the sequence of operations performed when a level is started in multiplayer mode. It is very similar to the flow of single player mode, since the changes are mostly limited to the Supertux subsystem. In this case, the change is seen in the Sector object, as that is the one that keeps track of the game objects. In multiplayer mode, the Sector has two factories – one for MainPlayers and one for SecondaryPlayers. Each is used to generate a player, and both players are appended to the game_objects collection.

Updating the game

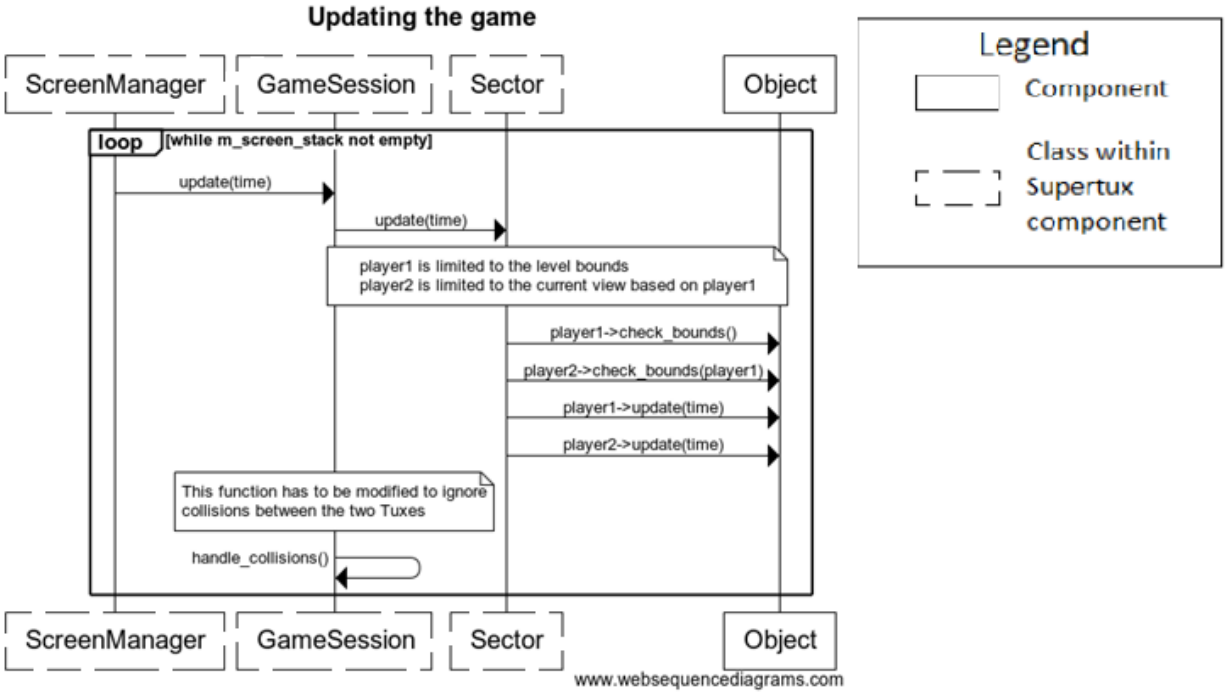

Figure 4: Sequence diagram for updating the game

Figure 4 shows the sequence of operations involved in the top-level game loop in multiplayer mode. Like the level initialization, the changes in this sequence are mostly limited to the Sector, as well as the Object component. Both players have their bounds checked, but player2 is bound by player1 since the screen follows player1. Both players are then updated along with all other game objects, and the specifics of those updates are dealt with by the individual objects. That is why the division of types between MainPlayer and SecondaryPlayer work so well – each one has its own update function, so they can have different behaviour. The handlecollisions function also has to be modified to handle (or ignore) collisions between the two players.

Impacted Directories and Files

control – This directory will need to be modified to allow for the possibility of multiple different inputs – eg. a keyboard and a controller.
supertux/Sector.cpp – This is the file that handles the gameobjects, including the player, so will see a lot of the changes. Every place that handles the player will now need to handle both, or specify only the main player.
object/Player.cpp – This is the base class we will use to create the MainPlayer and SecondaryPlayer classes. Some functionality, which is not wanted for the SecondaryPlayer, will be removed from this class into MainPlayer.
supertux/object_factory.cpp – This is the existing object factory. It will be modified to also provide factories for MainPlayer and SecondaryPlayer. This solution doesn't quite match the Abstract Factory design pattern, as there is no abstract PlayerFactory, but it meshes better with the existing system.

New files added to the system

object/MainPlayer.hpp, supertux/MainPlayer.cpp – This object represents the main player, which still has full functionality. It extends Player.
object/SecondaryPlayer.hpp, supertux/SecondaryPlayer.cpp – This also extends Player, but it is a secondary player and will lack some functionality.

Plans for Testing

There are four aspects to our approach to testing: unit, integration, system level and regression testing.

Unit Testing

Unit testing at the code level will be run using a test harness for newly written blocks of code, ensuring they function as intended. Ex. Check the input handler can handle multiple new inputs at the same time by using a test suite and console logs.

Integration Testing

Integration testing will be used to ensure that modifications within a subsystem does not change the overall functionality of the subsystem or cause any unexpected errors. This will also be run in a test harness. Ex. Check that the new input handler can still handle single inputs for one player by using gameplay testing with a single player

System Level Testing

System level testing is to ensure that the system as a whole runs to specifications after full integration of the feature. Including overall functionality and no errors. Ex. Have play testers run the game checking that both player one and two can play as intended, and look for any unexpected errors.

Regression Testing

Regression testing will be used to ensure that any fatal errors found in old builds or during development do not cause an error in the current version of the program. Specifics for running these tests will vary from error to error.

Potential Risks

If this enhancement were to be implemented, there are a number of potential risks and limitations that exist. The first is that no member of our group has played through the entirety of SuperTux, and there is a possibility that our enhancement may change the game in a way that would negatively impact gameplay, perhaps making specific levels harder to beat with two Tux's taking up space on the screen. Additionally, there may be small technicalities within the code we do not know of, which would make implementing this a much more arduous task than we originally thought.

Another potential risk would be that the second player may have odd interactions with power-ups that we did not expect. Ideally, the second player will be able to pick up power-ups and hit enemies just as the first player does with minimal changing of other subsystems, but there is a chance that the process may be more complex than we thought due to particulars of the code. This could be addressed, but would simply take more time to implement.

A risk that we considered that is not a problem would be enemy AI - enemies in SuperTux do not have AI, and simply walk from side to side. A second Tux would not affect them.

References

[1] 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 December 2017].

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

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