Tuxedo Amigos

SuperTux Enhancement Proposal
CISC 326 - Assignment #3

Ashley Drouillard	10183354	14aad4
Cesur Kavaslar	10176178	14cck
Chris Gray	10185372	14cmg5
Diana Balant	10176211	14dab9
Matt Dixon	10185771	14mkd3
Sebastian Hoefert	10169337	14sbh2

Table of Contents

Abstract

This report introduces our Enhancement Proposal to the game SuperTux and discusses feature itself and the motivation for the feature. It then elaborates on of the different techniques that were considered and dives into a SAAM analysis of the two possible implementations, examining their pros and cons and how each relates to the stakeholders of the game.

The chosen implementation is then described in further detail and why it was chosen is explained. Then the chosen implementation of the feature's effect on the architecture is discussed; more specifically it's impact, the specific components affected, plans for testing the feature, potential risks and limitations of the feature, and concurrency. We then detail two separate sequence diagrams to further describe how the new implementation works. Lastly we discussed developer team issues, group limitations, some lessons our team learned, and briefly conclude the report.

Introduction and Overview

The contents of this report details our proposed enhancement feature for the game Supertux and subsequently goes into how this feature would affect the existing systems, conceptual architecture, and dependencies. Our proposed enhancement is a score system akin to similar systems found in games such as Super Mario Bros and Sonic Mania. This score systems seeks to motivate players to further play Supertux.

We began by looking at how we wanted the score system to work with regard to how many points would be gained or lost per specific action. Next we looked at the driving motivations behind our feature. We had different implementations considered for the score system which will be detailed. Next we looked at SAAM analysis for these different implementation styles so that we could decide which implementation would be the best suited for Supertux.

With our implementation decided upon, we discussed how it could affect the conceptual architecture and other components. Subsequently, how the feature is going to be tested has been detailed. Lastly, potential risks and limitations, concurrency, two sequence diagrams, Supertux team issues, group limitations, and lessons learned were discussed before finally concluding.

Enhancement Proposal

Idea: Score System

For our enhancement proposal, we plan to introduce a high score system to SuperTux. This would be another level statistic that would be calculated based on other existing statistics. The system would reward 500 points per coin collected, 750 points per badguy killed, and 1000 points for each secret found. Additionally, If you were to complete the level within the target time, it would double your score. As with the other best level statistics, your score would reset if tux were to die, and your score progress would save after reach a checkpoint.

Motivation for Feature

Our proposed feature was motivated by looking at other popular platformers, both old and new. We noticed that in games such as Super Mario Bros the score system was used to great effect in almost all of the main entries in the series. Points were awarded for performing a variety of different actions, such as collecting coins and defeating bosses (Point, 2017). Similarly, games such as the recent Sonic Mania and the far less modern Megaman 1 both implemented score systems.

As with these previous titles, a score system would bring a number of benefits to the game and the player. Firstly, it encourages players to replay levels to improve their score. Such as finding secret areas they missed or completing the level in a faster time. Secondly, it incentivises exploration of the level because it gives players a reason to collect every coin, find every secret, and kill every enemy. Lastly, it enables friendly competition between other players for who can get a higher score. These three points help to increase play time for Supertux and would increase the player base for the game.

Implementations Considered

We considered two different way of implementing the score feature. The first implementation we considered is the simpler of the two. In the standard version of the game, every statistic used to calculate score (coins, enemies killed, completion time and secrets explored), are already tracked within an object and displayed at the end of the level. To calculate and display the score at the end of the level, we simply would have needed to do some arithmetic operations and then write it to the end screen along with the other stats that are already displayed.

However, the second implementation we considered was similar to the score system in Mario. In this implementation, your score is continually calculated, updated, and written as you play through the level. This is similar to how coins picked up are currently updated and displayed on the screen in the current version of the game. We would display the score to the top centre of the screen.

SAAM Analysis

Stakeholders

Since SuperTux is open-sourced, there are no investors and there are no publishers of the game. The only stakeholders involved are the users and the developers.

We categorized the non-functional requirements we will discuss with their related stakeholder. In the next step, we will analyze the possible implementations and assess how each individual method would influence these requirements, and in turn the stakeholders of the game.

Users & Players	Developers
● Usability ● Performance	● Maintainability ● Evolvability & Modifiability ● Testability

Implementation 1

Our first method of implementation involved only calculating and displaying the new score at the end of the level. This method is the most straightforward way of implementing the score feature because it collects the end level statistics, calculates the total score, and displays it on the end of level screen.

Pros	Cons
Evolvability & Modifiability ● Much more simple to add to the game, easy changes to already existing functions & files Maintainability ● Easier to go back and correct errors in the future Reliability ● Less chance of making errors Testability ● Fewer different ways to test Performance ● Fewer operations occurring less often increasing performance of system	Usability ● No impact to the user throughout game play, however the user will only see it at the end

Usability

A downside to this method is that although it does not directly affect the user's experience. The player will not see their score until they complete the level, making the scoring system slightly less intuitive, less incentivizing, and less worthwhile to implement.

Performance
This method looks like it could be the prefered technique because there are less calculations and operations occurring during game play and as a result not causing an impact on the performance of the game.

Reliability
The implementation is also likely to be more reliable because there would be less and more simple code, lowering the likelihood of coding errors and in turn increasing the reliability of the feature.

Maintainability
This version of implementing the feature make maintenance very easy. The code to calculate the score is more organized and in fewer different files and functions, simplifying the process of finding and correcting any errors.

Evolvability & Modifiability
In addition, it is an easier method to modify if we decide to make changes to the feature in the future. For example if we wanted to change how the score was calculated, we would need to change fewer lines in fewer different locations. This also increases the reliability and testability because the code will be simpler to locate and write tests for.

Testability
Using this technique would be beneficial for testing because there would be less code in fewer different places. This would make it easier to write tests for and to analyze the results and interactions between the components.

Implementation 2

Our second method, the method we ultimately chose, of implementing our feature involves updating the score and displaying it as the player progresses through each level. Although there are more potential downsides to this implementation, we believe it is worthwhile as it provides a more engaging and rewarding experience for the players as they try to better their high score.

Pros	Cons
Usability ● Slightly more user friendly because the score is displayed as the user progresses through the level	Performance ● Potential to slow down game play as operations are occurring more frequently Maintainability ● More difficult to modify because code will be distributed across multiple locations Reliability ● Variety of locations increases the opportunity for small errors or miscalculations

	Testability ● More tests required Evolvability & Modifiability ● More tedious to modify and evolve in comparison to the first method.

Usability
This implementation is more user friendly in which the player can see their score progress as they play through the level opposed to only seeing their score display once at the beginning or end. This makes it more obvious to the player on how the system works and provides a more engaging experience to the user.

Performance
This implementation technique has the potential to slow down performance since mathematical and drawing operations would occur more frequently, and the score would be displayed concurrently as the user plays.

Maintainability
This implementation would be more inconvenient to maintain relative to the first method because the code to calculate and display the score would be distributed between a wider variety of files and functions so finding the source of error would be more difficult.

Reliability
This implementation is less reliable as there is a wider range of files and functions which may increase the opportunity for errors to occur and can be more difficult to trace.

Evolvability & Modifiability
The current SuperTux code can easily be modified to implement this feature as it'd only require adding a few extra lines of codes to existing functions. However, in comparison to the first method, it would be a more tedious task as you'd have to go through all of the additional functions and files that are not impacted in the first implementation. Additionally, evolving the feature further such as altering the amount of points awarded or introducing new factors that impact the score would also be much more time consuming than the first implementation.

Testability
This implementation would require more testing of the feature. As there are more impacted files and dependencies within the architecture , more tests would be required to ensure that these files and dependencies interact correctly.

Architecture

Impacts of Enhancement

Implementing our new feature has minimal impact on the SuperTux source program. The new feature will not create any changes to our existing architecture. Since all changes occur in pre-existing files, no new components are created and no new files are added in any of the components. As well, all dependencies that are created in the new code already exist in the original source program. Therefore, our original architecture remains unaffected.

Affected Components

The files that are modified for the implementation of our feature are:

- supertux/Statistics.cpp
- supertux/Statistics.hpp
- supertux/LevelIntro.cpp
- supertux/game_session.cpp
- badguy/Badguy.cpp
- object/Coin.cpp
- level/Secret.cpp
- supertux/player_status.cpp

All of these files were all mapped to top-level components during Assignment #2. The affected top level components of our conceptual architecture are therefore:

- Game Logic
- Game Engine
- Libraries

These changes have very little impact on our concrete architecture. As mentioned previously, no dependencies are created between components which did not already have a dependency between them and no new components are needed to implement our changes.

Plans for Testing

We plan to test our feature by playing the game, doing systematic white box testing of the code related to the features implementation, and then subsequently looking at how the feature could impact interactions between other subsystems and their dependencies.

Firstly, once the code has been implemented, we will test the source code we wrote via systematic white box testing methods. These methods would include statement coverage or basic block coverage, which would highlight any discrepancies between the code we wrote and the original Supertux source code. Subsequently, it would highlight any syntactical errors in the code that may have an impact on how the feature works when used.

Secondly, we would play through the first level of the game making sure that each implementation of the feature worked as intended. The goal of this is to make sure that there are no errors in the semantics of the newly written code, such as awarding 500,000 points per coin instead of 500. This would include making sure that the following points function properly:

- Picking up a coin
- Killing an enemy
- Finding a secret
- Dying
- Going past a checkpoint
- Completing a level within the time limit
- Completing a level outside of the time limit

Lastly, we would make sure to test that existing dependencies aren't affected by the implementation of this feature. Such as but not limited to, calls from game logic to GUI that make sure items are drawn to the screen correctly, as well as calls from game engine to libraries that make sure that the saved player status file is read correctly.

Potential Risks and Limitations

Since our change does not make any major architecture changes, there are minimal risks and limitations involved with the change. One concern that does exist is that our new feature could impact the performance of the game on lower end systems, as it adds more data to be stored into memory and more calculations occurring throughout the course of the game. With the addition of another image being drawn at the start of each frame could also lead to slowing down the game if there is a frame where a lot of objects need to be drawn. Drawing the score requires function calls and calculations to make the text the right size and location for the aspect ratio of the game, so it takes more time than just drawing a simple sprite. Another risk that could present itself is that drawing the text adds more clutter to the screen, so it could obstruct the screen of players with smaller displays, negatively impacting their experience.

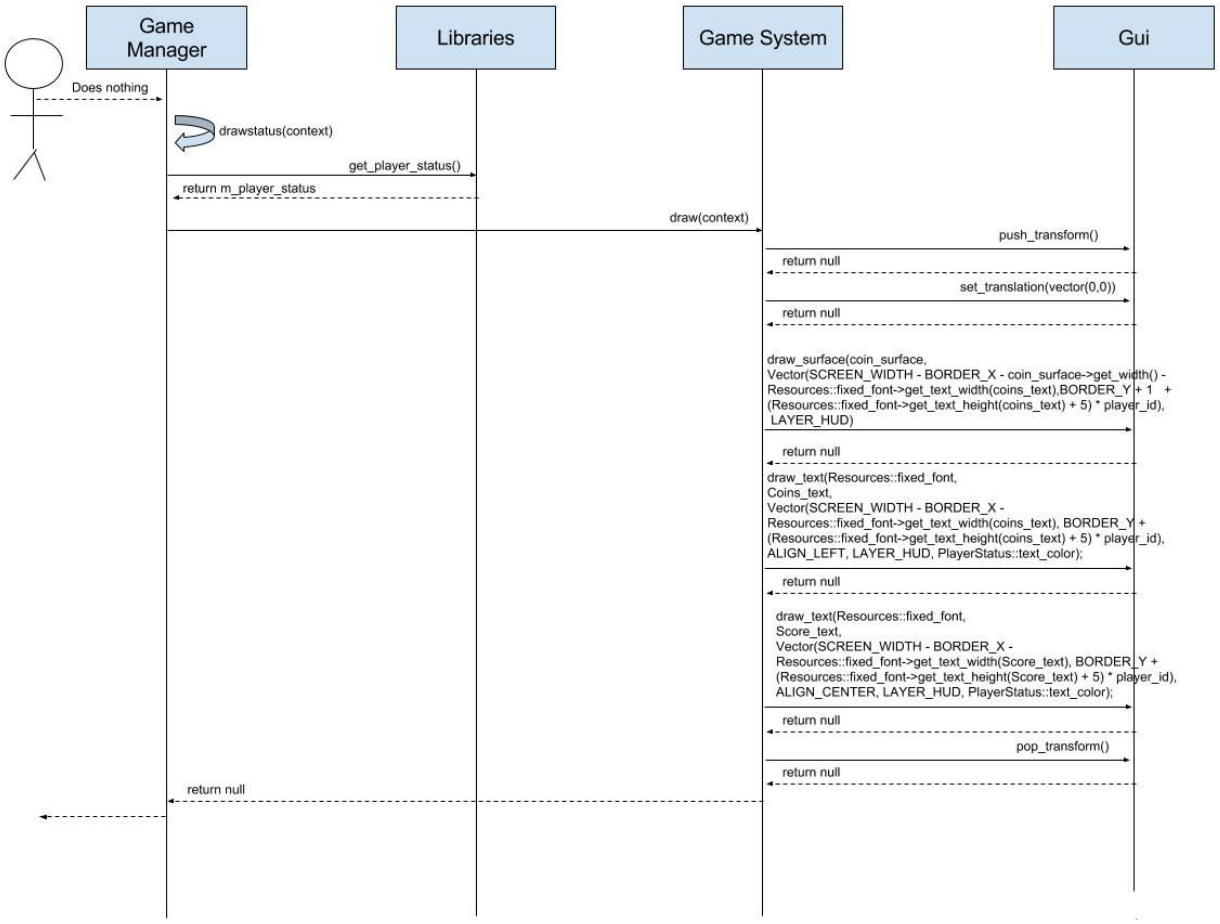
Concurrency

Throughout game play, a variety of things are occurring concurrently. This concurrency is managed in the game engine and supports audio and animation happening at the same time. Our implementation of the enhancement proposal of a score system will also be utilizing this concurrency. For example, the user will continue to play/interact with the user interface as the new score system is being tallied and displayed to the screen.

The reason we chose to implement the feature concurrently is because it improves the user's experience by allowing for the score to be displayed as the player progresses through the level. Our proposal is unlikely to significantly impact performance and therefore we do not need to modify the process of concurrency in the game design, only make some additions.

Sequence Diagrams

Sequence Diagram #1 - Coin Collected



Score gets updated after a coin has been collected

This use case occurs after the player had picked up a coin, thus increasing both the coin count and score variables in the statistics object. During the update step in the game session file there is a call where all the objects in the current sector are updated and drawn, including the game session object itself. When the rest of the game is finished drawing the game session finally calls the internal method of drawstatus(context) which draws the information from the current player status object to the screen.

To do this it must first get the player status object. This is done by invoking the get_player_status method() from the savegame object which returns the current player status. After the object is returned, the draw function is called for the player status object. Before drawing, there is a check to make sure that the current coins collected, and current score are not equal to the coins collected and score that are already being displayed, as to not waste time drawing something that is already there. Since the user has just picked up a coin, there is a discrepancy between these counts and therefore the new counts are scheduled to be drawn.

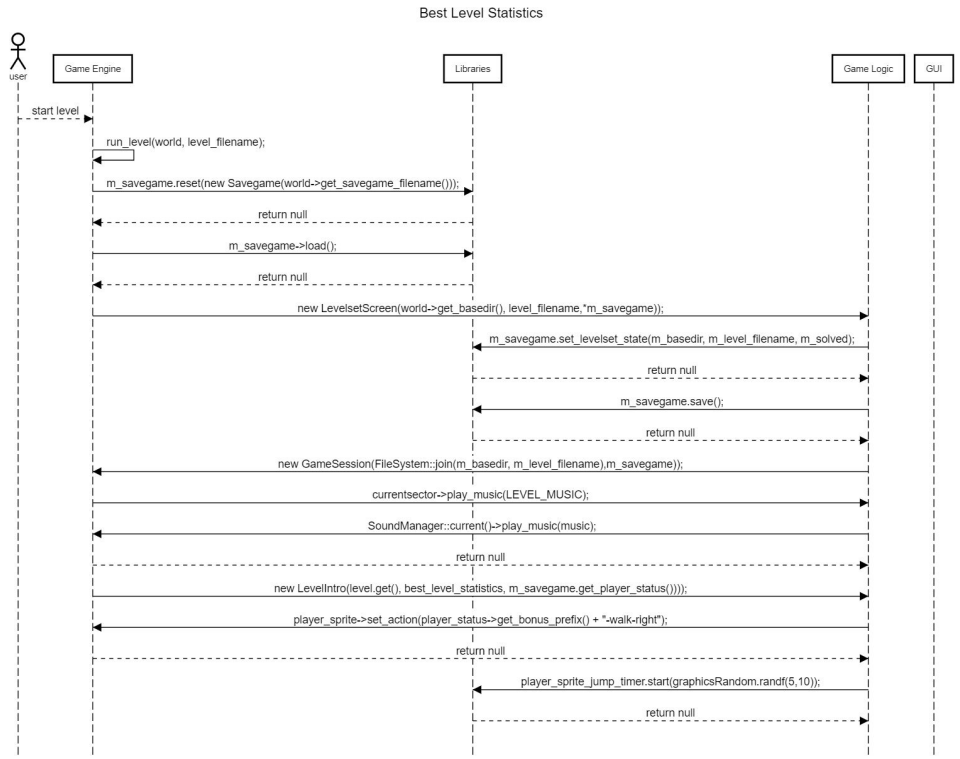
To do this, the coins and score integers are converted into strings so they can be drawn with the draw_text method. Once all of this preparation is done a new layer is pushed onto the transform stack, which is a stack that text and other images can be drawn to and transformations can be applied to. Once a layer pops off the stack it is set to be drawn to the screen. The first thing to be applied to the layer is the coin surface, which is the little picture of a coin beside the count. This is done via the .draw_surface method.

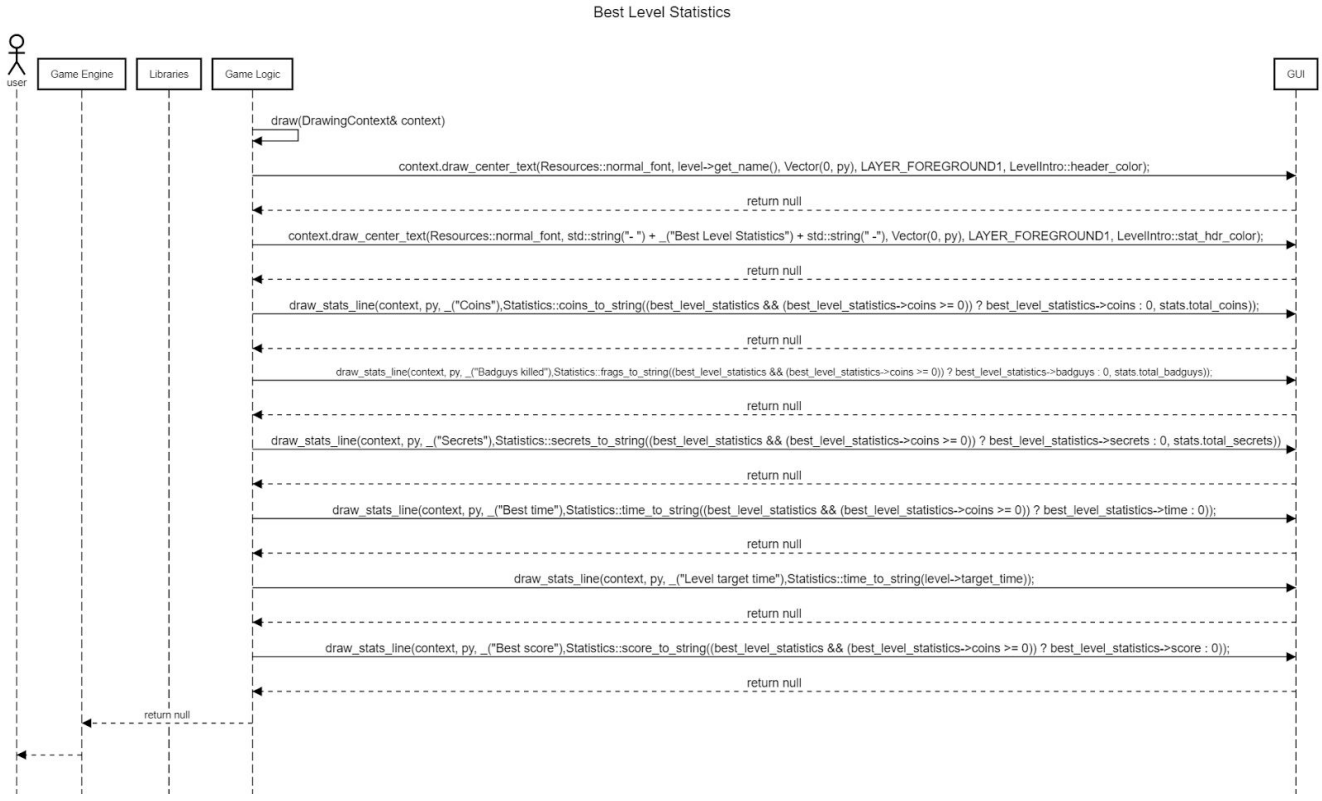
To break down this call a bit, first it identifies what its drawing, the coin_surface. Then is calculates a vector with x,y coordinates. The calculations make sure that the image is the same size relative to the size of the text that will be drawn next. Finally it chooses what layer, or plane to be drawn on, for all of these drawings they will be on the HUD plane, meaning that they will appear above everything else on the screen.

Next thing that is drawn is the number of coins collected, this time with the draw_text method. Which works very much the same way as the draw surface, this time with a few more variables such as the font, text colour and text alignment. Finally our method gets called which draws the score text to the center of the screen on the x axis, by positioning it at SCREEN_WIDTH/2. Our text uses the same font, y axis position and colour as the con text to remain consistent. With all these entities drawn the layer is popped off the transform stack and is drawn at the start of the next frame.

Sequence Diagram #2 - User Starts Level

This sequence diagram has been broken up into two parts for readability and simplicity.





User starts a level

The first part of this use case starts with the user starting a level. This information gets relayed to the game engine which calls the run_level function. This function resets the savegame object for the level and loads a blank savegame. Once this setup is complete it loads the level. Setting the current save game state to reflect the level that is being loaded, and saving the changes.

Next a game session object is created for the level. In the set up, the music for the level is loaded and played. Next the game engine creates a new levelintro object which is the screen that displays as the level is loading. There is some setup for this object as it first must see what sprite to load in for tux (ie. if it should be normal tux, santa tux, or a powered up tux). It sets the sprite to a walking version of that tux. As well, a timer is started from the library timer object to set a random time for the tux sprite to jump in the intro animation.

Once this is done it moves onto the second part of the sequence diagram. When the draw function is called onto the level intro, it goes into a sequence of drawing all the text for the players stats. This uses the transform stack as mentioned in the first use case. One by one each stat is drawn to the screen based on the user's best statistics. If it is the first time the user has played the level it will just display 0. The function we implement here follows suit, by checking and comparing if there is already a high score. Iif so draw it and if not, it draws a zero instead. Once all of this is completed the control is handed back over to the player and as soon as they press a button they will be able to play the level.

Team Issues

This feature wouldn't require a large team to implement it because it relies on a lot of features that are already a part of the game, such as drawing, updating the value, and receiving information of events. Therefore, with the small team, you would perhaps need at least one person who is knowledgeable of the system to be able to find, understand, and utilize these existing features. The issues with this is that many of the original developers who have a deep understanding of the system, are no longer on the team, or are not as involved with the project. Lastly, being an open source game, open source developers may try to add on to this new feature by creating new possibilities to earn points. Although, many open source developers may not be experienced in what they are doing, or may have too complex of an idea, and may inadvertently create more coupling in the system.

Group Limitations

When it came to us implementing the feature, we were strongly limited by the SuperTux code. The SuperTux code is very poorly organized and is scarcely commented, so it is very difficult to trace the code and understand each file. We had a general idea of what components would be used by our new feature, but when looking at the code, it was hard to find where exactly items get drawn to the screen and where files such as the player status file get invoked. As well, we were strongly limited by none of us knowing C++.

Lessons Learned

What we learned from this assignment was that a new feature doesn't have to be a large change to the game. New features are often thought of as large changes or additions to the system, that have a large obvious effect on the program. Compared to other larger, ideas we had, for example local multiplayer, this feature is a lot simpler to implement and poses a lot fewer potential risks. Our feature may not make much of an overall change to the gameplay, but it is still an enjoyable feature that gives an added challenge to players.

As well, we learned that it would be more time consuming to try to implement the feature going into the source program blind than it would be to first become familiar with the architecture and then implement the feature. Since we had a very good understanding of the architecture from the previous two assignments, it made it easier to begin implementing the feature as we were already familiar with the function of each of the components, and the dependencies between them.

Conclusions

In conclusion, our proposed score system offers replay value and provides a more enriching experience to Supertux players. We chose our second implementation method of updating and displaying the player's score as they progress through the level. After conducting a SAAM Analysis we found this method to be more work to implement compared to our first implementation in terms of performance, evolvability, reliability, maintainability, and testability. Although the usability makes it more worthwhile as it provides a more engaging experience to the player. Ultimately, this feature is a straightforward and beneficial addition to the Supertux game.

Data Dictionary

Add-Ons - component of game engine, downloads levels and language packs from an online repository

Animations - component of the game engine, stores and plays all animation files

Audio - component of game engine, stores and plays all audio files

Enemies - component of game system, contains all of the information for the various bad guys

Game Engine - contains game manager, audio, animation, add-ons, and P&C components

Game Logic - contains game system, level editor and input components

Game Manager - component of game engine, controls flow of game

Game System - component of game logic layer, home of interactive objects, level and enemies

Graphical User Interface (GUI) - draws to the screen

Input (HID) - component of game logic, handles user input

Interactive Objects - component of game system, handles info of non-enemy interactables and player

Level - component of the game system, acts a a blueprint for each level of the game

Level Editor - component of game logic, where players can create/modify levels

Libraries - contains third party code libraries that are used in game engine

Physics & Collision - component of game engine, handles object interactions, collisions, and forces

Naming Conventions

GUI : Graphical User Interface
HID: Human Interface Device
ID: Identification
I/O : Input and Output
OO: Object Oriented
OS : Operating System
P&C: physics and collisions
TA: Teaching Assistant

References

Point. (2017, October 9). Retrieved December 1, 2017, from https://www.mariowiki.com/Point