Tuxedo Amigos

SuperTux Conceptual Architecture
CISC 326 - Assignment #1

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

SuperTux is an open sourced game developed by Bill Kendrick inspired by the Mario Brothers series. Throughout this report we will outline and discuss a variety of different aspects of the game's conceptual architecture.

The initial architecture was built using our previous knowledge of conceptual architecture, resources like our textbook, and short looks into the documentation of the game. It was a hybrid architecture using a layered and object-oriented design. We refined it to our final conceptual architecture by making a few notable changes. We reversed the dependency of audio, added an animation component, and rearranged the game system component. In our final architecture we included four layers: game logic, game engine, libraries, and the OS abstraction layer.

In our highest layer, game logic, we have the game system, audio, animation, level, and input/HID components. The game system houses the core components for the game logic and is an object-oriented design. The audio and animations components are designed to streamline the calls to the resource manager. The HID contains interfacing so that users can interact and play the game.

In the game engine, we have the resource manager, camera, physics and collisions, and a dependency on add-ons. Physics and collisions is responsible for managing the object interactions to ensure the game executes the correct actions. The camera provides the view to the user and follows Tux as he travels through the map. The resource manager is called to retrieve specific files, such as audio and animation sprites. The add-on component allows users to download different levels from an online repository.

Following game engine we have the libraries layer, which holds the premade third-party methods used in the game engine. Finally we have the OS abstraction layer, which allows the game to be played on a variety of different platforms.

To get a deeper understanding of the conceptual architecture, we have included two sequence diagrams, one where Tux jumps and breaks a block with his head, and one where Mr. Iceblock walks into Tux standing still and causes him to become small Tux.

Throughout the construction of the conceptual architecture, we learned a variety of things. Most notably, we learned that no conceptual architecture is perfect. We spent a lot of time discussing, changing, and revising the initial architecture before realizing that the conceptual architecture isn't going to be exactly like the game. There can be many different equally correct conceptual architectures as long as it can be justified.

Introduction and Overview

SuperTux is a free and open sourced video game available on Linux, Windows, Mac, and Android devices, inspired by the Super Mario Brothers series. It is a two-dimensional "jump'n run" style game that features Linux's penguin mascot, Tux, and his heroic efforts to save the princess.

The game was developed by Bill Kendrick and is currently maintained by the SuperTux Development Team. It was published under the GNU General Public License. The game was originally released as version 0.1.1 on April 2nd, 2003. The most recent version, version 0.5.1, was released on November 5th, 2016. As of May 2017 Supertux aggregated also over 850,000 downloads via Sourceforge.net.

In this report, we will go through our process of deriving the conceptual architecture of SuperTux. We begin by examining the derivation of our initial architecture hypothesis and the justification we had for the decisions we made. This includes justifying our choice in architectural style as well as the components we included.

Next, we will look at the changes we made to our initial hypothesis and how it evolved into our final revised architecture. We will break down the changes we made to our hypothesis and the reasoning behind them.

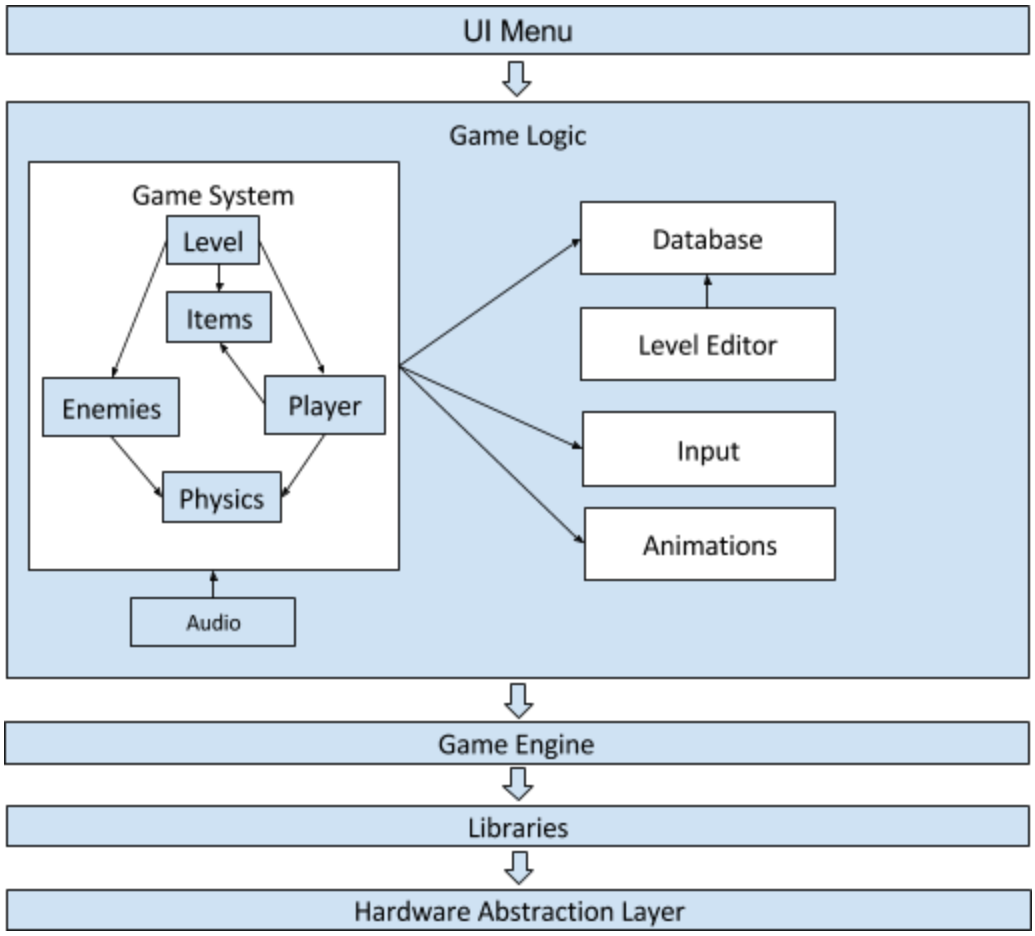
From our new revised architecture we will take an in-depth look at our architecture, breaking down our architecture by component. We will provide a quick explanation of each component as well as explain how the components work and fit together, with emphasis on high cohesion and low coupling between the components.

We will then work through a use case diagram for the game and two sequence diagrams to demonstrate how our architecture functions in practice. The first sequence diagram will run through the event *"Tux jumps and breaks a block with his head"*. The second sequence diagram will deal with the event *"Mr. Iceblock walks into big Tux standing still and causes him to become small Tux"*.

Finally, we will end with a conclusion summarizing the main points of our report as well as a proposal for future directions. This is followed by a reflection on the assignment in our section titled Lessons Learned containing what we learned by doing this assignment and some things we wish we knew ahead of time.

At the end of the report you will find a data dictionary glossary that briefly defines the key terms used in our architecture. There is also a section titled naming conventions that breaks down any short hands we use throughout the report and finally you will find a list of web links used as references we used in writing this report. These can be used for additional readings for the reader that expand on some of the ideas mentioned in this report.

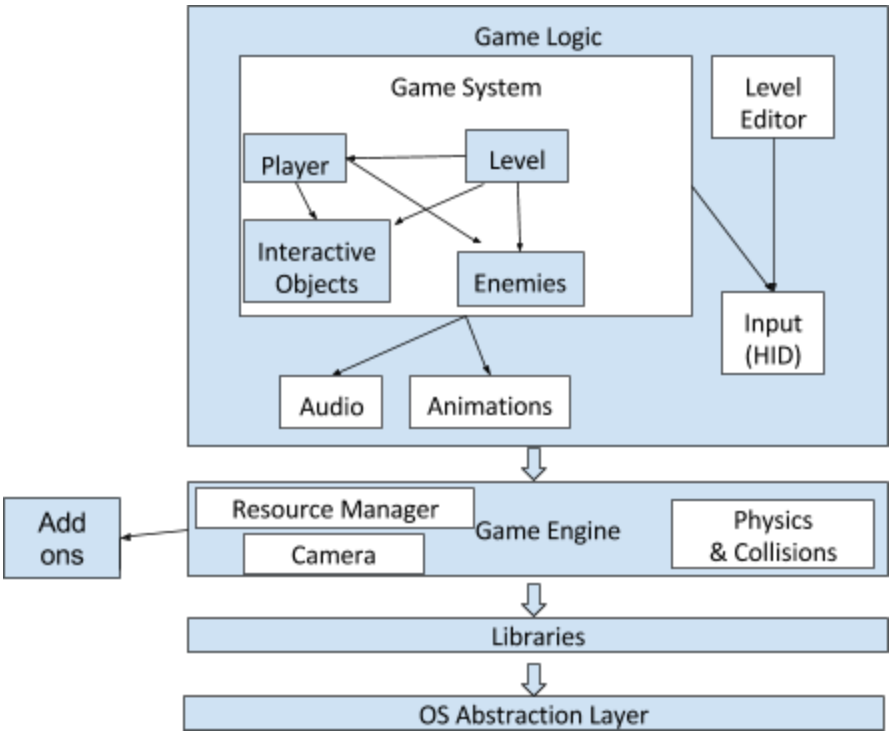
Initial Architecture



Our initial architecture is a combination of the layed and object oriented styles. We derived this architecture using a combination of different techniques. We started by looking at some of the conceptual architectures from previous years to give us a general idea of how to structure an architecture for a video game. We then read through the components found in the textbook and compiled a list that we believed would be present in SuperTux, such as powerups, physics, animation, etc. We also briefly looked through a few online resources relating to the design of SuperTux.

With this information we constructed a crude initial version of our architecture, not really sure what edges should be connected to what, as well as the direction of some dependencies. We looked into many different styles of architecture before finally deciding on a combination of layered and object oriented, such as a client server style to handle the add-ons sections of the game, a repository style, an entirely layered, and an entirely object oriented style. We quickly realised that these styles were not what we were looking for as they would just not work or would have major coupling and or cohesion issues. After many team discussions and a meeting with our TA we realized that overall our architecture was structured well, but just needed some trimming, such as removing and combining some components, switching dependencies and adding forgotten components.

Revised Architecture



Our revised architecture was created after much deliberation on our part, help from our TA, and a realization that there were many flaws with our first endeavour. We continued using a mix of object oriented and layered design for our architecture.

The first change was removing the user interface layer entirely, based on the assumption that the user interface will be stored in all components that require it. As well, we removed the database component because it didn't fit in our high level architecture.

Several other changes were made in the game logic layer. The game system component now depends on the audio component instead of the other way around. Next, items was changed to interactive objects so that it could be a more inclusive category that would include elements such as switches and ladders. We moved physics and collisions to the game engine component because we agreed this is where it fit logically, instead of being a component of the game system itself.

In addition to physics and collisions, we added a camera and a resource manager to the game engine because we had missed them entirely the first time around. We also added the add-ons section of the game to the architecture as a component dependent on the game engine.

Finally, we replaced the hardware abstraction layer with an operating system abstraction layer. This was because in our case what we meant by this layer more closely aligned with the definition of an operating system abstraction layer, than a hardware abstraction layer. This layer was intended to allow SuperTux to run on many different operating systems.

Components

As previously stated, our conceptual architecture is a hybrid between an object-oriented and layered systems. It is composed of higher-level components, with OO style within. The tiers we selected are game logic, level editor, game engine, libraries, and the OS abstraction layer. In this summary, we will briefly go through the intended purpose of each tier and their components.

Game Logic

The game logic is the highest layer of our architecture and has a dependency on the game engine. Inside the game logic component there is an object-oriented design of the game system, level editor, input, audio, and animation.

Game System

This component is where the core components of the game logic reside. It is broken up into yet another object oriented design. The four main objects that make up this component are as follows. All of the components are grouped in the game system object because it allows for easy access to shared components within the game logic layer, such as audio and animation.

Level

The level component acts a a blueprint for each level of the game. It will have information stored for each level about where to place the player, all of the enemies, and interactables objects when the stage loads. It will also make sure to load in the correct music and graphics for the level at the start and finally set the camera to the desired position, usually following the player. As a result of these actions, this is why the level component has dependencies to the other three components of the game system.

Player

This is the object that is going to hold the information for the player character, such as the current power-up state of the player, total number of coins the player has collected while playing, projectile fireballs created by Tux, etc. As well, it will also contain methods to determine what happens when a collision occurs with enemies or interactable objects. Such as when the player is colliding with a coin, the player object will know to update the player's current total coins by one. These methods will usually also talk to the other components to tell them what is happening, so they can run commands related to the collisions. A detailed example of these interactions is illustrated in the sequence diagrams.

Enemies

The enemies object contains all of the information for the various bad guys found within the game. This information can be anything from their size and collision boundaries, to what happens when the player collides with them based on the location of the collision. Since the player object is responsible for telling the enemies object where they collided there are no outgoing dependencies for the component. We chose to do it this way to help reduce the amount of coupling within in the game system.

Interactables Objects

This component of the game system is responsible for holding the information of pretty much everything else the player can interact with in the game. We defined an interactable object as something that is not an enemy but has some interaction that occurs when the player collides with it. Such as a powerup disappearing and playing a sound when it collides with the player. Just like the enemies component, there are no outgoing dependencies and relies on the player object relaying the collision information. Once again this was done in an attempt to reduce the overall coupling in the system.

Level Editor

Level editor allows the player to make or modify levels in SuperTux. The level editor depends on the HID for user input. It uses in game assets such as projectiles, bosses, or interactive objects for level creation and modification. It would get these from the game engine layer which contains the game assets.

Human Interface Device (HID)

The human interface device (HID) contains game specific interfacing such as the keyboard, mouse, and controller. The game system and level editor both depend on the human interface device, because they need user input for player actions.

Audio

Initially in our original architecture, we had the dependency for our audio system going the other way around. We've since changed it so that the component is inactive until invoked by the game system. Once invoked, audio pulls the files from the resource manager which is a much more efficient and effective use of resources.

This was explicitly added to the conceptual architecture after a discussion and decision that it streamlines the process. This allows the audio component to process or wrap the file quickly before returning it in a more usable and friendly condition to the game system.

Animations

Quite similar to audio, the animation component is invoked by the game system which controls how different objects interact together. This component goes and retrieves the file and information from the resource manager and then returns the specified sprite to the game system.

Both Audio and Animation are able to run concurrently due to the design of our game engine. A more in-depth example is provided in our sequence diagrams.

Game Engine

The second layer in the architecture is the game engine layer. This layer is home to the main loop of the game. This loop controls the current state of the game and allows for different components to run concurrently such as camera and physics. There are three main components in the game engine layer.

Physics & Collision

The first of these components is physics and collision. This component is responsible for handling object interactions and collisions. As well, it controls the forces in game such as gravity and momentum. This component alerts the game system when a collision is happening so the game system can then execute the correct sequence of actions. This is demonstrated in our sequence diagrams later in the report.

We originally believed physics was a component in the game system but moved it to the Game Engine as it fits more logically here. Collision detection being handled in the main loop of the game allows it to happen concurrently with other components. Removing physics from the game system also lowered the coupling of components in the game system as everything depended on physics.

Camera

The camera component is responsible for providing the view of the map to the player. For the most part, the camera is locked to the player and follows them as they progress through the level. However, there are several other camera modes. Manual camera is used in the game editor as the editor controls where on the map they are looking. Fixed camera and scroll to camera modes are also used in cutscenes in order to create a more cinematic experience for the player.

Resource Manager

The final component of the game engine layer is the resource manager. Fairly intuitively, the resource manager is responsible for accessing the game's resources and assets and retrieving them for the game logic layer. For example, the resource manager is called to retrieve an animation file for the player. More examples of this component in action are found in the sequence diagrams later in the report.

Add-Ons

The Add-On section of the game is what made us originally consider a Client/Server architecture for a brief time. The reason for this was that it was server side element that users/clients didn't directly influence. Going into the Add-Ons section of the game allows the user to download additional levels from an online repository which the user can install at will. These are then stored in the Game Engine component where the Resource Manager then sorts them into the game's asset folders which is illustrated via the dependency arrow in the conceptual architecture. In the same menu, the user is able to install a large number of additional language packs that have been made for the game.

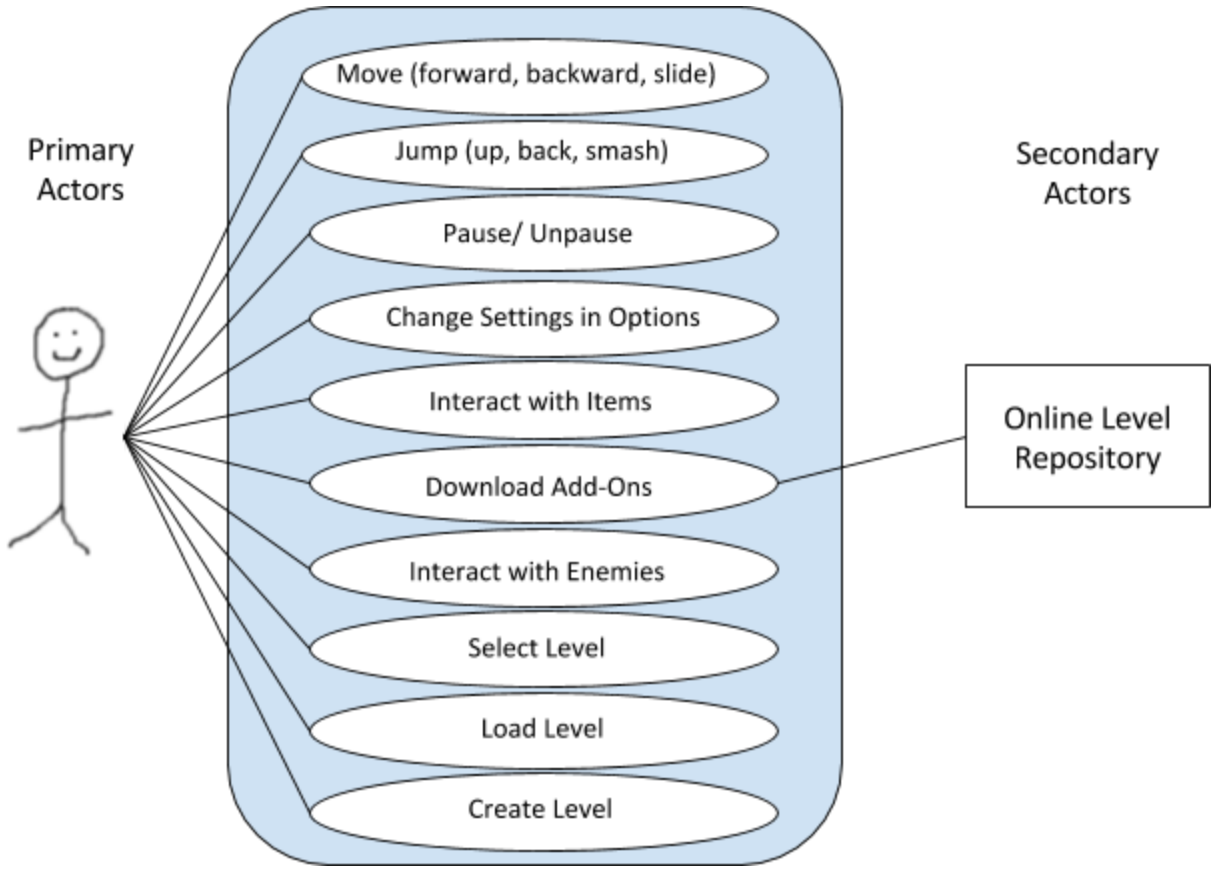
Libraries

The next layer of our architecture is the libraries layer. The libraries layer is home to the premade third party methods that are used in the game engine. These utilities are imported into the game engine to assist with things such as math and file I/O.

OS Abstraction Layer

The final layer of the architecture is the OS Abstraction layer. This layer allows the game to be played on different platforms such as Windows, MacOS, and Ubuntu by abstracting the differences between APIs across the target platforms. The higher level rendering is then built on top of this lower level abstraction.

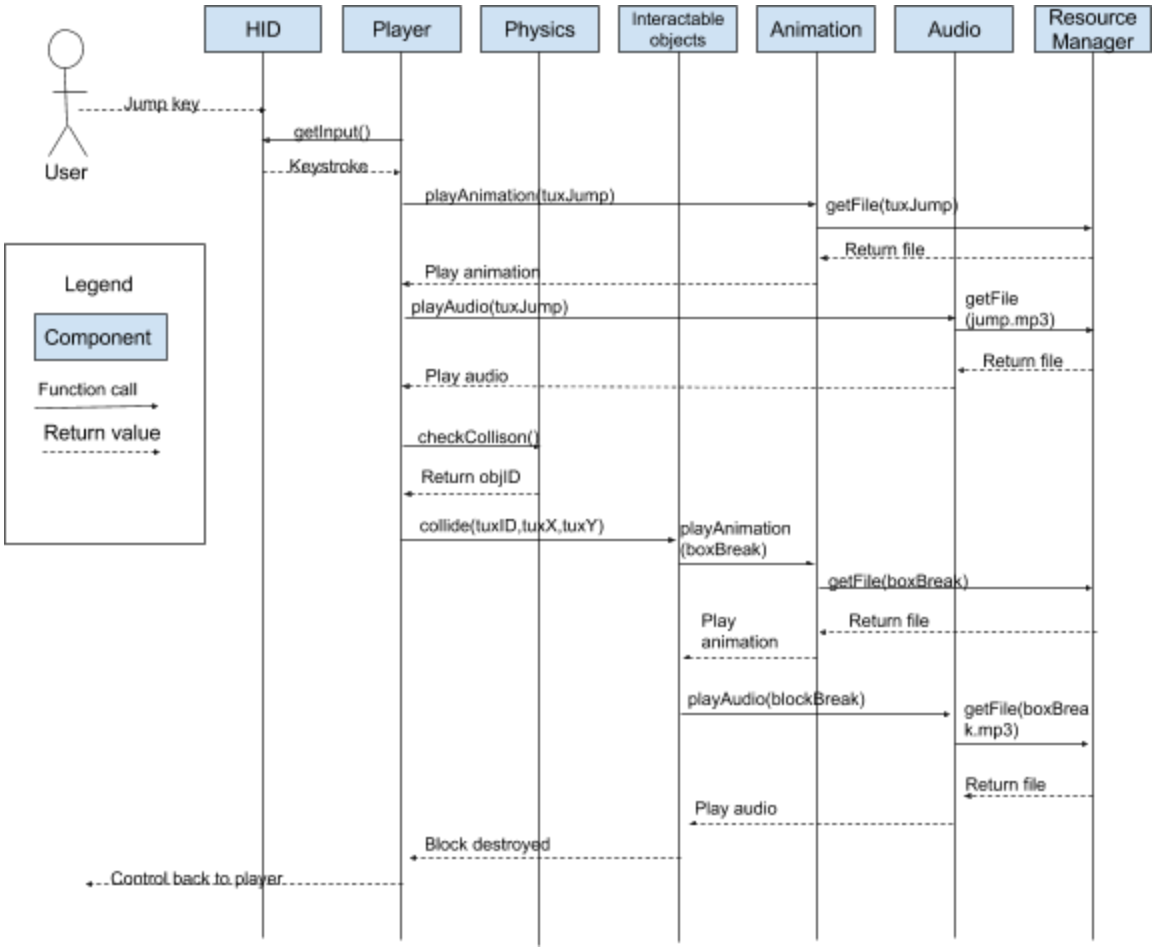
Use Case Diagram



The use case diagram contains one primary actor; the player. As the primary actor, player is able to start actions in the game, as well as actions for navigating the menus and managing game add-ons. While playing the actual game, the player is able to control Tux's movements, make him interact with items and enemies, and pause the game. Outside of the actual game, the player is able to navigate the menus, changing the settings in options, creating new levels, moving Tux around the level map, choosing a level to play, and requesting to download add-ons to their game.

The online level repository is the only secondary actor. Unlike the primary actors who are able to start actions in the system, secondary actors are only able to complete actions. When the player starts the action "download add-ons", they send a request to the online level repository servers to download specific add-ons and user made levels. The online level repository completes that action by sending the requested add-ons and levels to the player's computer for them to install in their game.

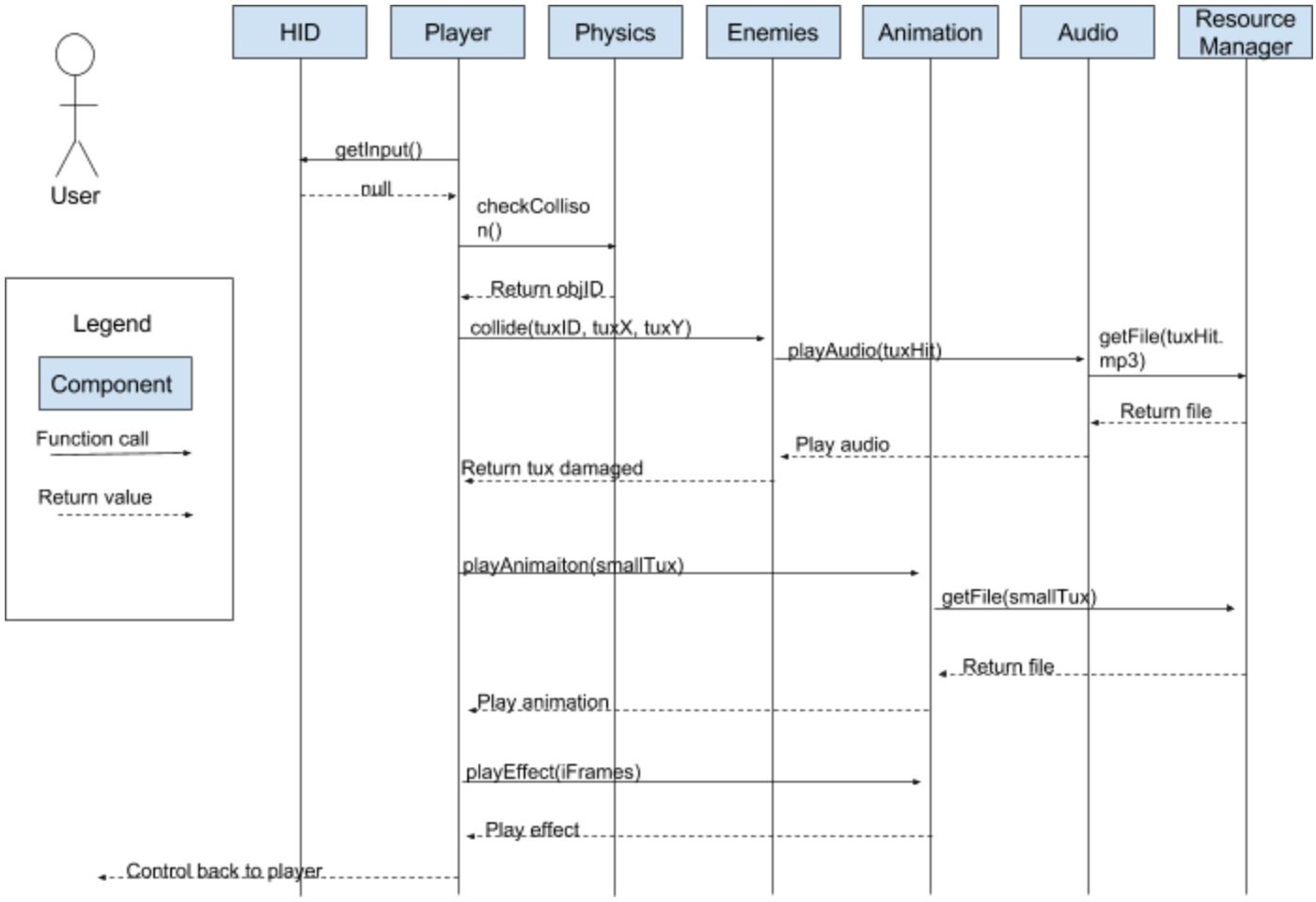
Sequence Diagram #1



Sequence diagram for "Tux jumps and breaks a block with his head"

In this sequence diagram the Tux jumps and breaks a block with his head. First the player presses the jump key. At the start of every frame, the player object polls the HID to get the current keystroke. The player object processes this data and determines the user wants to jump, so it tells the audio and animation to go play the tux jumping sound and animation. They do this by asking the resource manager for the respective files and then processing it. After this is done, the player component then checks the physics engine to see if it has collided with anything. The physics component returns that it did and sends back the object ID of the object it collided with, in this case, a block. The player object then calls the collide method in the objectID sending it the ID of the player along with the current X and Y coordinates that Tux is currently at. The interactable objects component then calculates what happens based on its current position and Tux's and realizes that since Tux hit the box from the bottom, the box should break. So it plays the animation and audio in the same way that the player object did for jumping. After this is all done the interactable objects component tells the player object that it is all done, then the player object returns control back over to the user for their next input.

Sequence Diagram #2



"Mr. Iceblock walks into big Tux standing still and causes him to become small Tux"

In this sequence diagram the bad guy "Mr. Iceblock" walks into Tux while he is standing still and causes him to become small. The user has no part in this this case since there is no input given to Tux. After the player gets null back from the HID it then checks collisions to see if anything has collided into it. The physics engine returns the ID of the enemy that walked into Tux. The player component then invokes the collide method in the enemy object, send over Tux's ID, Tux's X and Y coordinates to the enemy. It determines that the two objects collided head on and that means Tux should take damage, and plays the corresponding sound in the same way in the first diagram. It returns this information back to the player component. With this new information the player object first reduces its current powerup state by 1, checking to see if it is a game over or not. In this case it just transitions large Tux into small Tux, changing the sprites via the animation component. The player object then would call some function to turn on the invincibility frames for a couple seconds so the player has some breathing room. The player object will then tell the animation component to play an animation effect over Tux's current animation, in this case it's to cause the spire to blink on and off to show to the player that Tux is invulnerable for a few seconds.

Conclusion

Overall, we believe our hybrid architecture of layered and object oriented styles best suits SuperTux's design. We were able to come to this conclusion through reference architectures, game documentation, and by simply playing the game.

Through our object oriented and layered approach we are able to achieve a high cohesion and low coupling, making our high level architecture easy to read and understand. Additionally, our architecture allows for reusability whether it be for the reuse of layers in other projects, or the reuse of objects through polymorphism and abstraction. Each component has at most two other components that depend on it, meaning that a modification to any module will require very little changes to its dependants. This allows for easier modification and maintainability which is crucial in an open source game.

As this is our conceptual architecture, it is what we believe to be an ideal structure. Our future work will recover the concrete architecture of SuperTux.

Lessons Learned

The main lessons we learned were what conceptual architecture means and how to know when our conceptual architecture is as good as it can get. When working on the assignment, even though we were not able to look at the source code for the game, we felt that our conceptual architecture had to perfectly resemble its architecture. We spent copious amounts of time looking at resources that could potentially help us determine the actual architecture of the game. What we later realized, is we were trying to recreate the concrete architecture, rather than the conceptual architecture. The conceptual architecture is like the blueprints of a house, it is an abstract structure that details the interconnections of components in a large software system. Therefore, it doesn't have to be exactly like what is used in the actual game, and there can be multiple correct structures. From seeing different conceptual architectures during other groups' presentations, we could see how they may have had a different perspective on the system than us, but their architecture were just as correct.

From understanding what conceptual architecture is, we learned how easy it is to start creeping excellence for our architecture design. When we began working on our assignment, we felt that our conceptual architecture had to be perfect before we could continue with the rest of the assignment components. We spent the majority of our time searching for ways to confirm if our design was correct. We continuously made small changes to improve our design, trying to optimize the design we currently had, yet there was no way for us to know if it was perfect without looking at the source code. It was not until later we learned that sequence diagrams are one of the best methods to test the functionality of our conceptual architecture, our design does not have to be perfect, and that many of the assignment components were not dependent on the revised conceptual architecture.

Data Dictionary

Animations - component of the game logic layer, responsible for playing all animation files

Audio - component of game logic responsible for playing all audio files

Camera - component of the game engine, provides the view of the map to the player

Game Engine - second layer of architecture, contains main loop while controls the game state

Game Logic - initial layer of the architecture containing within it an OO structure

Game System - component of game logic layer, home of the core components of the game logic

Human Interface Device - component of the game logic layer, physical I/O device to get user input

Interactive Objects - component of the game system, responsible for holding the information of non-enemy interactables

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

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

Libraries - third layer of architecture, contains third party code libraries that are used in game engine

OS Abstraction Layer - final layer of architecture, allows game to be played on multiple platforms

Physics & Collision - handling object interactions, collisions, and controls the forces

Player - component of game system, holds the information for the player character

Resource Manager - component of game engine, accesses game's resources and assets and retrieves them for the game logic layer.

Naming Conventions

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

References

About. (2009, August 2). Retrieved October 22, 2017, from http://supertux.lethargik.org/wiki/About

Chapter 3: Architectural Patterns and Styles. (n.d.). Retrieved October 22, 2017, from https://msdn.microsoft.com/en-us/library/ee658117.aspx

Main Page. (2016, August 6). Retrieved October 22, 2017, from http://supertux.lethargik.org/wiki/Main_Page

Operating System Abstraction Layer (OSAL). (2006, June 11). Retrieved October 22, 2017, from https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080040870.pdf

SuperTux Documentation. (2014, June 9). Retrieved October 22, 2017, from http://supertux.lethargik.org/development/doxygen/html/index.html

Typical Software Architecture. (n.d.). Retrieved October 22, 2017, from https://academlib.com/22045/computer_science/typical_software_architecture

W. (n.d.). Super Tux. Retrieved October 22, 2017, from https://sourceforge.net/projects/super-tux/files/stats/timeline?dates=2000-05-16%2Bto%2B2017-05-22