Tuxedo Amigos

SuperTux Concrete Architecture
CISC 326 - Assignment #2

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 goes into the derivation process of our concrete architecture for the game SuperTux. This began with a re-evaluation of our conceptual architecture from assignment one after being given access to the SuperTux source code. Going through the source code, we were able to stick with an object-oriented and layered architecture but with many changes applied to the game engine and game logic components, as well as the introduction of the graphical user interface component.

We went through an updated component breakdown for our new conceptual architecture. Subsequently, alternatives for our concrete architecture were deliberated upon before we finally settled on a structure that was essentially entirely object-oriented. We followed this by doing a reflexion analysis and expanding on two of the subsystems in our architecture; game logic and game engine. We then went through two sequence diagrams; one for "player pauses the game" and another for "player unpauses the game." Lastly, we discussed the lessons learned in the process of creating our concrete architecture, such as the importance of having a clearly defined coding structure before beginning any development.

Introduction and Overview

The contents of this report details how we derived our concrete architecture for the game SuperTux. This was done primarily through the Understand tool. With this tool we were able to build graphs and diagrams that allowed use to see the dependencies, flow, and function calls for the SuperTux source code. This tool was integral in allowing us to collate the different directories and files together into a concrete architecture that reflected our ideas for how it should flow and our newly revised conceptual architecture (Features, n.d.).

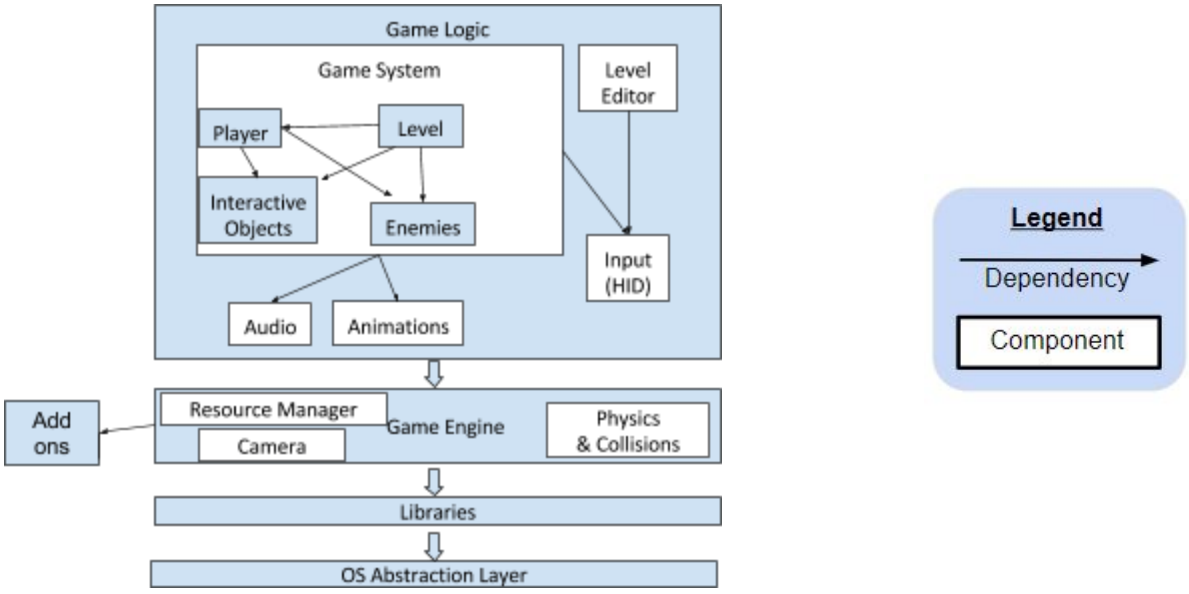
We began by re-evaluating our conceptual architecture after we started to delve into the source code for the game. This resulted in our finalized conceptual architecture, which was the basis for beginning to derive our concrete architecture. We broke down and examined the main components that underwent the most change.

Next we began deriving our concrete architecture, we came up with a couple of alternatives before finally settling on our concrete architecture. With this we were able to do a reflexion analysis between our concrete and conceptual architecture and investigate the unexpected dependencies. Subsequently, we delved deeper into the game logic and game engine components.

With our concrete architecture complete, we were able to create sequence diagrams that reflected specific actions performed in the game. Finally we illustrate the lessons learned during the course of this assignment. Following this is a conclusion going over the main points of the report and a data dictionary that outlines key terms mentioned during this report. There is also a naming conventions section that details any abbreviations used during the course of this report.

Architecture

Derivation Process



Going through the source code and comparing its behaviour to our architecture, our team noticed a handful of redundancies and possible errors with a wide variety of causes and solutions.

We had initially imagined that the Audio and Animation components were used to streamline their process and retrieve the actual files and content from the Resource Manager. However upon investigation we discovered that no such component exists and we were able to remove the Resource Manager entirely and modify the Audio and Animation components to store the files and moved them to the Game Engine layer.

One of the first things we noticed as we began flipping through the code was that there was a component explicitly titled “GUI”. We had initially implicitly grouped GUI into the Game System but after this discovery we decided that it should be a component as to increase the cohesion in Game System.

We also discovered that there was a lot of overlap and bidirectional dependencies between the Player and the Interactive Object components. After looking at a few of the causes we discovered that the two were highly cohesive and could be merged into one component.

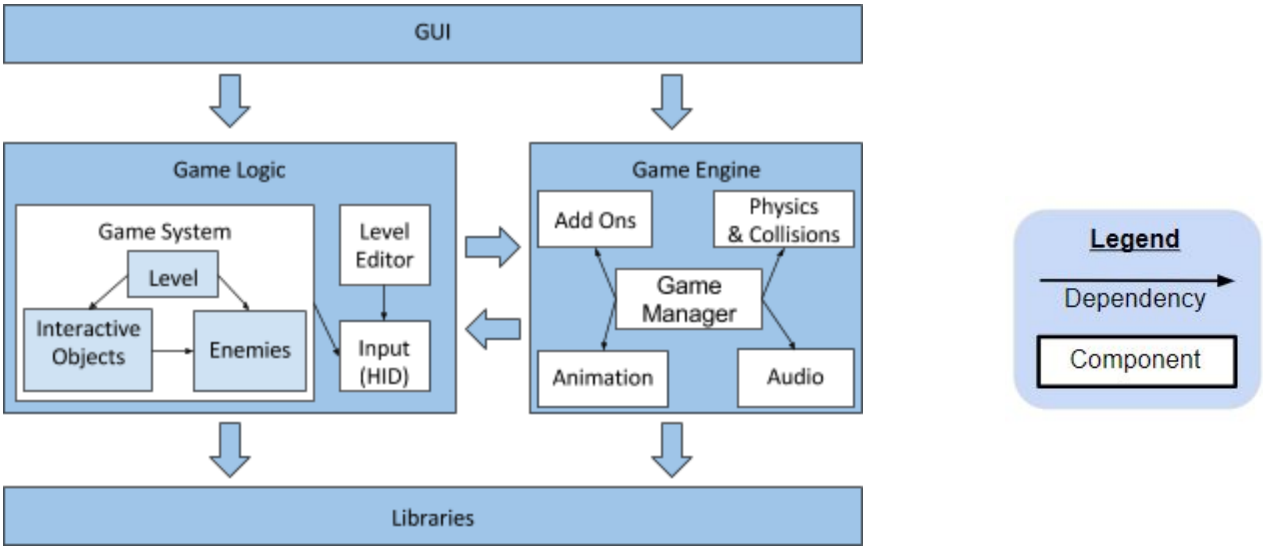
The camera component in our initial architecture was there to help map what should be displayed on the screen. However, as we were categorizing the source code we found that most of the elements in Camera would be a better fit in either GUI or in the Gamer Engine. As a result, we removed the component to lower the coupling of our architecture and sorted the files accordingly.

The OS Abstraction Layer caused a few difficulties for us. We looked carefully into the source code and found that most of the abstraction happens in only one file; supertux/main.cpp. We decided that since this file also contained a lot of other content, we could find a more appropriate home and abolish the OS Abstraction Layer.

We added a Game Manager subsystem to contain the functions within the Game Engine to include all the files that were used to manage the flow of the game. This helped clean up our dependencies and increased the cohesive behaviour of the Game Engine subsystems.

After separating the disorganized chaos that was the source code into what we believe to be the most appropriate components for this game, we realized that the architecture is very likely to be more object-oriented than we had originally expected. Following a lot of deliberation, we were able to finally complete our Revised Conceptual Architecture.

Revised Conceptual Architecture



With all the changes our architecture underwent, it still maintained a hybrid design but definitely shifted slightly away from Layered and more towards an Object-Oriented style. As discussed in the derivation process, our new architecture introduced a handful of new components and made modifications to some of the old pieces. The addition and explanation of the GUI and Game Manager components and the changes to Game Logic and Game Engine will be elaborated later on in our report.

Alternatives Considered

One alternative we considered was including the GUI component in game logic however, this isn't cohesive and created a lot of new unexpected dependencies. We also considered add-ons as its own top-level component as that was what we had in our original conceptual architecture. However, the dependencies of add-ons were very similar as they are for the game engine component, so we decided it was a better fit to place it in game engine.

As well, we considered Animations and Audio in Game Logic as this is what we had in our original conceptual architecture. However, animations and audio are two components which often run concurrently with other processes. Since concurrency is handled in the main loop of the game, which is found in the game engine, coupling is reduced when the components are cohesively placed at the source of the concurrency, the game engine component.

Concrete Architecture

Mapping Rules

Throughout the process of mapping the source code of SuperTux, we followed a variety of steps. Our first approach was to use the folder and file names to help categorize the uses of the files and this was a good place to start. After having mapped most of the dependencies, we looked at the graphical representation created by the Understand software and were relatively pleased with the results. Mapping the SuperTux folder was much more of a challenge. We went through the source code of each file and classified it according to its dependencies and contents. After all of them had been mapped, we graphed the results and cleaned up some of as many of the unexpected dependencies as we could. Some of the reasons we found these dependencies included initially misunderstanding the purpose of the file and some were caused by low cohesion within the file (too many things of different varieties). We cleaned up as many as we could with trying to keep cohesion high and coupling/dependencies low.

Design Patterns

After having investigated the source code, we believe that the design pattern implemented is the abstract factory pattern. Some evidence we were able to find to support this includes the fact that there is a file titled object_factory.cpp containing code that creates abstract objects for later use without explicitly specifying the concrete class. This style is useful because it allows the game to create many objects as it loads the level. It also permits the game to specify what those objects are and when it will need them without creating them on the spot, which would significantly slow down the game and require higher processing power.

There are also hints of the template pattern because most objects in the game have a collision method that gets called when it interacts with other objects. The operation that is being abstracted is the general collision method, which is called when the objects are interacting. Following this, the definite objects are then calling their own collision methods which are no longer abstract and carry out the required operations. With this design pattern, it allows for the physics component to be more general and not require it to explicitly call a specific object's collisions method.

Components

Graphical User Interface

The GUI component handles the drawing to the screen.

Game Logic

The Game Logic component contains three separate components; Game System, Level Editor and Input.

Game System

The Game System is comprised of three components as well; Level, Interactive Objects, and Enemies. The level component acts as a blueprint for each level of the game. The Interactive Objects component stores info about the player state and items the player can interact with. Finally, the Enemies component contains all of the information for the various bad guys found within the game. Game system is dependent on the HID component.

Level Editor

The Level Editor component allows the player to make or modify levels in SuperTux. Is dependent on the HID component.

Input (HID)

The Input component contains input handling such as the keyboard, mouse, and controller. Is not dependent on anything.

Game Engine

Home to the main loop of the game, the Game Engine component controls the current state of the game and consists of five components.

Add Ons

The Add-Ons component allows the user to download additional levels from an online repository. It is dependent on the Game Manager.

Physics and Collisions

The Physics and Collision component is responsible for handling object interaction and collisions as well as forces in game. It is dependent on the Game Manager.

Animation

The Animation component stores and retrieves animation sprite files for the game system. It is dependent on the Game Manager.

Audio

The Audio component stores and plays all in game sounds and music. It is dependent on the Game Manager.

Game Manager

The Game Manager component manages the flow of the game, calling all other subcomponents of the Game Engine component.

Libraries

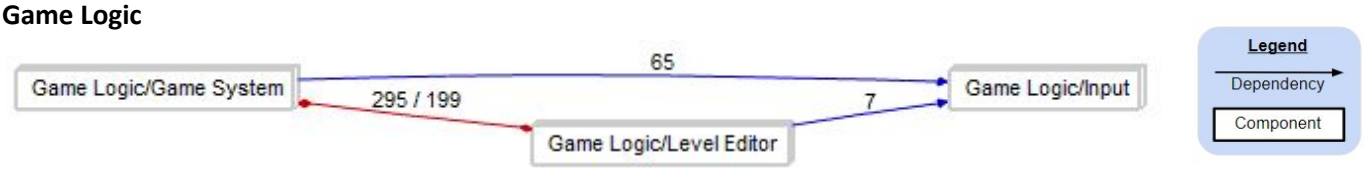
The libraries component stores the premade third party methods that are used in the game engine such as math and file I/O.

Reflexion Analysis

Top Level Systems

Our conceptual architecture was designed to be a mix of layered and object-oriented style, while in actuality, the concrete architecture is solely object-oriented. Although these architectures are fairly similar, there were four unexpected dependencies in the top-level systems of the concrete architecture that caused them to be so different. Firstly, there was a dependency between GUI and Libraries. This is caused by the fact that Libraries contains the math functions which we did not expect GUI to use to calculate the locations of where to place objects in the game. Secondly, there was the two-way dependency between GUI and Game Engine. In the conceptual architecture we expected a dependency between GUI and Game engine, but we did not expect there to be the menu manager files in GUI that are called by the game manager in Game engine. Similarly we did not expect the two-way dependency between Game Logic and GUI. In the conceptual architecture we expected the mouse cursor to be stored in the input component of the Game Logic subsystem. While in the concrete architecture, the mouse cursor is stored in GUI, where it then sends info to Game Logic when there is an action. Lastly, we did not expect the second dependency between Libraries and Game Engine. There is a file in Libraries that records the current state of the game to keep track of what is going on in the game. Therefore whenever this file is called, it access the gameConfig file in Game Engine to receive information on the current game state.

Game Logic



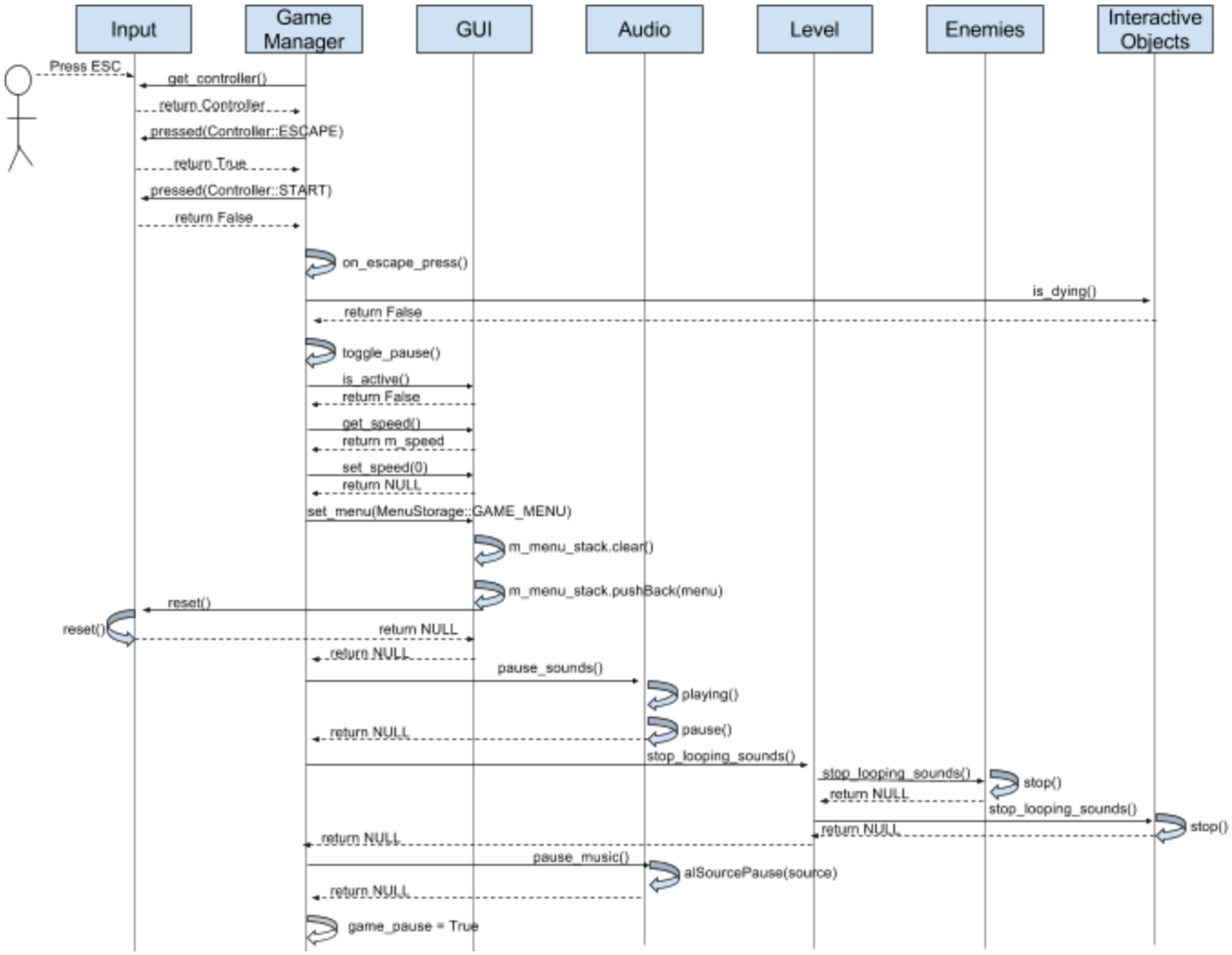
The Game Logic subsystems in the concrete and conceptual architectures had very few differences. The main difference we observed was a two-way dependency between the Game System subsystem and the Level Editor, where we expected there to be no connection at all. This dependency likely exists because the Level Editor will need information information on the pieces from Game System for the user to edit the levels, and Game System needs to talk to the Level Editor to receive new levels that are made, or changes that are made to pre-existing levels. We also did not expect there to be two-way dependencies between all of the components in Game System. These dependencies likely exist because of the open source development of the game. In the conceptual architecture, we envisioned these components to be more hierarchical, but with open source development it would be difficult to uphold a strict architectural style in such a commonly used subsystem of the software. Therefore instead, a lot of these developers likely opted for quick and efficient methods to implement their improvements to the game, creating a lot of these extra dependencies between components.

Game Engine

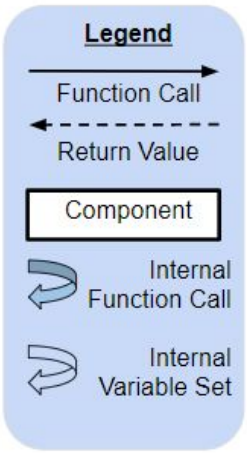
There were only some minor differences between the concrete and conceptual Game Engine subsystems. One being a dependency from Add-Ons to Game Manager, and another being Animation to Game Manager, each to create two-way dependencies. Secondly, the unexpected dependency between animations and the game manager, as well as the unexpected dependency between add-ons and the game manager, are caused by files that depend on gameconfig.hpp in the game manager. Our initial thought was to move the file to libraries, however that would cause unwanted outgoing dependencies from libraries into the game engine component, and it is a more cohesive fit in game manager.

Sequence Diagrams

Pause Game Sequence Diagram



Sequence Diagram 1: Player pauses the game



Explanation of Pause Game Sequence Diagram

For our first sequence diagram we chose to do the player pauses the game. One of the main reasons we choose this was because it was easy to understand but also had a lot of interactivity between the different components and really showed off the object oriented nature of the code.

To start off the sequence diagram the player must press the ESC key. The game manager component handles the input by retrieving the controller object located in the input component via the get_controller() method. With the controller the game manager then checks if the last input was either ESC or START, if playing with a gamepad, which returns true. This triggers the on_escape_press() method inside the game manager, this method initially checks if the player object is dying as the developers didn't want the player to pause as they died as it would cause problems with handling the player's death.

Since the player is not dying the game manager then calls toggle_pause() which runs through the steps required to pause the game. The first thing this method checks is if there is a current menu drawn by invoking the MenuManager object's isActive() method, in this case returns false. So with this information it then gets the speed of the screen from the ScreenManager and stores this in a variable for when the game gets unpaused.

Next the game manager sets the screen speed to 0, effectively pausing all the motion in the game. After the screen is frozen the game manager tells the MenuManager to set the current menu to be the game menu, the menu that shows up when the game is paused. To do this the MenuManager must first clear the the current menu stack.

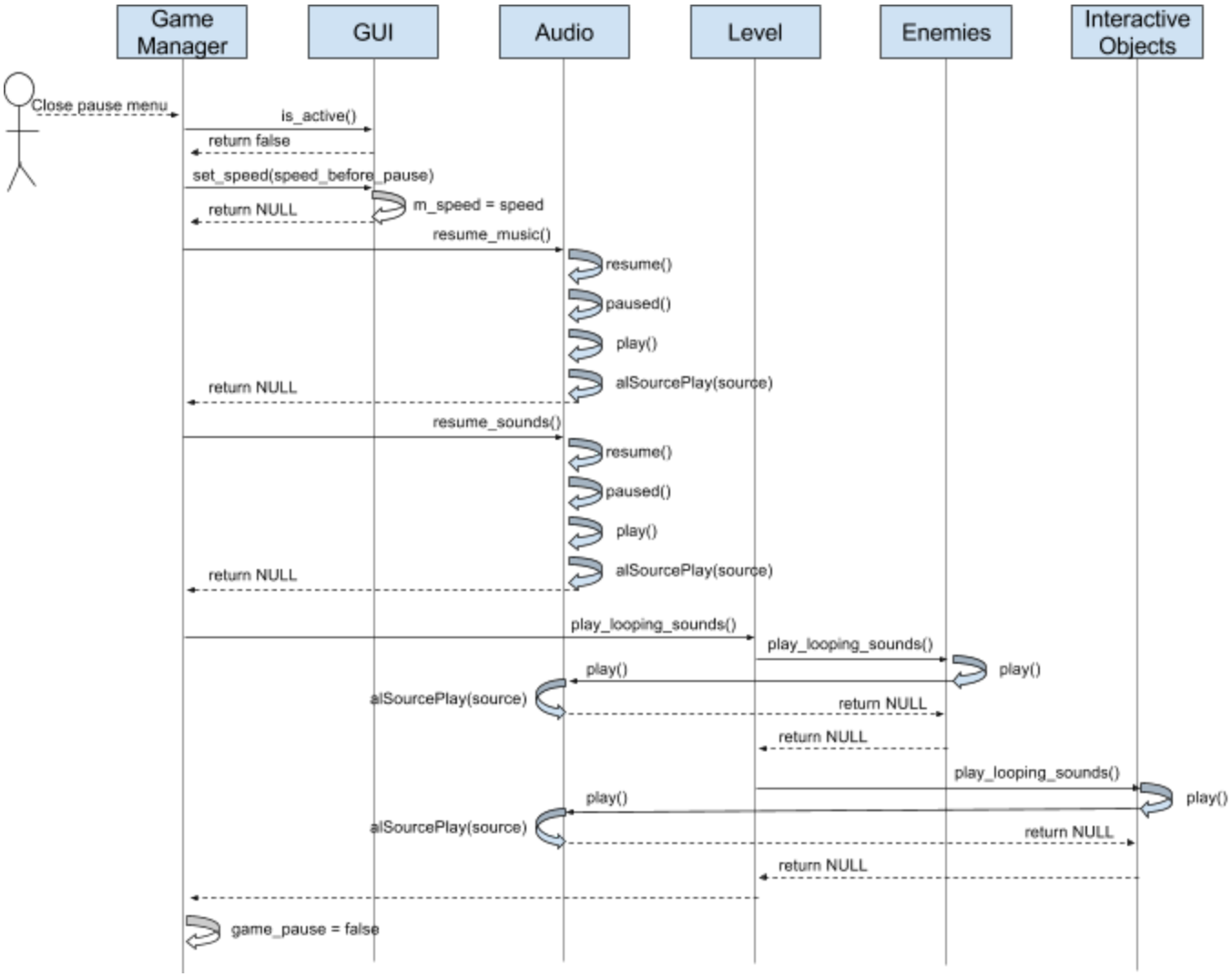
With the stack cleared it then pushes the game menu to the top of the stack, as well as setting flags in the menu so that it will be drawn at the start of the next frame. After the menu is set to be drawn the menu manager calls for the input to be reset, to handle the switch from the user using keyboard to using a mouse.

With this all done the game manager's next task is to stop all the sounds from from playing. To accomplish this it calls the SoundManager's method of pause_sounds(). This method checks if there are currently playing, which there is so then it calls the pause() method, temporarily stopping the sounds.

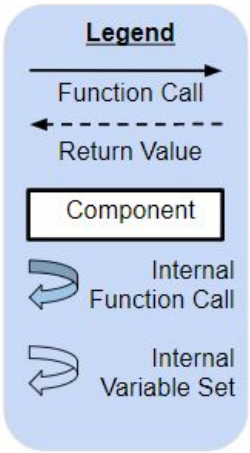
With this done the game manger then talks to the sector object, which resides in the level component and represents the current chunk of the level that is being loaded and the player can see. Within the sector the level component loops through all enemies and interactable objects calling the stop_looping_sounds() method which in turn calls the internal stop() method, pausing the sounds.

With all of the objects and enemies no longer emitting sound the final step is to pause the background music. This is simply done by having the game manager to call the pause_music() method in the SoundManager, which simply calls the alSourcePause(source) method, pausing the music. Finally the game_pause boolean is set to true and the game is fully paused.

Unpause Game Sequence Diagram



Sequence Diagram 2: Player unpauses the game.



Explanation of Unpause Game Sequence Diagram

Our second sequence diagram is the player unpausing the game. This event is triggered when the player closes the pause menu. In the update method of game_session.cpp there is a check to see if both the game is paused, and if there is no current menu menu active. This is done by invoking the MenuManager method isActive(). If there is no active menu but the game is still paused then that means the player must have closed the menu and wants to resume play. So to do this the game manager must first reset the speed of the screen. It calls the set_speed method of the ScreenManager giving it the speed that was saved when the game was being paused.

Next to do is resuming the music and sounds of the game which are done in the same way of invoking a method in the SoundManager to resume playing. To do this the SoundManager must first check if the music is paused, and since it is it calls the alSourcePlay(source) method which calls the OpenAL functions to start playing the sounds again.

With the basic audio resumed the game manager moves on to turning on all of the object and enemy sounds within the current sector. Much like with pausing the game, the sector object loops through all the interactable objects and enemies and one by one resumes the audio by first calling the play_looping_sounds() method which causes the object to communicate with the audio component to once again call alSourcePlay(source) to resume the playing of sounds.

When this loop is completed the game_pause boolean is returned back to a false state, and the player is allowed to resume play of SuperTux.

Lessons Learned

Overall, we learned a lot about the concrete architecture of SuperTux and about good practices for software development. Throughout this project, we obtained a better understanding of the Understand tool and discovered its value not only for this assignment, but for future use in the software industry. Understand is able to quickly analyze source code and display dependencies between components. When a project is halfway through the development cycle, being able to easily derive the concrete architecture is valuable for contributing code that aligns with the current dependencies.

With this, we also realize the importance of having a defined structure at the beginning of the software development process. The architecture should be considered during all of the other phases of development such as design, implementation, and maintenance. We discovered when looking through the SuperTux source code that it did not have a clear structure and we found many unrelated files together in the same folder (supertux folder for example). Additionally, after deriving our concrete architecture we saw it was highly coupled. Being an open source project, the game may have had a defined architecture at some point but this original vision may have been altered over time. This could be a result of the addition of new features that were not initially considered, or potentially due to new contributors who may have had their own interpretation of the architecture or simply just chose to bypass dependencies.

Finally, we learned the importance of documenting as you go in terms of source code documentation as well as documenting our progress throughout this assignment. Our group faced many limitations as a result of the lack of source code documentation. Trying to map some of the files to our components was challenging because there was minimal comments and we often struggled to figure out what the code was doing. Not only is this limiting to us, but it also limits new developers who want to contribute to the project. Additionally, we learned the importance of documenting our progress as a group throughout the assignment. This would have helped us when talking about our derivation process afterwards and would have also saved us time when we had to remap all of our files again after some technical issues.

Conclusions

In conclusion, with the help of the Understand tool we were able to derive the concrete architecture for SuperTux. We found that the concrete architecture differed greatly from our conceptual architecture. While our conceptual architecture was a layered and object oriented hybrid, the concrete architecture was primarily OO. Additionally, we were surprised by the high coupling SuperTux had. Through code inspection, we were able to refine our concrete architecture to the best of our ability but were still left with a great amount of unexpected dependencies. Ultimately, being an open source game it is difficult to maintain and follow a solid structure.

In the future, we will look at the introduction of a score system and analyze how it will affect the game's concrete architecture.

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
OSAL : Operating System Abstraction Layer
P&C: physics and collisions
TA: Teaching Assistant

References

Features. (n.d.). Retrieved November 11, 2017, from https://scitools.com/features/