CISC 326 A1
Conceptual Architecture: SuperTux

Group: TBA

October 22nd 2017

Instructor: Ahmed Hassan

Coco Chen	14kxc@queensu.ca
Yihao Chen	15yc9@queensu.ca
Yuhao Chen	14yc37@queensu.ca
Brayden Dewar	13bad2@queensu.ca
Lena Krause	14lk6@queensu.ca
Selin Onsoz	13bso@queensu.ca

Abstract

The purpose of this report is to determine a conceptual architecture for the game *SuperTux* and describe the process of how the team came to conclusions. It was determined that the highest level subsystems in *SuperTux* are connected in a layered style while connections between subsystems in layers exhibit object oriented qualities.

The team was able to conclude that the conceptual of *SuperTux* is a hybrid of Layered and Object Oriented. The highest level layer encompasses four main subsystems; user interface, game specific subsystems, resources, and a hardware abstraction layer. Data flows from the highest layer of the user interface subsystem to the lowest layer hardware abstraction subsystem. The resources and game specific systems subsystems have lower level systems which interact with each other – giving the architecture object oriented qualities. Examples of the specific use cases of a user controlling the character Tux to activate a power up block are dissected in order to show lower level data flow and function calls between subsystems.

The benefits of the conceptual architecture are high levels of cohesion in subsystems and the ability to implement a highly functional user interface to facilitate fun gameplay. Concurrency is handled appropriately so that the architecture's many resource subsystems and user interface can work together for a smooth user experience. Throughout the development of the conceptual architecture the group learned lessons about game architecture and group work that will aid them in future assignments together.

Contents

Abstract 2

1.0 Introduction 4

2.0 Derivation Process 4

3.0 Architecture Overview 7

3.1 User Interface 8

3.2 Game Specific Subsystem 8

3.2.1 Game States 8

3.2.2 Game Data 9

3.2.3 Add-on Repository: 9

3.3 Resources 9

3.3.1 Physics Engine 9

3.3.2 Graphics Engine 10

3.3.3 Audio Engine 10

3.4 Abstraction Layer 10

4.0 Sequence Diagram analysis 11

4.1 Sequence Diagram I 11

4.2 Sequence Diagram II 12

5.0 Concurrency 12

5.1 Audio and Graphics 13

5.2 Graphics and Gameplay 13

5.3 User Input and Gameplay 13

6.0 Lessons Learned 13

7.0 Conclusion 14

References 15

1.0 Introduction

Before work on the assignment began, the eight different architecture styles were discussed in class. Architecture styles are blueprints for how to structure code in terms of what features of a software are grouped together in subsystems and how these subsystems interact and depend on each other. The architecture styles are pipe & filter, layered, object-oriented, repository, client server, implicit invocation, and process control. All eight architecture styles have their own unique advantages and disadvantages, making it important to think carefully about which architecture style most appropriately suits the software to be implemented. The goal of this assignment was to study the game SuperTux and design a conceptual architecture that would suit the game using the information presented in class and individual research of software architectures and SuperTux.

The developers of SuperTux describe the game as “free classic 2D jump’n run sidescroller game in a similar style to the original Super Mario games”(supertux.org). The game is free to download and play, and supports offline play. Its code is open source and anybody can contribute to the game, as while it has been released since 2003, it is always in development. SuperTux is released in “milestones” that the developers set for themselves to ensure ongoing enhancement and quality of the game. The SuperTux team is currently working on milestone 3. In addition to the main gameplay, SuperTux has a level editor mode where users can create their own levels which they can submit to be used in the story mode, allowing users with non-technical skills to contribute to game content. SuperTux is coded in C++ and can run on many different platforms including Windows, Android, Linux, and MAC OS.

2.0 Derivation Process

Deriving the conceptual architecture involved playing SuperTux, using reference architectures, and reading the many resources regarding SuperTux’s long development history.

Playing the game allowed for familiarization with the game mechanics and how the game expects users to interact with the game. After experimenting with SuperTux the team was able to construct a user-case diagram to describe interactions between the player and the game.

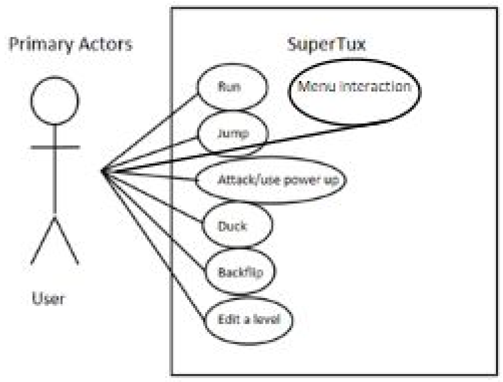

Figure1: User case diagram

The user-case diagram allowed the team to see that there are two main in game modes that the user is expected to play. The first is the story mode, where the player navigates through a level, collecting coins, and killing enemies on their way to a goal. The second is the level editor mode, where the user can create their own levels using a click-and-drag interface. The conclusion from these findings was that somewhere in the architecture there should be two distinct gameplay subsystems – a subsystem for playing levels and a subsystem for editing levels. A user interface subsystem was also determined to be fundamental as this allows the user interactions with the game. This was a starting point, as it had yet to be determined how these subsystems would interact.

A reference architecture is used as a general software architecture for a specific category of software. The below software architecture is a reference architecture targeting video games, one that can be used for video games of different genres.

This reference architecture displays a general architecture for a game engine. The architecture style of this reference architecture is layered – data flows from the higher level subsystems to the lower level subsystems. Studying this reference architecture allowed the team to learn two key things that aided in the construction of SuperTux's conceptual architecture:

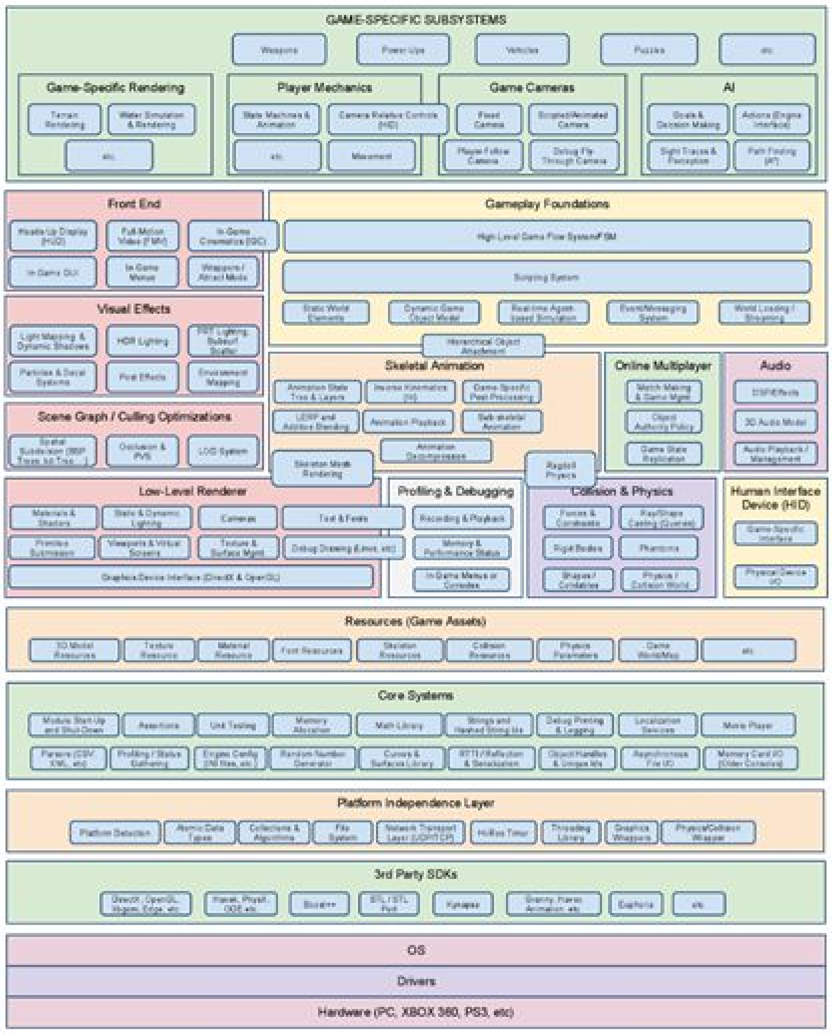

Figure2: Reference architecture diagram “Game Engine Architecture”

1. Game architectures are frequently layered style
2. The types of subsystems that are usually present in the architecture of a video game.

Studying this architecture facilitated the understanding of interactions between the subsystems that were determined from playing the game and the user-case diagram.

The initial architecture that was proposed was primarily a layered style. Limitations that we found with this architecture was that it was unclear how the game specific subsystems interacted with the user, while being too specific about how the in game objects interact with each other. It was decided that an improved architecture would need to be created to clear up any ambiguity about user interface and create an all-encompassing ‘game state’ subsystem which would allow for higher cohesion among the in game features.

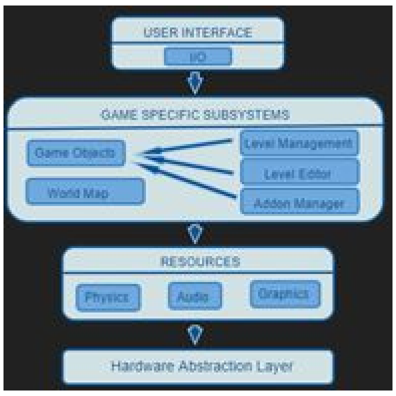

Figure 3: Initial Conceptual architecture proposal

3.0 Architecture Overview

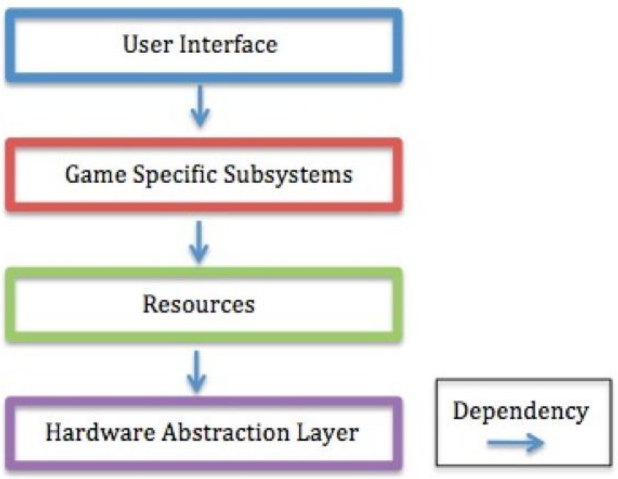

Figure 3: High level layers of the SuperTux

SuperTux is a mix of layered and object-oriented style system that is shown in Figure 3. The relationships between the major components in the system are shown in Figure 4. There are four main layers in the system; User Interface, Game Specific Subsystems, Resources and Hardware Abstraction Layer. The arrow between each two layers shows the dependency. For example, Game Specific Subsystems are depends on Resources so that it can refer to the data in Resources.

The User Interface returns and displays the result on the screen from the Game Specific Subsystems. The Game Specific Subsystems layer consists of "Game Data", "Game States" and "Add-On Repository" components. This layer acts as the core of the SuperTux, it decides how the game runs in progress and outputs into the User Interface layer. The Resources layer consists of "Physics", "Audio", "Graphics" and "User Input" components. It manages the game recourses, user input references and real world physical rules to support the Game Specific Subsystems. The Hardware Abstraction Layer makes sure that SuperTux can run on multiple platforms.

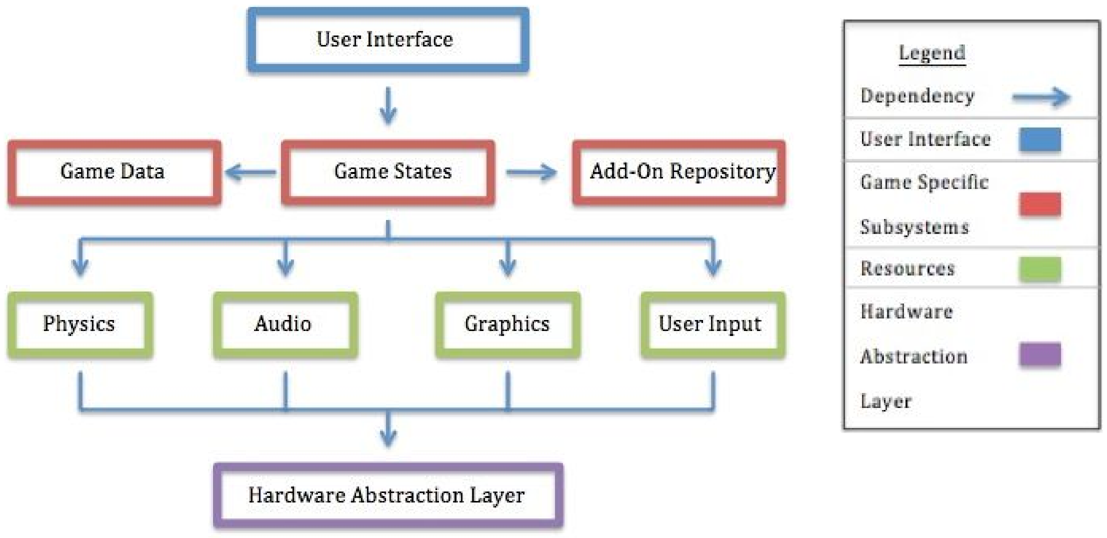

Figure 4: UML diagram of SuperTux

3.1 User Interface

The User Interface layer means by which the user and SuperTux game system interact, and manages settings throughout the game. It gets all the results from the Game Specific Subsystem layer and outputs everything about the game that the user can perceive to the screen, such as graphics and audio.

3.2 Game Specific Subsystem

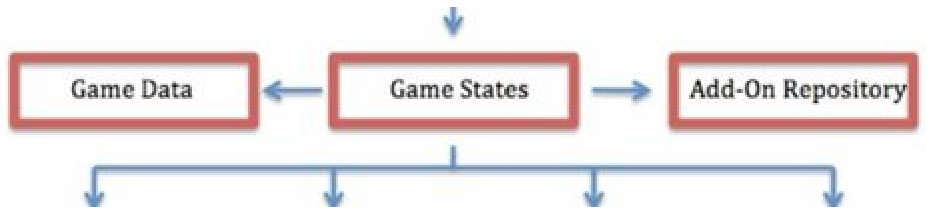

Figure 5:Subsystem dependency diagram

This subsystem depends on the user input. The job for a person who plays the game is only to press the keyboard and moving the mouse. However, how can these actions influence the game to let the tux move or do some other actions such as adding a new language package?

The answer is calling the functions in game-specific subsystems. It can be broken down into three parts; Game State, which contains the main ideas of the game and responsible for the rules of the game; Game data, which contains the functions, data and record object; and Add-on repository, which is a storage of the language package of the game.

3.2.1 Game States

Game States sub-system consists of game logic, current state and level editor.
Game logic is the main logic of the game. Players have to follow the rule of the game. They cannot control the tux to do something that not allow by the logic. For example, the tux cannot fly, and the bad guys cannot speak. Current state is the current status. Level editor is an editing tool which can let the player create their own level to the story.

Depend on Game Data
If the system receives a moving instruction (bases on the logic) from user interface, one of the game object in Game data will be called to move the tux. Another example is for the current state, which is also better for understanding this state. Current state is just a state that record nothing by itself. It needs to call the record object in game data to get the record and form a current status.
Depend on Add-on Repository
The state will be changed, if the language package is changed.

3.2.2 Game Data

In a game, every object is a subset of game data which means that all the stuff such as the tux, bad guys, button, level, record object, world map and so on inherit from this big container (just like a list). The functions such as movement and power up also belongs to this storage. Overall, game object bases on the source code of SuperTux.

The record object:

It is a repository for the game recording. It records the status of the player, the number of coins you get, the level you reach and everything that need to be recorded. The data of current state will also be saved here.

Power up function:

It is a function which can give power to the tux.

Moving function:

This function is responsible for the movement of all stuffs in the game. It also provides a position detection.

World Map:

The map of the game. Player can see the storyline on it.

3.2.3 Add-on Repository:

This is an online system which allows players to add some additional elements into the game, such as the language package. It also contains the levels which are created by other people and allows players to download them.

3.3 Resources

The resources layer primarily includes systems known as middleware. In game development, middleware is typically defined as anything provided by a third-party. This means middleware is not programmed by the game developers, but programmed by someone else. The game developers integrate these systems in because they perform specific, important roles, and often times are very complex with a higher processing power than any other part of the game. Using third-party software allows game developers to focus on developing content specific to their game.

In this architecture, the Game Logic system relies heavily on this middleware for lower level, intensive computations.

3.3.1 Physics Engine

A physics engine's purpose is to handle any calculations the game requires concerning the laws of physics. The Game Logic system requires this engine to calculate things like acceleration and deceleration when moving, the effects of gravity to movement and player location when jumping and falling, and handles collision detection and any resulting pushback movement on the object or objects that collide.

3.3.2 Graphics Engine

More commonly known as the renderer or rendering engine, this engine essentially draws the 2D image of the game onto the computer screen. Game Logic calls on this every single frame per second to redraw the game as things move and change. Because this is a simple 2D game, things like shading and reflection do not need to be taken into account so graphics aren't as intensive as they would be in 3D games. The rendering engine needs to keep in mind things like overlapping sprites, objects and images, and when these things are animated and move to different positions every frame.

3.3.3 Audio Engine

The audio engine is also called by the Game Logic to output the necessary audio files to the machine's speaker, and when it needs to do so to keep in sync with what is being shown on screen by the graphics engine.

3.4 Abstraction Layer

As stated earlier, the game *SuperTux* can run on Mac OS, Windows, Linux and on mobile Android operating systems. Each of these operating systems have some fundamental differences that can make software development complex. Many different types of software around the world are only compatible with a certain number of operating systems as a result.

The abstraction layer’s purpose is to act as an interpreter between the game and the operating system and its subsequent hardware. This way game developers don't need to take operating systems into much account when writing the core of the game, but rather have a system specifically in charge of handling such matters so they can reuse the higher level systems in the architecture for each different platform.

4.0 Sequence Diagram analysis

4.1 Sequence Diagram I

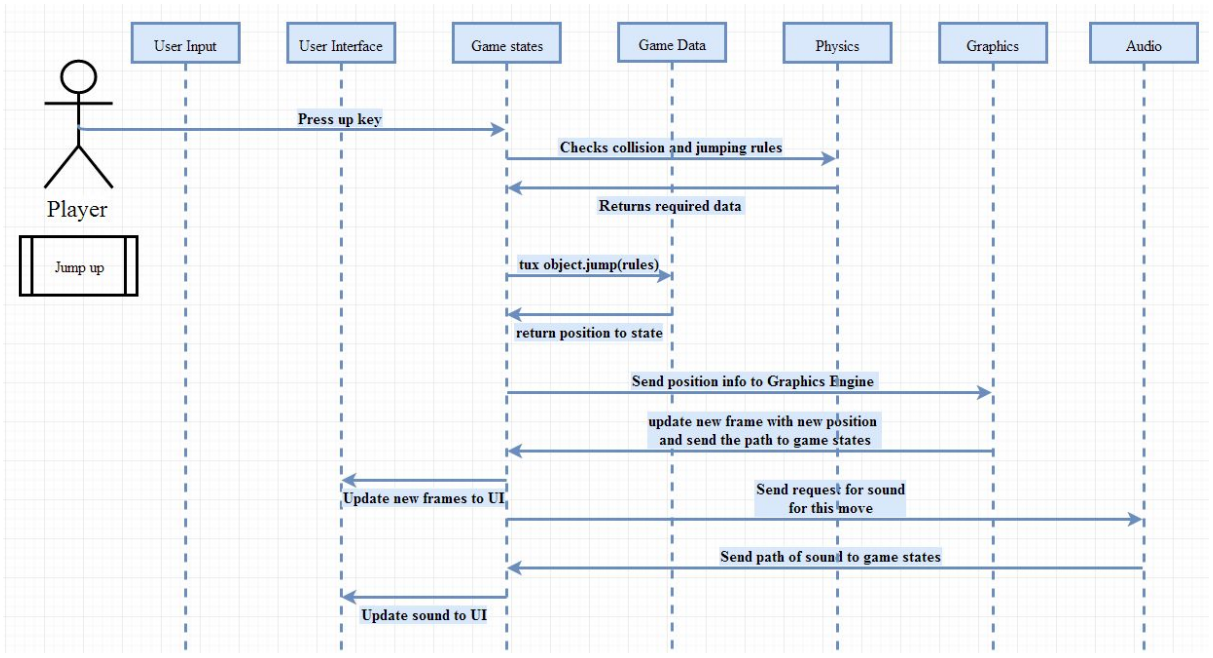
User Input
User Interface
Game states
Game Data
Physics
Graphics
Audio
Player
Jump up
Press up key
Checks collision and jumping rules
Returns required data
tux object.jump(rules)
return position to state
Send position info to Graphics Engine
update new frame with new position and send the path to game states
Update new frames to UI
Send request for sound for this move
Send path of sound to game states
Update sound to UI

Figure 6: Seq1. (TUX jumps up)

As is shown in figure Seq1, the sequence of jumping up is explored.

First, by pressing the up key from an input device, the game states subsystem receives the command and forwards a checking request to the physics engine. The physics engine will be responsible for producing a return data set that contains the rules for jumping and collision detection. Based on the rules acquired from physics engine, the game states subsystem will then call the tux.jump() function from game data subsystem using the rules as parameters, getting the resulting position of tux as the return value.

After this, the graphics and audio subsystems will respond to the request from game states by producing required graphical and audio effects and their reference paths. Finally, user interface will be updated with new frames and sound effects via the paths provided by the game state.

4.2 Sequence Diagram II

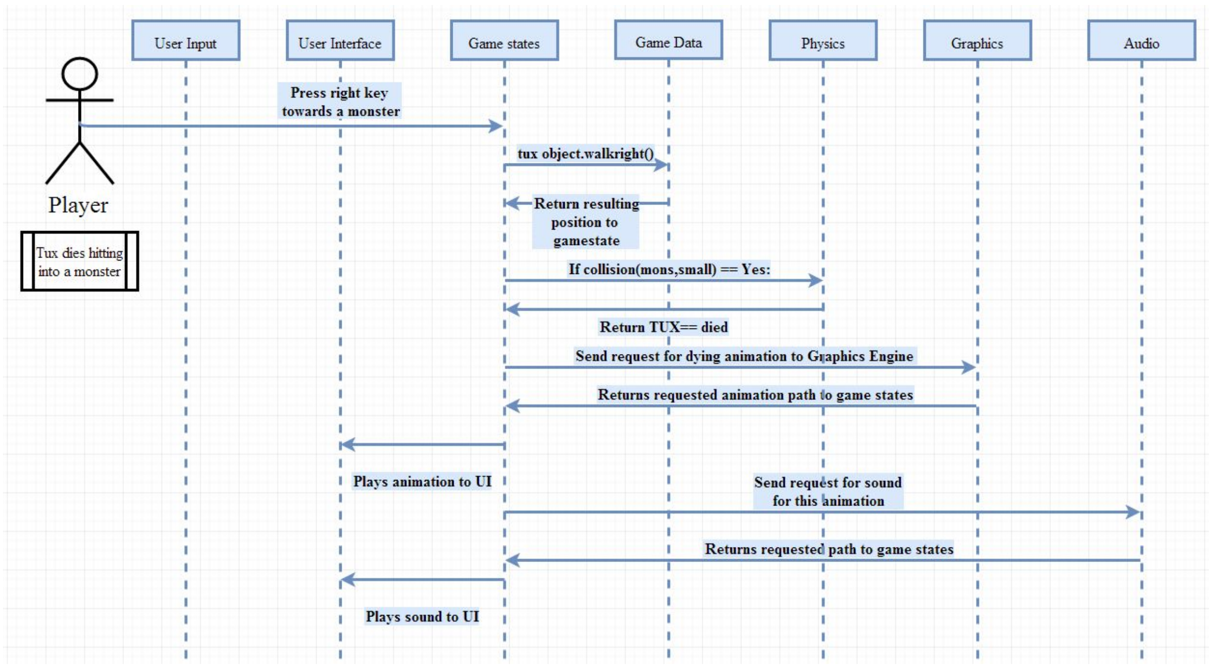
User Input
User Interface
Game states
Game Data
Physics
Graphics
Audio
Player
Tux dies hitting into a monster
Press right key towards a monster
tux object.walkright()
Return resulting position to gamestate
If collision(mons,small) == Yes:
Return TUX== died
Send request for dying animation to Graphics Engine
Returns requested animation path to game states
Plays animation to UI
Send request for sound for this animation
Returns requested path to game states
Plays sound to UI

Figure 7: Seq2. (Player dies running into a monster)

As for figure Seq2, this sequence diagram explains how a tux dies from running into a monster.

Followed by a user input, which is pressing the right key, game states will call the walkright method inside of the tux object from game data, resulting a position update to the tux. After this, a request for collision checking from game states is going to check the collisions for this move. Resulting value indicates that the tux has collided with certain monster and will end up to be dead.
The graphics and audio engine, again, returns the paths for the dying animation and sound of tux upon requests, which are from the game states subsystem. The animation and sound will be played via the user interface after the paths being delivered to game states successfully.

5.0 Concurrency

For games such as SuperTux, there are some concurrencies as they are inevitable in the gameplay. Increasing concurrency is encouraged as this will increase the performance of the game. (Ref –google books, p473/4) This can be done by grouping together systems together, for example by event-driven (ex. gameplay) and data-driven (graphics and physics).

For our architecture we have found three concurrencies between: audio and graphics, graphics and gameplay, and, user input and gameplay.

5.1 Audio and Graphics

The concurrency between audio and graphics is that they complement each other, and so they need to be played at the same time, without delay.

5.2 Graphics and Gameplay

During the gameplay, the game will be calculating elements and will be displaying the resultant graphics on the screen.

5.3 User Input and Gameplay

For the gameplay of SuperTux, user input is required to generate output and to update the game state, so no delay is tolerated between UI and gameplay.

6.0 Lessons Learned

Working on the conceptual architecture of SuperTux, our team has learned some lessons both about the game and how the team should be working together.

First of all, as a team, most members weren't familiar with the game, however everyone focused on their research before playing the game at all. After when everyone was familiar with the game, we had were able to see things more clearly, which helped us come up with our game architecture solution. While doing so we have learned that layered architectures were common in video games, which resulted in the architecture being a mix between a layered and an object oriented architecture. Finally, we have found that brainstorming together and talking about our ideas were more helpful at times than doing research individually.

We realized, in the future, we should have more meetings to discuss our ideas and have multiple meetings to do so.

7.0 Conclusion

The most significant takeaway from this report is the team's conceptual architecture. A mixture of two architectural styles were utilized; object-oriented and layered. Object-oriented style was used because 2D platformers like SuperTux have object oriented programming (this was also seen in the game's actual development logs), and the way the different systems interact and depend on each other in a hierarchical manner. A layered architecture was also incorporated, as this architecture style is frequently used in video games, and the dependencies of the systems naturally fit in a layered style. The combination of these two architecture provides a very clear idea of how the game operates with high cohesion, and allows for systems to be added or altered easily.

The user interface is what the user sees on the computer screen, hears through their speakers, and what they interact with. It depends primarily on the game state, which handles all logic and events of the game. The game state runs the menu, sets up and runs levels and the world map, and can run the level editor. This is done by retrieving game specific assets from Game Data (levels and their content, player stats, etc.), checking for user input, and relying upon third party engines to handle specific, computationally intensive information. Finally, there is an abstraction layer that acts as the interpreter/translator between the core of the game and the different operating systems it can run on.

References

Application_programming_interface. (n.d.). In *Wikipedia*. Retrieved October 22, 2017, from

https://en.wikipedia.org/wiki/Application_programming_interface

Game Object. (2010). In *supertux.lethargik.org.* Retrieved October 22, 2017, from supertux.lethargik.org/wiki/Game_object.

Gregory Jason. (2009). Chapter 1.4-1.7. In *Game Engine Architecture*. Retrieved from

https://326tba.weebly.com/uploads/1/1/2/4/112445829/game_engine_architecture_1.4_1.7.pdf

Middleware. (n.d.).In *Wikipedia*. Retrieved October 22, 2017, from

https://en.wikipedia.org/wiki/Middleware

SuperTux Wiki .(2016). In *Github*.Retrieved October 22, 2017, from

https://github.com/SuperTux/supertux/wiki