

Sensible Chuckle SuperTuxKart Conceptual Architecture Report

Sam Strike - 10152402
Ben Mitchell - 10151495
Alex Mersereau - 10152885
Will Gervais - 10056247
David Cho - 10056519
Michael Spiering

Table of Contents

Abstract	2
Architecture Derivation Process	2
Conceptual Architecture	
· Original Conceptual Architecture Diagram	3
· Revised Conceptual Architecture	4
· Conceptual Architecture Overview	4
Architecture Subsystems	
· Games Specific Subsystems	5
· Rendering / Animation	7
· Input Controller	7
· Physics	7
· Audio	8
· Player Manager	8
· Network / Database	8
· Hardware Devices	8
Use Cases	9
Sequence Diagrams	10
Concurrencies	11
Division of Responsibilities and Development Team Issues	11
Lessons Learned	12
Conclusion	13

Abstract

SuperTuxKart is a free and open-source kart racing video game. Developed in 2004 (Initial release) by a team consisting of various developers and artists, it is still being developed today, with their latest release being in 2016. The game features several different race modes such as story mode/challenges, single player and local multiplayer. This report will analyze the game's conceptual architecture, which, though similar to other games, has a few unique components.

At the highest level, the architecture can be broken into eight different subsystems: Game Specific Subsystems, Rendering/Animation, Physics, Audio, Player Manager, Input Controller and Hardware Device. Each component has its own responsibility and interacts with other subsystems in order for the game to function. Throughout this report we will discuss the overall architecture of the game, examine the functions of each subsystem and inspect how the subsystems interact with each other.

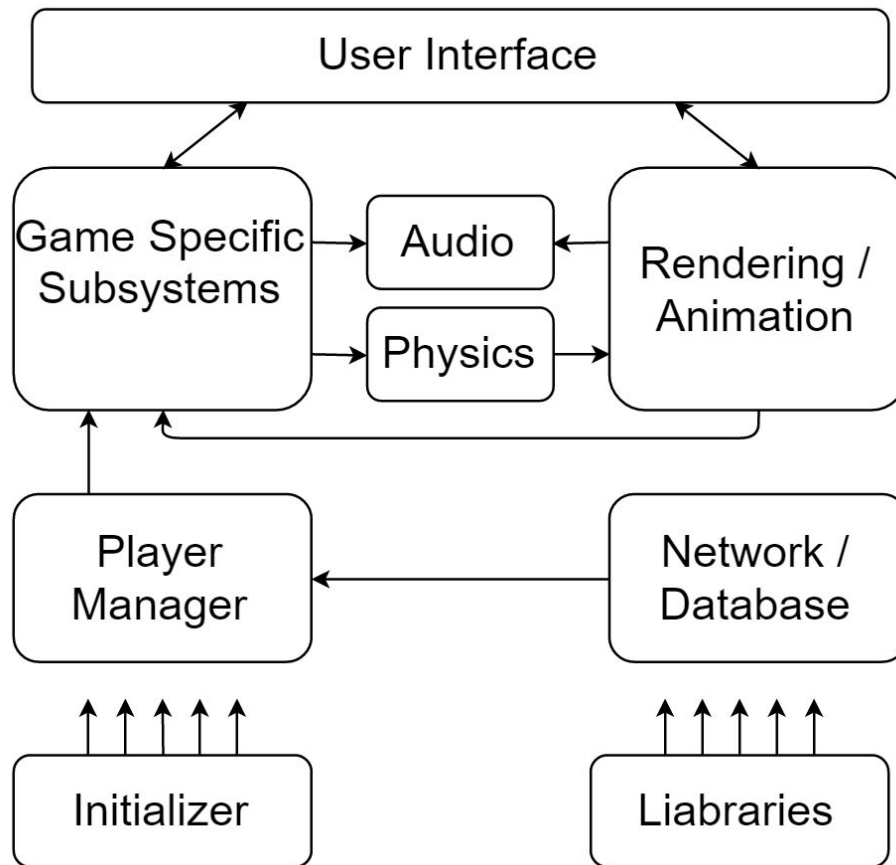
Architecture Derivation Process

To kick off the architecture derivation process, we first played *SuperTuxKart* to grasp what the game feels like and observed how each elements were interacting with each other. Once we had a general idea of how game looks and functions on an interface level, it was time to analyze the reference architecture handout that was provided in class. Upon reviewing the role of each system inside the reference architecture, we considered which systems in the reference architecture were crucial to *SuperTuxKart* and which were unnecessary. By deciding on the essential systems, we were ready to delve into *SuperTuxKart*'s source code to find the important components and generalize them to form the top-level subsystems.

During this time, we concluded that the overall architecture is in object-oriented style, as components behaved like an object inside their respective system, and the reference architecture in the game's documentation displayed the architecture as object-oriented. Then, we considered the dependencies between these systems and created many potential architecture designs while we attempted to group the components into various systems. The first few designs of the architecture were discarded as we decided that certain systems could be merged or removed, and they were not the highest level of systems (which is what a conceptual architecture diagram represents). We settled on a design for our presentation and, based on feedback, tweaked some of it to arrive at an architecture we were satisfied with.

Conceptual Architecture

Original Conceptual Architecture Diagram

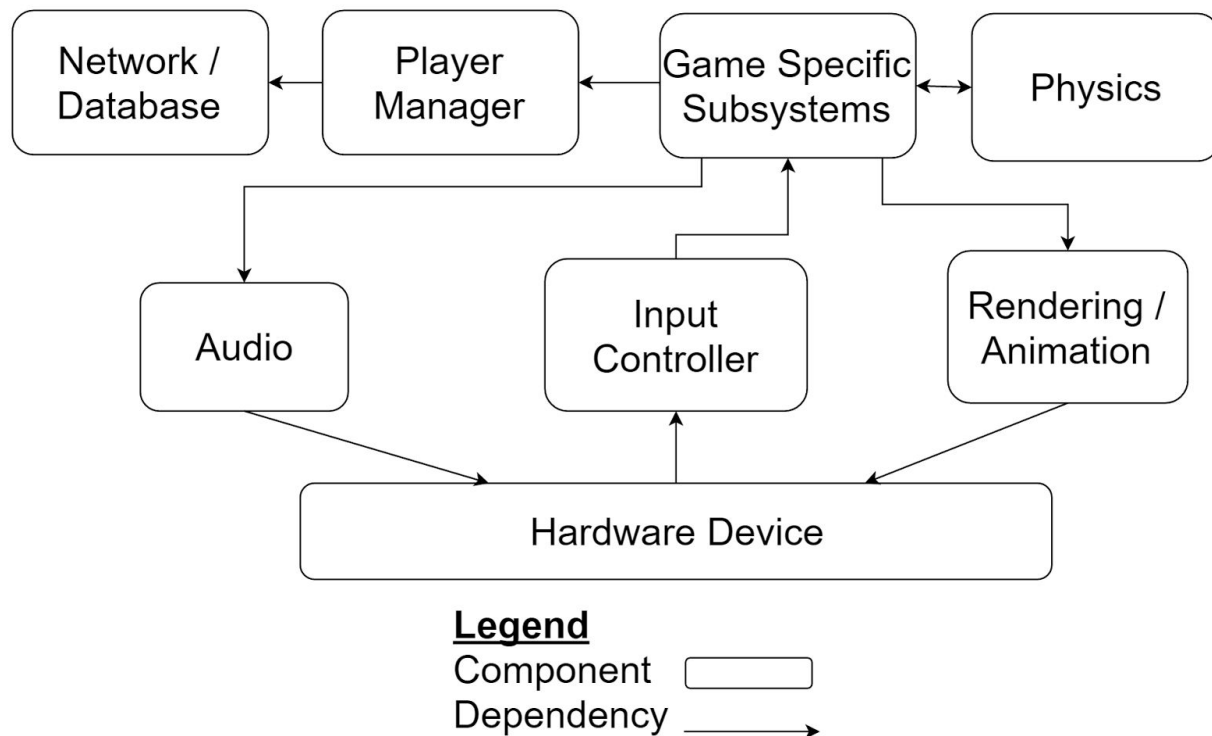


Legend

Component

Dependency

Revised Conceptual Architecture Diagram



Conceptual Architecture Overview

Our conceptual architecture follows an overall object-oriented style, although it also has qualities of a layered architecture. Due to the wide range of tasks that are involved in running the game, from physics calculations to networking to rendering, it makes sense to use this style in order to separate the operations and data related to each part of the system's operation.

In general, Hardware Device is receiving input from the computer's hardware, such as key presses, mouse movement and clicks. This information is then filtered by Input Controller to determine what is relevant based on the state of the game. This data is then stored in Game Specific Subsystems. Game Specific Subsystems interacts with Physics in order to determine, from the current game state and the current inputs, what the game state should be updated to. Physics then relays that information back to Game Specific Subsystems to update the game state.

Based on the game state, both Rendering/Animation and Audio determine what should be displayed to the screen and what sounds should be played respectively. Hardware Device then lets the physical hardware update the display and the speakers.

The secondary flow of information goes from Game Specific Subsystems to Player Manager to Network. This allows player information such as progress, unlocked features and high score data to the network to be displayed online. Since Super Tux Kart does not currently support online races between players, game state information does not flow from the network to Game Specific Subsystems.

This architecture is good for several reasons. Firstly, because of its object oriented nature, there is relatively high cohesion for all of the components. Secondly, the overall coupling is fairly low. With each component typically depending on one other component, and being dependent upon by one other component, (with the exception of Game Specific Subsystems, containing the core functionality of the system and interacting with most other components). Finally, because of the separation of the components, it allows for concurrent operations. For example, Audio and Rendering/Animation can work in parallel to speed up execution.

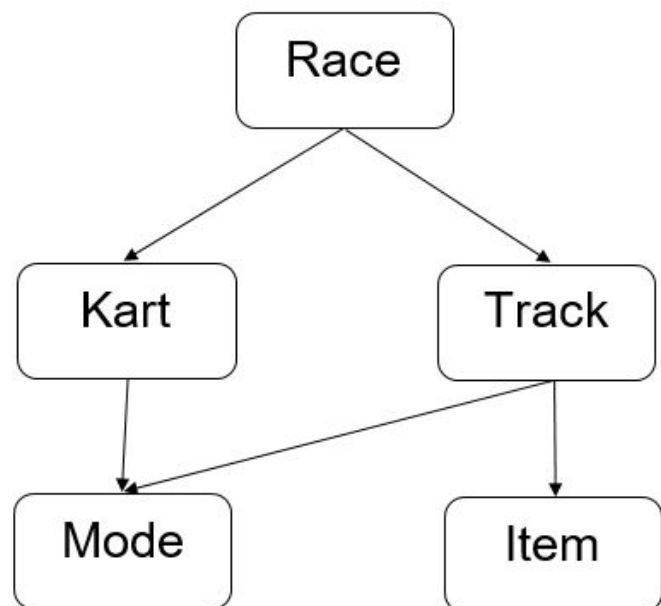
Overall our conceptual architecture is fairly simplistic which we feel is appropriate to give a clear conceptual understanding of how the system operates.

Architecture Subsystems



Games Specific Subsystems

One of the most intricate subsystems, the Game Specific Subsystems component essentially makes up the core of the gameplay entities that users interact with while playing SuperTuxKart. It can be broken down into various subcomponents that facilitate the formulation and execution of a typical race.

Race is the overarching subcomponent that coordinates communication between components, internally and externally. As its name implies, it handles the overall execution of a race; it takes information from the other



Legend

Component 
Dependency 

subcomponents and updates the game state based upon events / changes occurring within the game environment. It also tracks various statistics such as elapsed time and the current positions of the race's participants, and acts as the "controller" for any AI-driven participants. Within Game Specific Subsystems, Race depends upon two other subcomponents: Kart, which provides information as to the status of each kart in the race, and Track, which provides details concerning the current race's physical environment.

Every race participant is represented by their own Kart. In general, Kart keeps track of the details concerning the status of each individual kart – their position on the track, their current item inventory, their speed, etc. When an event occurs, the affected Kart object(s) pass information to Race for processing, and receive an updated list of details in return.

A race can take place on any given Track. This subcomponent represents the physical world in which the race takes place, from the roadway to the scenery to the boundaries. Race relies upon the information stored in a race's Track object to resolve any events concerning interaction with background entities (trees, rocks, etc), as well as any environmental hazards present (such as cliffs and lakes). In turn, Track keeps a list of the powerups present on the track, and where they are located; specific details concerning the nature of said powerups are obtained from the Item subcomponent.

A track holds a number of powerups arranged at fixed intervals that can be picked up and used by participants. The Item subcomponent embodies the list of these powerups and their various effects. Powerups provide a temporary effect that can either help the owner or hinder their opponents, and each individual powerup is represented by its own Item object. Track will ask for information concerning any individual powerup, and then pass this information up to Race, upon the latter's request.

In addition to those of a traditional race, races in SuperTuxKart can take on a variety of forms, each with its own set of rules and objectives. These forms, called modes, are denoted by the Mode component, and each mode is thus represented by a Mode object. Both Kart and Track depend on Mode for their various operations – Kart may need to store additional information on each kart, and Track may need to alter the current track based on different rules.

As a whole, Game Specific Subsystems depends upon three other components within the software system. Upon any interaction that occurs between two entities within the game, whether it's kart-to-kart, kart-to-scenery, or an item activation, etc, Game

Specific Subsystems sends details concerning the current game state and the state of the affected objects to the Physics component, which will process the result of these interactions and send the details of the updated game state back. Similarly, any event that occurs within the game that requires a change of animation or audio gets its details passed to the Rendering/Animation and Audio components respectively, which process and display the necessary assets.

Game Specific Subsystems influences a further two components. The Player Manager takes the results of races and updates the affected players' profiles, and the Input Controller receives information concerning the game state.

Rendering / Animation

The Rendering/Animation subsystem determines what will be displayed on the screen. It is an engine that generates an image from the game's 3D models and then passes those images to the user interface. The graphics depend on the animations because the animations are the illustrations of an object's movement within the game. The graphics are then plastered onto what is known as the skeletal animation. This subsystem renders game graphics such as lighting, textures, post effects, and skeletal movements.

The Rendering/Animation subsystem both depends on and influences other subsystems in the architecture. The Rendering/Animation subsystem receives data from the Game Specific Subsystem about the game's current state and then updates accordingly. For example, if the user has collided with another object within the game's world, the Game Specific Subsystem would tell the Rendering/Animation system to render what the new game world would look like, update the animations of the player and paste the game images onto the new renderings. Afterwards, the renderings would be sent to the users hardware and displayed to the user.

Input Controller

The Input Controller gathers all the information about what the user inputs on their hardware device. If a user presses a key or performs a mouse click, the Input Controller will be the first software system to know about it. After the Input Controller has received some data, it sends it off to Game Specific Subsystems, where the game state can be updated. The Input Controller must be able to handle multiple inputs at once because the game allows for local multiplayer.

Physics

The role of the physics component is to do all of the complex calculations involved in updating the game state. It receives the game state, and changes it based upon the information it has on how all the various game entities should interact. For example, what happens when a bowling ball and a kart collide. It calculates what should happen to the entities involved in an event and relays this information to the Game Specific Subsystems in the form of a new game state.

Audio

The Audio subsystem manages all of the sound-related functions, including background music and sound effects. It has a dependency on the Hardware Device subsystem because it sends sounds to the speakers, and it is referenced by the Game Specific Subsystems for everything related to sound.

Player Manager

When someone first installs *SuperTuxKart* they have the option of creating an account to track their progress, and this is stored in the Player Manager, the only component of the game's current architecture that utilizes network capabilities. A player profile is created locally, and the Player Manager tracks a user's progress through the game, storing unlocked cars, maps, course times/records and achievements. These records are uploaded remotely to a server for comparison with other users. The Player Manager provides user-specific information to the Game Specific Subsystem, such as a player's name. The Player Manager would be important in any further online implementations the game might consider, such as peer-to-peer play.

Network / Database

Though the developers of *SuperTuxKart* have established their intentions to eventually incorporate an online multiplayer mode to the game, as of the present time, they have not yet finished such an endeavour. Therefore, the current interactions between the game and an online resource is solely relegated to those of a simple remote database storage for player data. This data is comprised of player details and their respective records and achievements unlocked through playing the game's various modes.

The only component that is depended upon for this interaction is that which (locally) holds the player details in question – the Player Manager.

Hardware Device

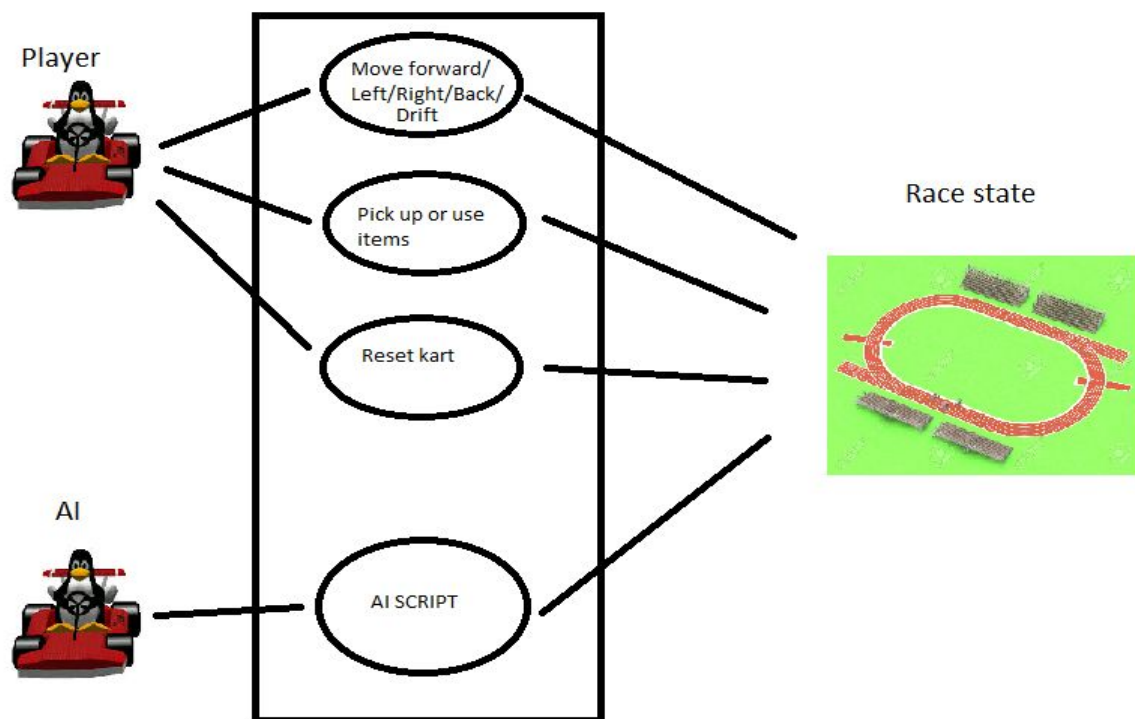
The Hardware Device is the physical device the user is using to play the game. It is not actually part of the software system, but nevertheless it plays a major role in the

overall architecture. When an event is triggered, such as a keypress or mouse click, the Input Controller subsystem receives information about what the user entered from the Hardware Device.

The Hardware Device also receives data from the Audio and Rendering/Animation subsystems. The former displays the game's visuals on the hardware, and the latter plays the game's sound effects on the hardware. The hardware also ensures the audio and visuals are playing simultaneously.

Use Case

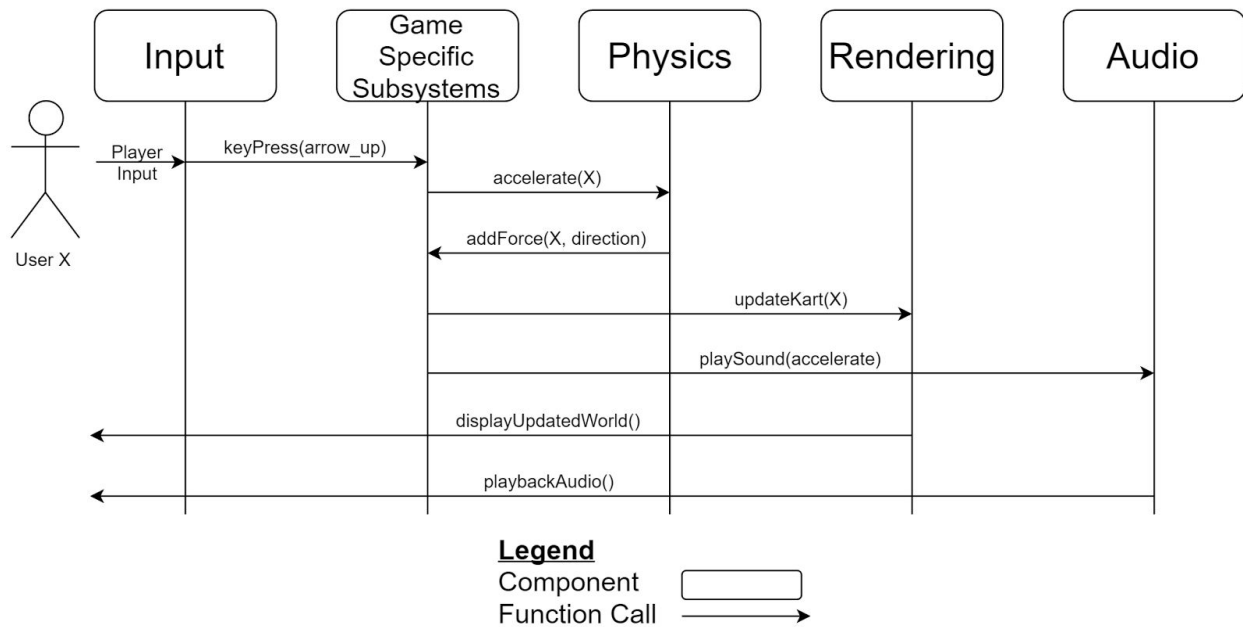
Use case of a Player in a Race



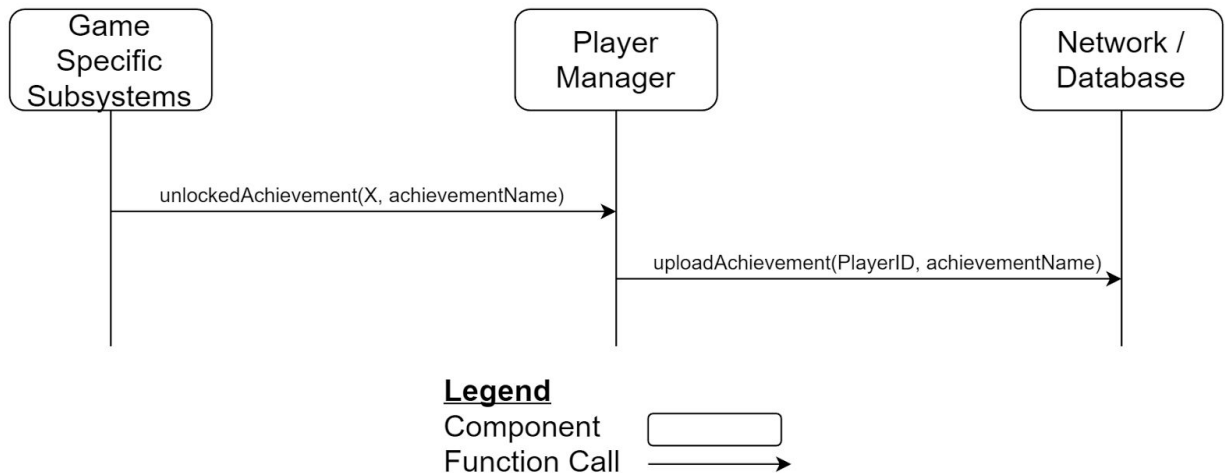
The figure above shows a use case during a race. In this example, a player and an AI are in a race and can both have an impact on the race state. The player chooses what they want to do, such as direct their car, use items, or reset themselves. This is passed to Race, within Game Specific Subsystems, and Race updates the track based on the decision at the next frame. The AI runs through its own scripts based on the game state, which is similarly updated. The game state is constantly updated by the decisions of the karts; the more players or karts the game has in it, the more concurrent and complex updating the game state becomes.

Sequence Diagrams

Sequence Diagram of User X accelerating forward



Sequence Diagram of User X unlocking a new Achievement and being uploaded to the game's network



Concurrencies

In SuperTuxKart multiple concurrencies exist throughout the game and must be properly managed at runtime for a proper experience.

First;y, the game is multiplayer, meaning that multiple inputs must be managed simultaneously. As an open source game, it offers many different platform and input options for the game, which, on top of the complication of additional client support, requires multiple different and concurrent inputs.

When racing, all karts must be managed and updated concurrently, whether human or AI, and they must all have the capability to pause and resume a race. As the karts move through the track, there are frequent position changes, item activations, and interactions between karts. The game must manage these interactions concurrently in the game state, which is an important part of a high quality game.

Throughout all menus and gameplay, there is background animation and audio constantly playing. Audio must be managed based not just on what menu or level you are on, but on what is happening. For example, when someone throws a bowling ball, the game must play the bowling ball sound bite, on top of the already playing music. When moving through a track, your kart's position and what your camera sees is constantly changing, based on the background as well as the game state.

Division of Responsibilities and Development Team Issues

The core *SuperTuxKart* team is comprised of three individuals:

- Joerg Henrichs (“Hiker”), the Project Leader
- Marianne Gagnon (“Auria”), the Main Developer
- Jean-Manuel Clemençon (“Samuncle”), the Lead Artist

Everyone has their own area of expertise, and therefore assumedly has their own set of responsibilities and duties to perform throughout the process of creating each successive build of the game.

In addition to these three, the open source nature of the project made it possible for a plethora of other people to contribute to the development of the game over time, whether as programmers, QA testers, web developers, graphic artists, or sound engineers.

Issues surrounding the development of *SuperTuxKart* would most likely stem from the nature of the project itself.

With multiple people working on different components, making sure that they all still function as a single system can be an enormous challenge. With respect to an object-oriented architecture, though every individual component exists independently from every other component, there are many avenues of communication that exist between them to facilitate the transfer of data. Should one component change its characteristics, it is essential that said changes do not react unexpectedly with other components when information is transferred – conversely, it is important to ensure that any changes in objects’ naming *are* in fact propagated to connected objects that invoke them.

Furthermore, the multidisciplinary nature of the team, coupled with the fact that *SuperTuxKart*’s open source format allows anybody and everybody to create potential content for the game, means that not only is the potential for scheduling conflicts high (for example, graphical assets for a new item may be ready, but the logic behind the item’s usage may still be in development), the need for a unified coding style and graphical format for development is increased tenfold. If someone intends to alter a specific component down the line, they need to be able to quickly and easily parse through the source code and documentation without wasting time deciphering the previous developer’s intentions. In addition, if someone submits a new kart model, it

needs to be in a format that can be read by the rendering engine. Thankfully, there is a dedicated section for style and consistently on the *SuperTuxKart* website for just such an occasion.

Lessons Learned

As a team, we learned a few things in the process of completing this assignment.

Probably the most important of those was “the simpler the architecture, the better”. It's easy to get bogged down by the sheer number of subsystems needed in the implementation of a complex piece of software, and the intricate dependencies between each. Our earliest designs had many boxes and arrows going every which way, but throughout each iteration we slowly took a few steps back and consolidated redundant components and connections, until we were left with something infinitely more manageable, and therefore infinitely more comprehensible.

We also fell prey to the common pitfall that frequents every group project. When dealing with group sizes this large, it's important to exercise high cohesion to ensure that everyone has their own task to complete. We were a bit complacent with respect to the presentation and report, and that led to the bulk of the work being finished slightly later than intended, rather than ensuring that the majority was finished concurrently earlier on.

Conclusion

Through trial and error, we arrived at an overall object-oriented architecture style, albeit utilising some layered architecture concepts as well in our final design. Our eight subsystems work in tandem to facilitate the flow of information necessary for SuperTuxKart to operate effectively, the most important ones comprising of Game Specific Subsystems, Rendering/Animation, and Input Controller. They employ a high level of cohesion as independent components, and coupling is as low as possible between them - dependencies are on a “needs-to-function” basis only.

Each subsystem serves a purpose, and in this report we walked through sequences diagrams showing some typical functionality. We also discussed some of the key issues that a developer would need to keep in mind throughout the process of designing a conceptual architecture, including concurrency and team-based complications, and cited some of the lessons we will use in future assignments.