Conceptual Architecture for SuperTuxKart

SuperFunKart

Alex Globus (a.globus@queensu.ca)
Annika Nicol (14ajn@queensu.ca)
Lucy Rowland (13lr31@queenus.ca)
Mahad Amir (mahad.amir@queensu.ca)
Matt Williams (Matt.Williams@Queensu.ca)
Alex Weatherhead (12kaw3@queensu.ca)

Abstract

This report looks at the architecture of the game SuperTuxKart. Through research of documentation, it derives a hybrid Object Oriented/Layered architecture style as a conceptual architecture. The architecture is broken down into the Game-Specific, Assets, Utilities and External Libraries layers. These layers have objects inside them. The layers are not rigid and elements of an Object Oriented style can be seen in the connections between modules. The report examines the interactions between these different modules and provides a look at the data flow between them. It also takes a look at how team composition can affect the architecture of a project.

Introduction

SuperTuxKart is a free and open-source racing game, programmed in C and C++, and made available for Linux, Windows, Mac OS, and select distributions of Unix [1]. The game itself comes from an older project, titled TuxKart, which was originally developed for Linux by Steve Baker between 2000 and 2004[2]. TuxKart was picked up after version 0.4.0, which was its last update, and further developed by the “Game of the Month” project of the Linux Game Tome from June to December in 2004; the result was the first, albeit highly unstable, release SuperTuxKart [3]. The game was then later fixed up by current project-lead Joerg Henrichs in 2006, and, under his direction, has been slowly moulded into what it is today[4].

The purpose of this report is to identify, dissect, and evaluate the conceptual software architecture of SuperTuxKart, as well as to discuss the architectural derivation process holistically, with particular attention to limitations and the importance of team dynamics. For the sake of this report, conceptual software architecture (referred to hereafter as either conceptual architecture, or just architecture) is defined as the blueprint for a large piece of software, and is thus comprised of the various components of that software system and the connections between said components. The derivation process on the hand, refers to the means by which one constructs a conceptual architecture.

This report begins by discussing the process through which our team derived the conceptual architecture of SuperTuxKart. After that, this report presents, through a series of detailed architectural diagrams, the layered/object-oriented style architecture that was ultimately decided upon. This report then spends some time explaining and decomposing the major components of the conceptual architecture, and assessing the high-level interactions and dependencies of each component, as well as their concurrency, evolvability, and testability. This report then examines two important use cases related to the system: the process by which a user starts a new single-player race from the main menu, and the process by which a user activates their special power-up item during a race.

Next, this report discusses some of the architectural styles that were considered in place of a layered/object-oriented style; these styles include: layered, object-oriented, client-server, and repository style architectures. This report then delves into a discussion of the developers’ team dynamics and how they may have impacted the

architecture of SuperTuxKart, as well as a look at the lessons learned by our own team throughout the course of devising the conceptual architecture for SuperTuxKart. This report then wraps up with a summary of, as well as some reflections on, everything discussed above.

Derivation of the Architecture

The derivation of our conceptual architecture began with a rational process of elimination. As there are relatively few (pure) architectural styles commonly used in software, and more specifically games, our team simply made a list of all possible options and began to cross out those which would not make sense in a game like SuperTuxKart. We were left with the following options in architectural style: layered, object-oriented, repository, client-server. Since the game was developed for single-player only originally, our team was able to confidently cross client-server and repository (since there is no need to centralize data management) off our list. Thus we were left only with a choice between layered and object-oriented.

Next, our team began to research the game more thoroughly to find any existing material on its architectural style. One of the most significant sources that we came across was the developer documentation for SuperTuxKart [5]. This documentation helped us to realize that the system's components could not fit neatly within either a layered or an object-oriented style architecture, and would instead be better off within a hybrid of the two styles.

Knowing that we wanted to create a layered/object-oriented architecture, we decided to use the reference architecture provided to us in class to help us construct our conceptual architecture for SuperTuxKart. With the knowledge we gathered from the developer documentation, we moulded this reference architecture to fit the specifics of SuperTuxKart, while keeping the general structure intact. After a number of iterations, we arrived at our current architecture.

Throughout the derivation process, our team faced a number of key constraints. First, since SuperTuxKart has switched project-leads and, as a result, directions a number of times, we can guess that development has not coalesced around a single shared architectural vision; however, the effects that such changes have had on the architecture as the game has evolved are largely unknown to us, meaning that our architecture does not necessarily reflect the likely ad hoc nature of the contemporary system.

Second, and relatedly, our knowledge of the system was limited to what is explained within the developer documentation, which seems oftentimes outdated, unfinished, or both. Thus, our architecture may not reflect the architecture of the system as it is today.

Conceptual Architecture

After careful consideration of the documentation, our team derived the following architecture. The architecture is in a layered style with the most general layer at the bottom followed by layers in increasing specificity as we traverse up the layers.

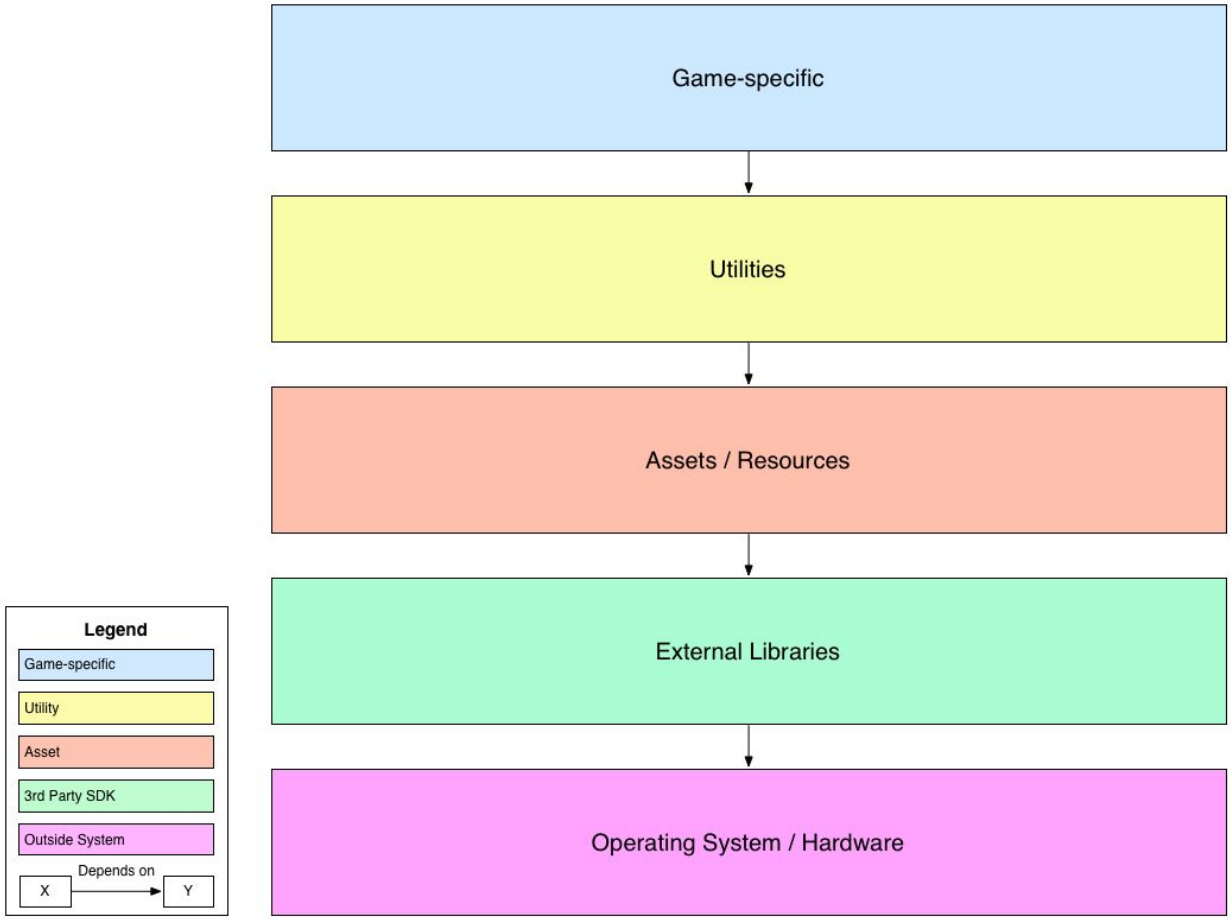

Figure 1: Simplified Conceptual Architecture

The hardware and operating system forms the base of our architecture; it is the core platform that provides all running programs (not just SuperTuxKart) with the means to communicate with the hardware through operating system primitives such as

mutexes and locks for concurrency, system calls, and memory management interfaces. The operating system provides mechanisms for higher layers to communicate with the hardware in a standardized fashion.

Above the hardware layer lies the 3rd party SDKs SuperTuxKart requires to function. Libraries that are independent of the game but still provide necessary functionality comprise this layer. There is still a fair amount of generality in this layer. Libraries such as OpenGL, DirectX, Jpeglib, etc., do not need to know anything about SuperTuxKart to perform their job. They depend only on the operating system, and in some cases each other, and expose general APIs that SuperTuxKart uses. This is one key benefit to the external libraries layer; it wraps operating system specific functionality into libraries that are written to be cross-platform. Therefore, applications relying on such libraries can expect consistent behavior across a multitude of operating systems.

Since SuperTuxKart is a game, and games have a plethora of graphical resources that need to managed, it makes sense to have a dedicated layer for handling that. We represent such a layer as the assets/resources layer sitting directly above our external libraries. Logic for how to handle the game's resources is determined in this layer. When our graphical assets need to be drawn on screen, the resources layer sends this information down the architecture to be handled by the external libraries.

Utilities and game-specific logic are the final two layers operating at the highest level. Game-specific modules are those that are directly pertinent to the racing style game of SuperTuxKart, hence the "specific" in the name. We can generalize the lower-level components as being common across games of different types. For instance, first person shooter games still require a way to manage in-game assets. However, for SuperTuxKart, game-specific modules include the kart, the different kinds of racing modes, the track layout and racing rules, and so on. The utilities layer acts as an intermediary between game-specific and resources, taking care of any processing that needs to happen between the two.

We can show our chart in more detail by classifying each submodule in their respective layer. Within each layer, submodules are organized in a hierarchy with each submodule acting as a client to the submodules directly beneath and providing an interface for submodules above it. These submodules can be thought of as independent acting objects whose dependencies dictate the relationship each object has with one another. The advantages this hybrid approach offers is two-fold. First, an object-oriented approach provides us with data encapsulation. This means that an object contains its own private data and allows an object to change its private

implementation without affecting its public behavior. This is especially useful in a layered architecture as we can re-organize layers at a later time without worrying about tight coupling.

Observe that our derived submodule architecture is not strictly layered. That is, some modules pass through layers to modules further than its immediate neighbors. For example, although the world module connects with components in the utilities layer, it also has a dependency on the graphical component residing in the assets/resources layer. However, a general order is still preserved. For example, the module responsible for rendering the GUI/HUD does not talk directly to the graphics driver to render itself. Logic for drawing is handled implicitly through the module that handles all textures, which handles rendering itself implicitly through the graphical module and so on. Submodules are not connected to layers beyond their immediate neighbor unnecessarily and the delegation of logic flows from top to bottom. A more detailed diagram is shown as follows.

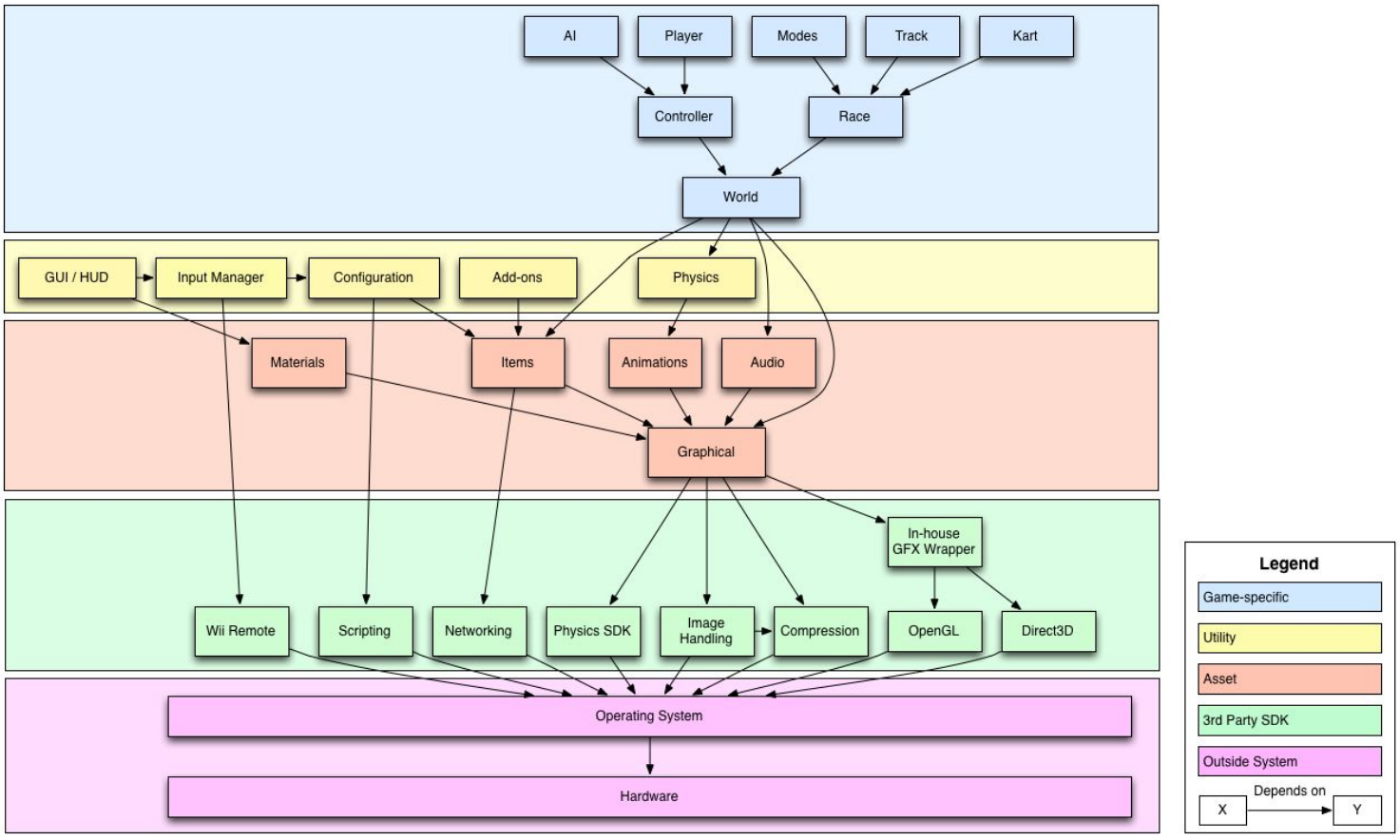

Figure 2: Conceptual Architecture

Major Sub-Systems

The following sections provides a deeper look at each of the layers in the architecture diagram shown above.

Game-Specific

The game-specific layer is highest top layer in SuperTuxKart's layered architecture system. This layer contains the game play and racing elements of of the video game. This layer also contains information on the controller, players, and the AI. This layer also handles user input, through the player and controller module. In total, there are eight different modules contained within the game-specific layer, with the world module being the most important. The layer contains:

- AI: Controls the non-human players
- Player: Takes user input and dictates the movements of the user's character
- Controller: Controls, moves, and sends actions to the in game characters
- Track: Checks and maintains the properties of the track and race
- Kart: Dictates the properties of the karts
- Race: Controls in-game race data
- World: Runs the race

The world module is the class that runs the race and is used to implement the karts, tracks, and game-modes through accessing the race module. Also, it is the catalyst in which the game-specific layer interacts with other layers in the system. The world module is dependant on the utilities layer through the physic module and also dependant on the asset layer through the items, audio, and graphical module.

The game-specific module runs at a high level of concurrency. The controller module and the race module will have to run in tandem with the world module. The race module is handling the data on the race like high scores, network information, numbers of AI racing, and then controller module is dictating how the racers are moving around the track. These are all running while the world module is running the race.

Evolvability in this layer is fairly straightforward. There are very low instances of coupling amongst the modules, aside from the world module. Additionally, no modules in the system depend on AI, player, modes, track, and kart.

Utilities

This layer acts as a collection of processes that are required to present the game world to the player. It lies above the Assets layer, and beneath the Game-Specific layer. This layer is bypassed by a few of the game-specific objects so that they can interface directly with the Assets layer to provide the relevant information. This layer provides user interaction through a series of GUI objects that represent screens as well as a module to handle input events for these screens.
The layer contains:

- Configuration: Deals with propagating the settings stored by the user in config files
- Add-ons: Deals with expansions made to the game that can be downloaded and added in
- GUI/HUD: Creates menu screens and the HUD overlay in the game world
- Input Manager: Deals with mouse and keyboard events in the menu
- Physics: Provides a base layer of physics to deal with animations

Concurrency is not expected from the layer as a lot of the modules perform their function in the beginning of the game and then stop. Physics is expected to be running concurrently with the world to ensure basic physics capabilities during the game.

Assets

The assets layer unifies the interface using the inner modules and the external library on the lower level. The graphical module acts as the resource manager to constructs how each module interacts within the layer.

The utilities layer consist of user inputs so asset layer needs to retrieve those inputs and put them into action. Therefore the assets should be layered underneath utilities. The graphical module is then dependant on the STK's external library, which is the lower level. The graphical module relies on engines like physics and image handling so it can correctly compute how items, audio, animations and materials interacts with one another simultaneously. There is a more active appearance of concurrency in assets where the graphical module is concurrently interacting with the other assets. Although there are certain cases where the graphics module needs to engage the physics library quickly in order to continue through to the next updated calculation. Therefore there has to be concurrency control within the graphical module when interacting with the external libraries.

The asset module in an important part of the system because in the absence of this layer the game would not have an interface. Karts would not move, the user input actions would not update to the screen and the start a race would not even begin. Consequently, the importance of the modules to be correctly working is vital to running the game.

Evolvability is not as prevalent in this layer as it is in game logic and utilities because the modules are coupled to the graphics manager therefore it creates more complications when adding and removing elements.

The layer contains:

- Graphical: This module contains the core graphic engine, which is a thin layer on top irrlicht providing some additional features the system needs for STKs.
- Audio: Handles the sound effects and music.
- Materials: Materials are used to create texture and 3D graphics for objects in the game.
- Items: Contains modules that manages the feature that the karts are able to use in a game.
- Animation**:** Manages the position, rotation and/or scale of the features of the game.

External Libraries

The External Library is part of the Layered architecture present throughout the structure. Lying just below Assets, the External Library is also joined to the Operating System, lying just above it. This architecture style implements a low coupling model that favours the ability to make changes with low overhead, which is especially useful towards the External Library which is likely to add and remove sections as the game grows in size and focus. This is evident in the Networking and Compression sub-sections of the External Library, as neither has been developed in full, but are aiming towards completion. The library has a unique quality in that the components range from almost always functional (e.g. 3D Physics), to almost not at all (e.g. Networking). The concurrency is quite high between Physics, Engine, and Image Handling. The evolvability of each component is quite high, as the layered architecture lends itself to low coupling.
The layer contains:

- 3D Physics [8]
- Networking [9]
- Game Engine [10]

- Image Handling [11] [12]
- Compression [13]

The 3D Physics and Game Engine are very similar; in testability through using known-interaction testing, and criticality as both are system critical. If either of these subsystems are not working properly, the system cannot run. On the other hand, networking, image handling, and compression are all not critical, and have similar testability. network and image handling can both benefit from emulation of possible computer systems, while compression is about compressing and decompressing files to check validity.

Sequence Diagram Examples

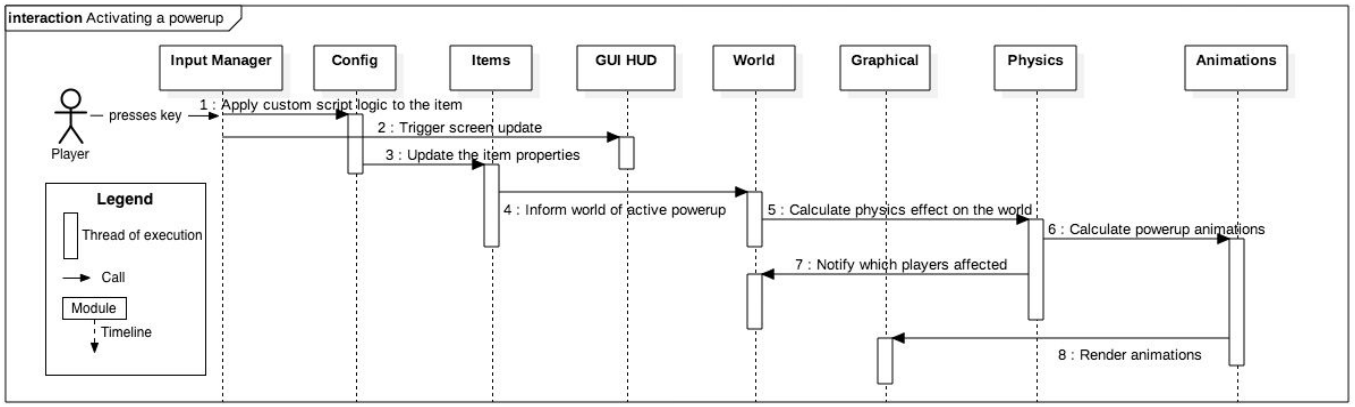

Figure 3: Sequence Diagram for activation of a powerup

The first sequence diagram details the process of a player activating an in-game power-up. Power-ups are in-game items that can be used to gain an advantage in the world. For example, some power-ups speed up your character and others hinder your opponent's progress. The following is the series of steps that occurs in the architecture when a player activates their power-up with a keypress. For the sake of simplicity, lower-level logic calling external library functions is omitted; the diagram focuses on the interaction between the top three layers (game-specific, utilities, and resources).

We also detail the process of starting a race from the main menu. The player interacts with the GUI component which forwards its events to the input manager submodule. From there, the input manager determines the type of action and updates the configuration accordingly. It does this for the kart, mode, and track selection screens. Each time the input manager communicates with the configuration, it fires off a callback to let the GUI draw the next screen. If the event passed to the manager signals to start the game, the manager triggers the configuration module to do any finalization before the items are loaded into memory and world is initialized. Like the previous example, the diagram omits showing the irrelevant flow of logic to external libraries to keep it simple

Alternative Architectures

At the beginning of the process of analysing SuperTuxKart's architecture, we pondered across the conclusion that the architecture was just a layered style. So first we categorized the high level layers into game specifics, utilities, assets, external libraries and the outer system. When we started to develop the diagram and analysing further into the doxygen we noticed the components did not follow the attributes of being a layered architecture. Instead of modules passing through each layers they would bypass a lower layer to the next. In addition, the data and procedures encompassed within the system's components would not be as dynamic, because each layer would represent only an increased level of abstraction, and not an autonomous component.

This led to an alternative solution that the architecture could be object oriented. We soon realized that it could not be strictly an OO architecture. The system lacked coupling, because in order for one object to interact with another in such systems, the first object must directly call the two one.

Finally, we agree since it encompassed both attributes of the architecture styles that the system is a hybrid of layered and object oriented. Therefore we could represent the diagram accurately following both restrictions of the styles.

Team Dynamics and Their Effects on Architecture

As was discussed at the outset of this report, development on SuperTuxKart has been done in spurts over the last decade and a half, and has been led by a number of different groups. During this time, the game's architecture likely underwent severe turbulence as it was adjusted to meet the needs of each new team. Presently, SuperTuxKart's development is tied to its open-source model, as it depends on the contributions and skillsets of its various community members. SuperTuxKart has

relatively few barriers to entry for developers, as it does not demand the same amount of polish as a commercial-release game. As a result, the number of contributors and contributions is quite high for the game. Due to the efforts of community developers, the game has been able to evolve according to the system's needs and not according to the system's constraints; thus, SuperTuxKart has largely avoided one of the major pitfall in architectural designs; that of designing a system around one's team, and not vice versa.[6]

An important matter to note is that the composition of this development team implies a relatively poor communication structure, as community developers are bound to have fluctuating commitment to the project, and are likely to find themselves out of the loop during times of inactivity. The effects of this inactivity on the team's communication structures may be mitigated by initiatives such as the mailing lists, which try to keep community members up to date. Conway's Law states that teams are constrained to design systems that match their communication structures;[7] this is important to consider in relation to this game's development team, and open-source development teams in general.

Lessons Learned

After many nights going over design documents and architecture specifications, it became easy to realise that open-source projects are not necessarily more transparent than their closed-source competitors. Even with the documentation present, much of it was either partially incorrect or completely false at the time of review. In addition, hobbyist projects (in comparison to commercial projects) seem to pay less attention to the formal conventions of software development, and as a result, skip right to the design and implementation of the system, forgetting about architecture. This leaves a very unorganised, disheveled mess when trying to revert it back to a known variant of architecture. For instance, after our examination, the closest we could see was a mix of a loosely layered architecture with object-oriented subcomponents. Finally, reference architecture can be an extremely useful tool, but nothing is ever "one size fits all" when it comes to conceptual software architecture, as systems all have their own unique requirements. Above all, this project has been a firm reminder that beginning with and sticking to an architecture is the foundation of clear and understandable code.

Conclusion

Our team derived a layered/object-oriented style architecture for the game SuperTuxKart. This architecture was presented through a series of system decomposition diagrams, which laid out the major layers of our conceptual architecture (the Game Specific, Utility, Asset, and 3rd Party SDK layers), and all of the significant components in those layers. The presented architecture was then later reinforced through a number of sequence diagrams, each of which exhibited how a particular use case would be handled by the system. The major subsystems of the architecture were also individually analyzed with particular attention to issues such as concurrency, evolvability, and testability. Beyond this, our report discussed a number of other possible architectural styles for the game, the dynamics of the development team and how they may impact the game's architecture, and the lessons learned by our team throughout the process of creating this report. .

References

[1] https://en.wikipedia.org/wiki/SuperTuxKart
[2] Ibid.
[3] https://supertuxkart.net/FAQ
[4] Ibid.
[5] http://supertuxkart.sourceforge.net/doxygen/?title=doxygen
[6] http://cs.queensu.ca/~ahmed/home/teaching/CISC326/F16/files/WhatIsSoftwareArchitecture.pdf
[7] https://en.wikipedia.org/wiki/Conway%27s_law
[8] http://bulletphysics.org/wordpress/
[9] http://enet.bespin.org/
[10] http://irrlicht.sourceforge.net/
[11] http://www.libpng.org/pub/png/libpng.html
[12] http://www.ijg.org/
[13] http://www.zlib.net/

Glossary

1. **STK** – Short for Super Tux Kart
2. **Layered Architecture** – Hierarchical tiers of differing responsibilities
3. **Object Oriented** – Methodology of modeling a system as a set of independent objects
4. **Evolvability** – A system's capacity and ability to adaptively evolve
5. **Testability** – A measure of a piece of software's extrinsic ability to find faults through testing
6. **Reference Architecture** – A set of documents that model the ideal architecture for the application in question
7. **Sequence Diagram** – A diagram that demonstrates how objects communicate with one another and in what order