Pangolin: Concrete Architecture of SuperTuxKart

Caleb Aikens
Russell Dawes
Mohammed Gasmallah
Leonard Ha
Vincent Hung
Joseph Landy

Abstract

For this report we will be looking at the concrete architecture of SuperTuxKart. We will start by going over the derivation process for the concrete architecture, as well as reviewing the various components and dependencies of the subsystems. We determined that SuperTuxKart is an object-oriented architecture style made of various subsystems such as the Game-Specific, Audio, Network, Gameplay Foundations, Graphics, Physics, HID (Human Interface Devices), Libraries and Utilities.

Introduction

SuperTuxKart is known to be one of the best open source multi-platform games. Through it's game engine architecture, it is able to deliver on a fun and interesting game play independent of the operating system.

SuperTuxKart's gameplay is similar to Mario Kart but has changed greatly over time. It's mascots are from open source projects such as Mozilla Thunderbird, Beastie (BSD), Gnu (GNU), Hexley the Platypus (Darwin), Puffy (openBSD) and of course Tux (Linux). Currently the game has single player and local multiplayer, and they are planning a release of networked multiplayer.

SuperTuxKart is a rerelease of the old TuxKart and uses a completely new rendering engine known as Irrlicht which helps to improve the graphics from the previous release of the game. With the new graphics, SuperTuxKart has also decided to include new tracks, new karts, and new online accounts to use for the achievement system.

Using the Understand Static Code Analysis Tool by Scitools, as well as the public source code for SuperTuxKart, we were able to establish a concrete architecture. The network subsystem was added as there was a heavy part of the code that included networking capabilities. The libraries has been separated into libraries and utilities so as to demonstrate some of the utility functions that are required by most of the subsystems. Examining these dependencies, we determined that the architecture for SuperTuxKart is an Object-Oriented style which is different from the initial Layered Object-Oriented style we had hypothesized in our conceptual architecture.

Concrete Architecture Derivation

Derivation Process:

We were initially provided with the SuperTuxKart source code and the Understand Tool. We opened up the source code in Understand and looked at some of the different sources in SuperTuxKart and reviewed our conceptual architecture with these new sources in mind. Using this new conceptual architecture, we created our new architecture in Understand and moved sources into the components as we deemed fit. Once this was complete, the subsystems and files that Understand provided were reviewed once more, and we noticed that some file dependencies did not match up with our conceptual architecture, and some source files were still not used. This led to the creation of the Networking subsystem as well as the removal of many of the original lower levels of the conceptual architecture that we had (Hardware Abstraction layers, Operating System Independance Layer, etc). This allowed us to change some more of our conceptual architecture, but even still some dependencies were seen that were not expected.

After this was done, we went back into our concrete architecture in Understand and we took some time to investigate the files and the dependencies in order to establish the reasons for the unexpected dependencies. We continued to make minor changes in which source belonged to which subsystem as best as we could until we were satisfied with our concrete architecture.

Concrete Architecture

Conceptual and Concrete Architecture:

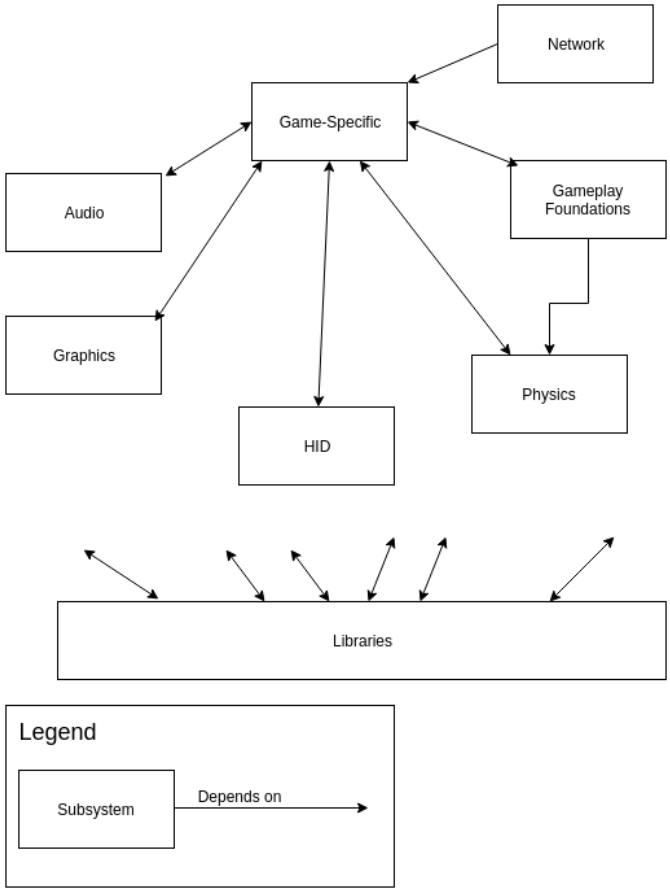


Figure 1. Conceptual Architecture Revised

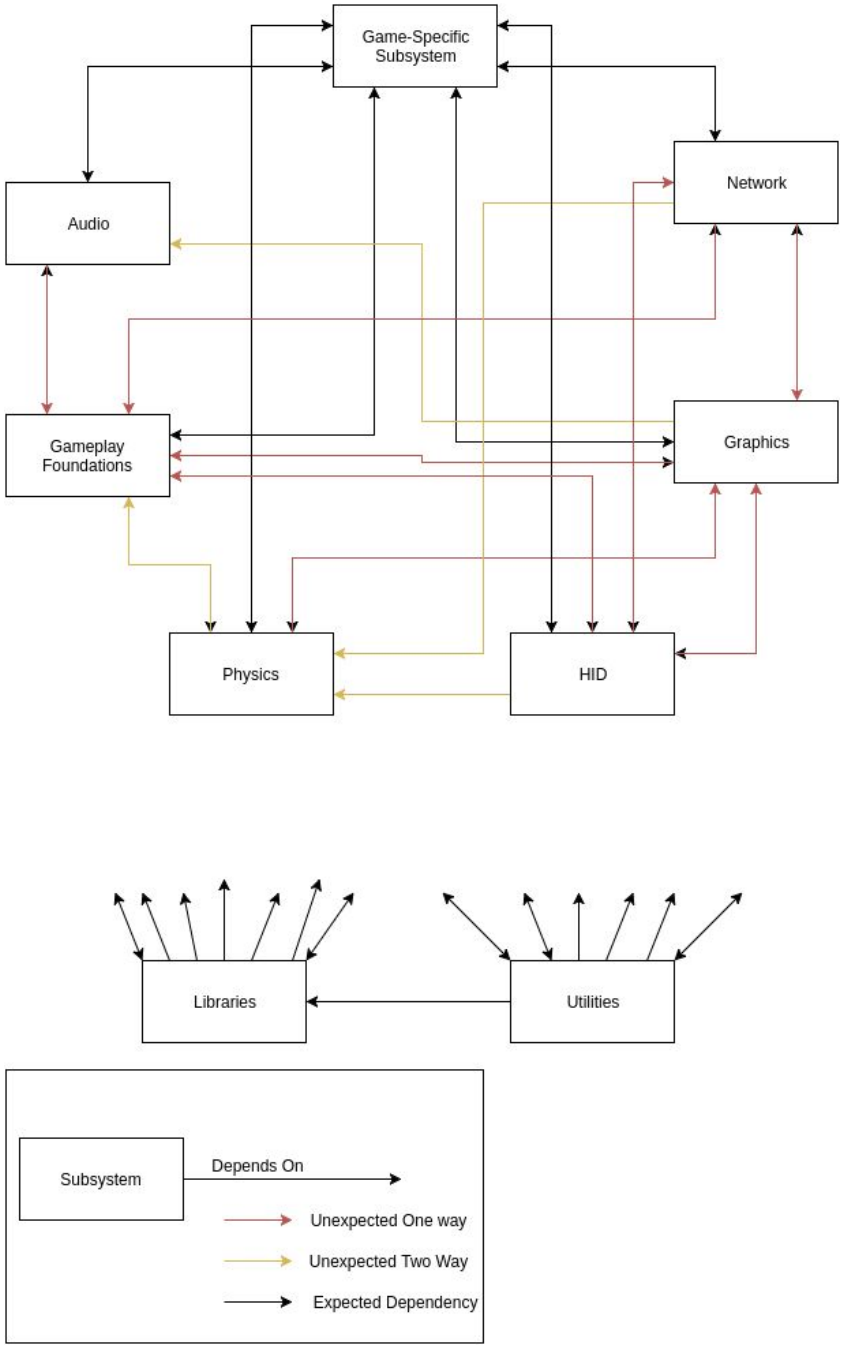


Figure 2. Concrete Architecture

A note on dependencies:

Unexpected Dependencies:

Game-Specific Subsystems -> Network: Game-Specific Subsystems updates based on information from Network

Graphics-> Audio: Graphics calls Audio functions to play sounds for graphical events.
Graphics-> Game-Specific Subsystems: Graphics obtains files relevant to karts, tracks, race and mode.
Graphics-> Physics: Partially explained by the functionality in Graphics for drawing information from physics for debugging purposes. Elements such as the functionality for drawing skid marks connect directly to the physics system.
Graphics-> Gameplay Foundations: Graphics requires access to some things in GSS in order to draw some types of content.
Graphics-> Network: The guiengine gets NetworkPlayerProfile for the player_kart_widget.hpp.

Gameplay Foundations-> Network: A few elements depend on Network. For example, the player_profile class depends on the online_player_profile class.
Gameplay Foundations-> Audio: Shares config and management.
Gameplay Foundations-> Graphics: Uses animations, irr_driver, show message, and engine.hpp.

Audio-> -Game-Specific Subsystems: Gathers information about world and track.
Audio-> Gameplay Foundations: Uses config files.

Network-> -Gameplay Foundations: Gets CurrentPlayer from Gameplay Foundations.
Network-> Graphics: Uses graphics for rewinds.
Network-> HID: Gets deviceManager, input_manager, and input_device.
Network-> Physics: Uses physics for rewinds.

HID-> -Gameplay Foundations: Gets bool, log, and config.
HID-> Game-Specific Subsystems: Works with race_manager, gets kart and track data.
HID-> Network: Works with rewinds.
HID-> Physics: Gets physics for input_manager.

Physics-> Gameplay Foundations: Uses the scriptengine to run functions, uses config and angelscript.
Physics-> Game-Specific Subsystems: Gets mode, items, race_manager, achievements, karts, and tracks.
Physics-> Graphics: Gets animations and models.
Physics-> HID: Uses Get and Xml Node.

Libraries-> Gameplay Foundations: Gets config
Libraries->Game-Specific Subsystems:Connects to states_screens, modes, replays, items, race, karts, and main_loop.cpp
Libraries->Graphics: Calls graphics and guiengine functions
Libraries->Physics: Sets Debug Mode

Conceptual-Concrete Comparison:

Clearly the largest difference between our conceptual architecture and our concrete architecture is that the concrete architecture is much closer to being a complete graph, with each subsystem depending at least three other subsystems, we suspect that this is not the result of necessity, but rather a lack of unified design.

The other large changes are the increase of action with the network subsystem, taking a much larger role than we had anticipated. In our conceptual architecture the network subsystem was an incomplete module that, if finished, would simply share information with the Game-Specific Subsystems. After taking a closer look it was revealed that the network subsystem was already serving some function, and that there was in fact data stored there necessary for the game to manage profiles and update the Game-Specific subsystems.

Design Patterns

There were a couple of design patterns that could be seen when analyzing the concrete architecture.

Facade Pattern:

In the main.cpp, the function setupRaceStart() sets up the kart for the player but it will decide what exactly to call as the inputs for the setPlayerKart parameters depending on kart_properties_manager. This is a facade pattern because it hides the complexity from the function that calls it. The following is the code from setupRaceStart():

```
void setupRaceStart()
{
.
.
.
    if (!kart_properties_manager->getKart(UserConfigParams::m_default_kart))
    {
        Log::warn("main", "Kart '%s' is unknown so will use the "
            "default kart.",
            UserConfigParams::m_default_kart.c_str());
        race_manager->setPlayerKart(0,
                           UserConfigParams::m_default_kart.getDefaultValue());
    }
    else
    {
        // Set up race manager appropriately
        if (race_manager->getNumPlayers() > 0)
            race_manager->setPlayerKart(0, UserConfigParams::m_default_kart);
    }

    // ASSIGN should make sure that only input from assigned devices
    // is read.
    input_manager->getDeviceManager()->setAssignMode(ASSIGN);
}   // setupRaceStart
```

Template Pattern:

Looking at the Kart subsystems, we can also see that there is a template pattern adopted by the karts. This is done in such a way that karts use abstractkart.cpp as a base to build off of. This implies that each entity kart implements the functions differently. This is a classic template pattern.

Assigned Subsystems And Interactions

Game Specific Subsystems

Game specific subsystems includes general logic for game functionality, managing and integrating the more specialized subsystems such as graphics and physics. As this is the central component of the architecture, we expected it to essentially depend on most of the other subsystems.

Graphics

The graphics subsystem is used to draw game content, handle animations, and includes the code used to define and render the graphical user interface. Though we expected graphics to depend primarily on libraries, calling functions from Irrlicht, it additionally depends on the following subsystems:

- Audio
- Game Specific Subsystems
- HID
- Physics
- Gameplay Foundations
- Network

Part of the reason for the numerous dependencies of the Graphics module is that various functions for visualizing functionality of other modules for debugging purposes were written directly into Graphics. Additionally, the graphics module defines a number of effects and other more complicated elements that involve elements of other subsystems such as playing the associated sound for a visual effect. The GUI functionality also requires access to elements defined in a variety of subsystems in order to represent the associated information to the user.

Gameplay Foundations

The Gameplay Foundations module contains a variety of basic foundational elements of STK. It defines the system for interpreting configuration options defined in config files. As many other systems in STK reference these options, most of the other subsystems depend on Gameplay Foundations. Gameplay Foundations also contains the scripting engine and the Angelscript library used to script game elements without requiring separate compilation for different platforms as well as the system for downloading and installing addons.

The majority of Gameplay Foundations' dependencies on other modules are due to the scripting engine. Because the scripting engine requires access to functionality in a number of subsystems in order to make them programmatically available to script writers, it consequently makes Gameplay Foundations dependent on Game Specific Subsystems, Graphics, Audio, HID, and several libraries. Additionally, the configuration system references the network subsystem in order to access online player profiles.

Audio

The Audio subsystem has relatively few dependencies and unlike other important components of the game appears to be fairly well separated. The primary dependency seems to be with libraries as was expected. Audio additionally has a few dependencies on Game Specific Subsystems notably calling the Track::addMusic method. It is also dependent on Gameplay Foundations, referencing the configuration functionality.

Network

The Network subsystem is responsible for the online aspects of the game, such as the friend list. As the online aspects of the game are still a work in progress, the Network subsystem is incomplete. We expected Network to depend exclusively on Game-Specific Subsystems, Libraries, and Utilities, but we found that it additionally has dependencies on Gameplay Foundations to get currentPlayer, Graphics and Pysics for rewinds, and HID to get deviceManager, input_manager, and input_device.

HID

The HID (Human Interface Device) subsystem is responsible for receiving and handling input from the various input devices usable with STK such as keyboards, mice, and Wii controllers. We expected HID to depend exclusively on Libraries and Utilities. We found that it additionally has limited dependencies on most of the other subsystems. HID depends on Graphics as it must apply input to objects defined by the Graphics subsystem such as the camera.

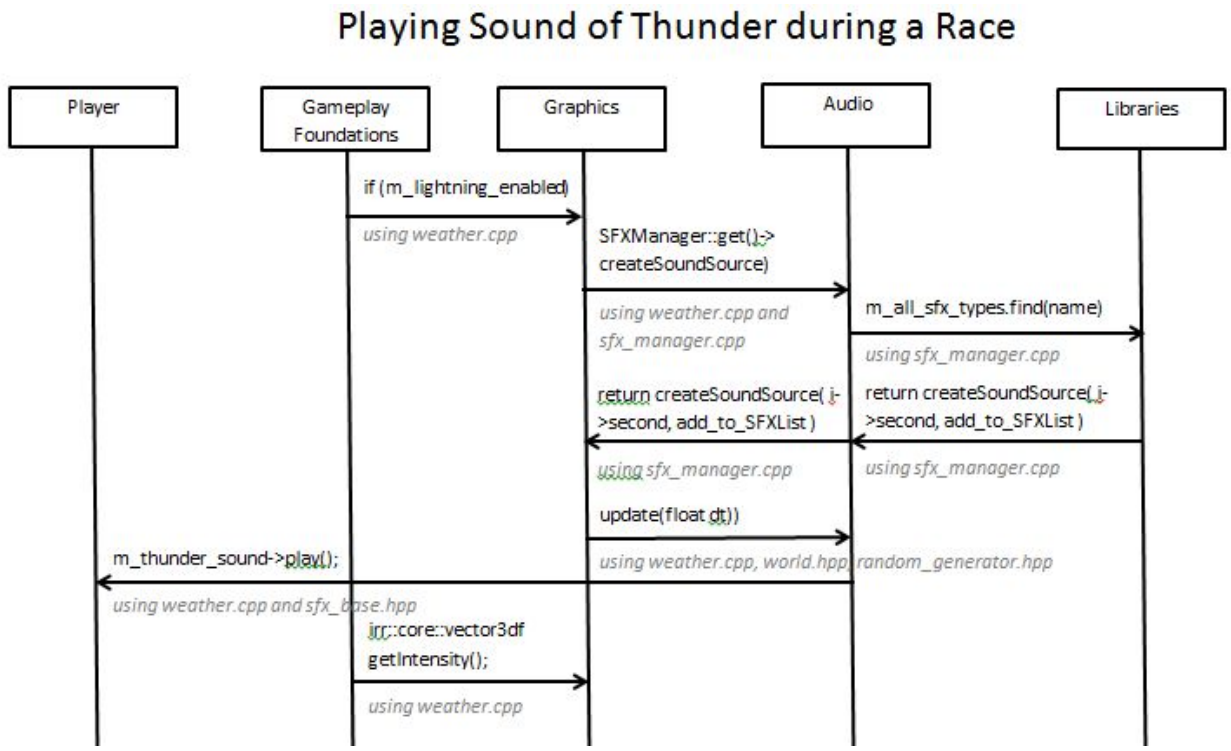
Physics

The Physics subsystem is responsible for the wrapping over the actual physics of the game from Bullet. We expected Physics to depend exclusively on Libraries and Utilities, but found that it had dependencies on most of the other subsystems. It depends on Gameplay Foundations to use the scriptengine to run functions, GSS to get mode, items, race_manager, achievements, karts, and tracks, Graphics to get animations and models, and HID to use Get and XmlNode.

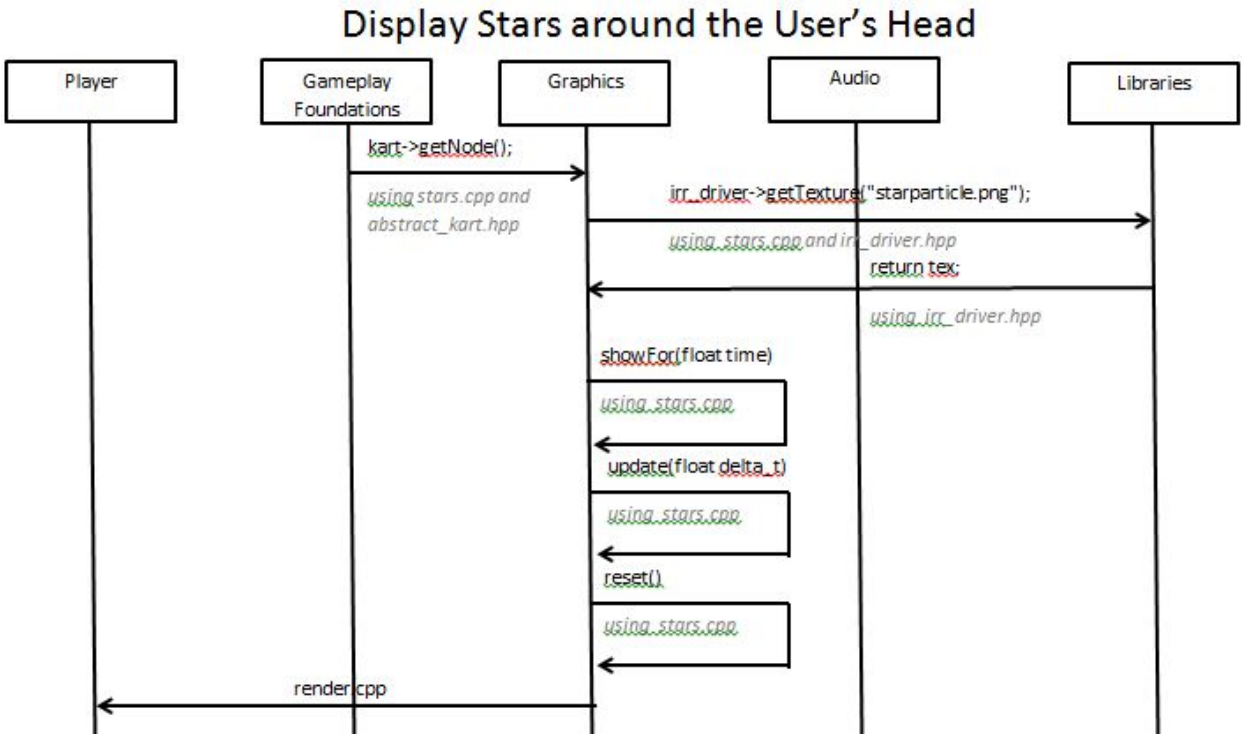
Libraries/Utilities

The Libraries subsystem holds all of the libraries and third party tools that were responsible for many of the core functionalities of the game. The utilities subsystem holds many utility functions that are used by almost every other subsystem. It couples highly with every other subsystem. At first, we expected only a libraries subsystem, but we found that there were many components in the subsystem that seemed to have a two way dependency with other subsystems in the architecture. This led us to rename and separate the Libraries into Libraries and Utilities so as to better demonstrate the differences between the components inside of the subsystems.

Example Sequence Diagram :



This sequence diagram is for playing the weather (thunder) sounds that correspond to it. It first checks if lightning is enabled, if it is true, then createSoundSource is called from the SFXManager with the thunder parameter. If it is false, then the same happens with a string indicating which sound is wanted. This file is returned if found or an error message “SFXManager::createSoundSource could not find the requested sound effect :” would be returned. A random number is generated and will be used to determine when to play and display the thunder. Update() is called and this is the main loop of the diagram where the random number is decreased by dt and if it reaches 0.0, then the sound is played and another random number is generated. After that we get the intensity in proportion to the random number generated to simulate the longer the wait, the stronger thunder is.



This is a sequence diagram for displaying stars on top of the user's head after getting hit by an item. First it gets the node of the user's kart. After that the library component is referenced to look for the star particle image to show above the user and will be returned to graphics if found. The method showFor(float time) is called with the parameter time to indicate the remaining time left to display the star image. After that update() is called with the parameter delta_t to decrease the time in the showFor parameter. If time is less than 0, then setVisible() will be false and user will not be able to see the stars. If time is greater than 0, then the stars will move in a circular motion where it will also fade in and out. Update will be called again to check if the time parameter is less than 0. After that, reset() is called to make the stars invisible and disable them.

Concurrencies

SuperTuxKart, as stated by the developers, requires multi-core processors to run optimally. As such, we have determined that the game supports multithreading, and with multithreading support, we can identify some of the concurrencies within our analyzed concrete architecture.

User input, also known as the HID in our concrete architecture, is in its own thread when the program is run, meaning that one of the cores within the processor is dedicated to handle this subsystem. This is logical as the game must be ready to process all user inputs at all times. Other concurrencies in the architecture include the sound subsystem and the graphics subsystem. Since report 1, we have established that these two subsystems are most likely

concurrent, and the concrete architecture shows us that this is true, as graphics and sounds do not require any kind of data transfer between them.

Development Team Issues

In examining the concrete architecture of STK, we can speculate on the issues that may have contributed to the disorganized state of the current architecture. As an open source project, the team of developers who contributed to the project were not full time workers and as such, the overall team vision for the project has changed many times over. This could have possibly led to some illogical dependencies in the project due to junior developers choosing to bypass architectural dependencies for the sake of ease. This illustrates the necessity of a software development documentations and an architectural diagram to force junior developers to consider the issues that arise from bypassing architectural diagrams.

Lessons Learned

We learned, while creating a concrete diagram for SuperTuxKart, that Understand is a useful and powerful tool. It is capable of quickly analyzing code and determining dependencies between components that we can modify. This allows the user to quickly and easily determine the concrete architecture of a source code. While performing a reflexion analysis of the source code, we also realized that designing a game engine architecture prior to starting coding is essential to mapping out any possible issues that may arise in the future.

Although Understand is a useful and powerful tool, we also realized that there are many issues that may arise from using it as well. For example, much of source has to be hand placed into subsystems or components which is an easy vector for errors to perpetrate into the dependency diagram. It also crashes many times depending on the source code you are trying to analyze and some difficulty can arise from trying to manipulate the subsystems in a natural way (trying to move subsystems or looking into dependencies inside of subsystems).

Another lesson we learned is that the open source aspect of the source code differs a lot from non-open source source code we have seen. Because of the fact that many different programmers have contributed to the game, it can lead to inconsistencies within the source code, such as comments and architectural styles in mind. Although we had trouble with the wide variety of coding styles, we learned more about open source projects in general.

Overall however, Understand is extremely useful and if you are jumping into a project halfway through its development cycle, it is very important to analyze the concrete architecture and the dependencies behind the system in order to better code for the project.

Conclusion

Our research into the concrete architecture of SuperTuxKart has given us some insight into the concrete architecture as well as how the development team decided to implement certain aspects of the source. While we initially thought of the architecture as a layered Object Oriented style architecture, we found that through our concrete architecture analysis, we concluded that the coupling in the source code only allows for an Object-Oriented architecture style. The source code contained many different subsystems which were very coupled with nearly every other subsystem. The development team had certain issues which are highlighted even by the concrete architecture and has been made clear in this report.

Appendix

Glossary

Architecture - *Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*[5]

Component - A part of a system which performs its own set of functions.

Dependency - The need for a component to use the functions of another.

Game State - A snapshot of the game containing all of the current objects and values, if either an object or a value changes a new game state is created.

GUI - short for “Graphical User Interface” it is a visual component with which the user interacts with the software.

Multi-platform - A term describing software that can be used across many different devices

Open source - An open source project has all of its source code (and often its resources) available to the public for modification or further development by anyone.

SDK - A software development kit is a set of tools that enable creation of software for a platform or system.

Subsystem - A component created from a collection of smaller components that work as a system on their own

References

1.Gregory, J. (2009). *Game engine architecture*. Wellesley, Mass.: A K Peters.

2.Super Tux Kart Documentation: http://supertuxkart.sourceforge.net/doxygen/?title=doxygen

3.Super Tux Kart Source: https://github.com/supertuxkart/stk-code

4. SuperTuxKart development blog: http://blog.supertuxkart.net/

5. IEEE Computer Society (2000). *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems: IEEE Std 1472000*. (also known as IEEE 1471)