Pangolin: A Look at the Conceptual Architecture of SuperTuxKart

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

Abstract

This report will be taking a look at the conceptual architecture of SuperTuxKart. We will show our rationale behind how we derived our conceptual architecture and the research that went into it. We will also take a look at some of the initial and improved hypotheses for the conceptual architecture that we developed.

Every subsystem in our architecture will be thoroughly explained and analyzed. We will show the importance and the reasoning for each placement of the subsystems as well as the dependencies. Certain sequence diagrams and use case diagrams are also present in this report to aid and display the use of each elements of the architecture.

Finally we will talk about the concurrencies in the architecture and certain development team issues that may have occurred. This will lead into a discussion on what we learned from this study of SuperTuxKart's architecture. In conclusion, this report gives a rationale and in-depth explanation to SuperTuxKart's conceptual architecture.

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.

Conceptual Architecture Derivation

Derivation Process:

To begin the derivation process, we analyzed the reference architecture provided in class (similar to the reference architecture provided in Game Engine Architecture 2nd edition[1]). After which, we delved into the documentation[2] of SuperTuxKart and found a high level module interaction diagram. By identifying key features in the reference architecture and key features in the high level module interaction diagram, we began creating a picture of the conceptual architecture of SuperTuxKart.

Looking through SuperTuxKart and the high level module interaction diagram, we found that certain key features were placed among different subsystems than in the reference architecture and that some key features were not even mentioned. For example the audio subsystem is not mentioned however is necessary to the game's functionality.

Taking major elements of SuperTuxKart, we boxed them into the proper elements of the reference architecture. Through conceptual analysis of the interactions between the major elements, we determined a plausible dependency structure for the architecture.

Initial Hypothesis:

As we derived the conceptual architecture, we derived a layered object oriented style and expected it to have many different elements separated from one another. When we took a closer look however, we found that some elements of this structure did not make sense. A fully layered architecture made more sense and elements that were more coupled could be merged.

This led to almost a complete scrap of our initial hypothesis. We used to have a network element for future implementation, but removed this as well to improve the simplicity of our architecture.

Improved Hypothesis:

Taking into account what we now know of the game's architecture and key elements, we decided on a layered object oriented style of architecture. Components in high levels of the system should not interact directly with the components of the low level system.

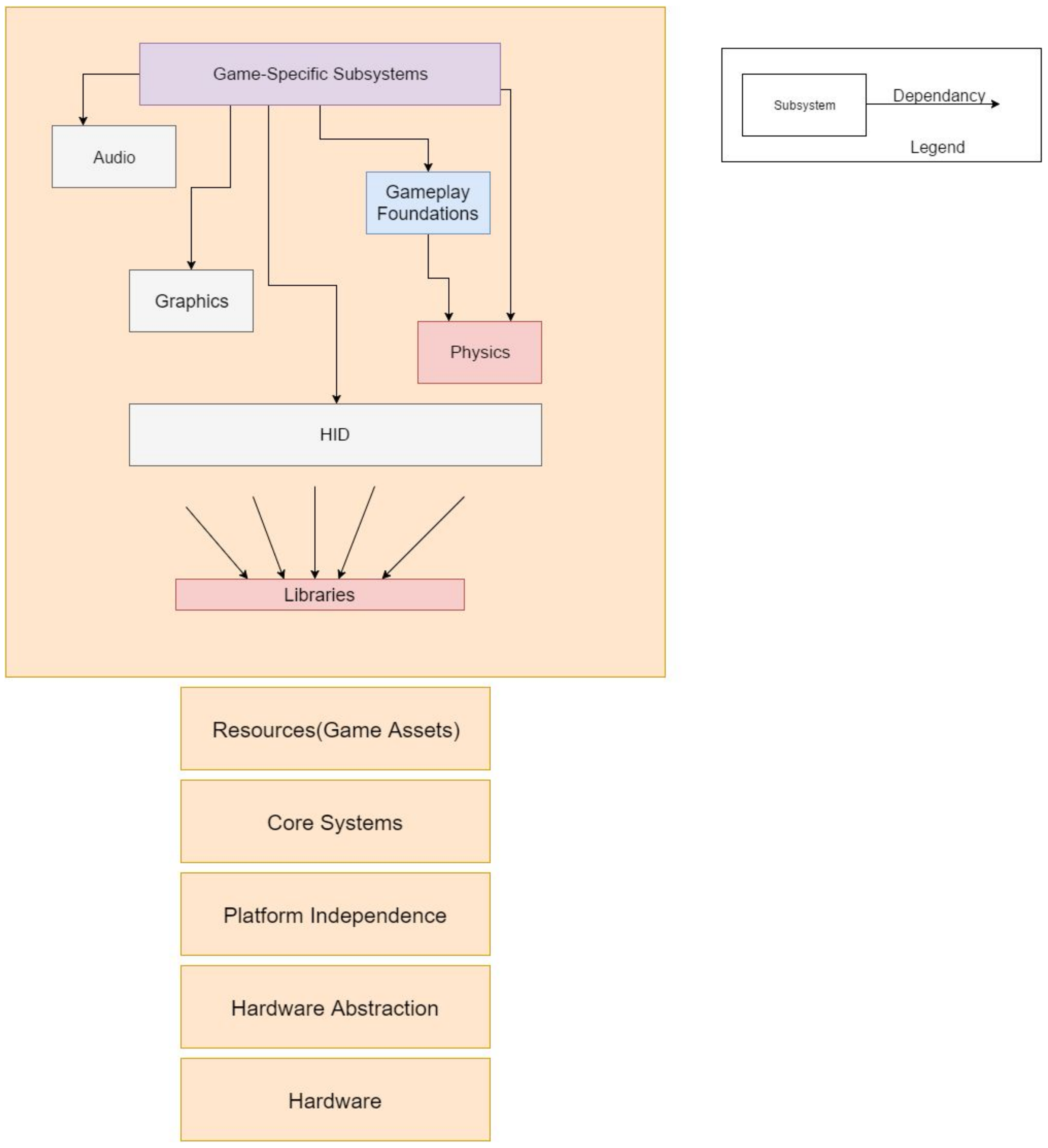


Figure 1. Above is our improved hypothesis for the conceptual architecture.

Conceptual Architecture

What follows is a breakdown of all of the subsystems in the architecture, including their function and their dependencies.

Game Architecture Subsystems

Game Specific Subsystems:

Includes elements such as items, tracks, karts, races, modes, AI, pathfinding, game cameras, game state information. Kart controller,etc. are also included here. Includes main game loop.

Depends on audio to play the specific kart/music/item sounds

Depends on graphics to draw out all elements of the race as well as gui.

Depends on HID to receive the human input in order to update game state.

Depends on Gameplay foundations to contain the scripting system used, to maintain static world elements.

Depends on Physics to obtain collisions and force effects.

Depends on libraries as it contains all static necessary libraries.

The Game Specific Subsystems include the core functionality of SuperTuxKart. It is the most integral component and highest level layer of the conceptual architecture, handling a wide variety of problems and dependent on all the other layers. It defines the actual game mechanics in the abstract, or the win conditions and rules of the game. As they relate to the essential elements of a mario kart-esque racing game, such as tracks, karts, and items. Also handled by the Game Specific Subsystems are artificial intelligence and pathfinding.

Audio Subsystem:

Subsystem that plays the audio of the kart/music/item sounds.

Graphics Subsystem:

Like many games, the conceptual architecture of Super Tux Kart separates rendering logic and rendering code from the actual game logic. It depends on Irrlicht, a third party open source renderer for games, to perform all actual rendering of game content. The graphics subsystem acts as a thin wrapper over Irrlicht, handling cameras and managing the materials and shaders used to render the game content. The graphics subsystem additionally builds up additional functionality not provided natively by Irrlicht, particularly a system for particle effects and a variety of visual effects both based on the particle effects system and independent of it.

The game specific subsystems handle the management of game assets and loads them directly into Irrlicht.

The Graphics Subsystem also contains the gui engine used in the game. SuperTuxKart appears to have written it’s own widget toolkit for use in it’s user interface. A widget toolkit is a set of separate user interface components or controls that can be nested and are used to construct the user interface. The user interface system calls on Irrlicht for rendering, presumably compositing the 2D user interface sprites onto the rendered 3d output.

Human Interface Devices (HID):

Contains game specific interfacing such as keyboard/mouse/gamepad/Physical Device input.

Gameplay Foundations Subsystem:

SuperTuxKart includes AngelScript, a scripting language similar to C++, but with garbage collection and no support for pointers. AngelScript is compiled to bytecode and run via an interpreter included with Super Tux Kart. Consequently, content programmed in AngelScript does not need to be separately compiled into native code for each platform (hardware and operating system) targeted, and can be distributed in a single form. This is extremely useful functionality for add ons, as it ensures that they are cross platform.

Physics Subsystem:

Since writing a physics system is mathematically intensive, the vast majority of games use a third party physics engine, to the extent that this arrangement could be considered a normal feature of the reference architecture. Super Tux Kart uses Bullet (http://bulletphysics.org/wordpress/), an open source engine that is widely used in a variety of industries, notably by Rockstar Games in their Grand Theft Auto and Red Dead Redemption series.

The Physics Subsystem acts as an interface between the Game Specific Subsystems and Bullet Physics. Indirectly, through Bullet, the Physics Subsystem is responsible for detecting collisions and intersections between mesh objects in the game. It also handles the dynamic simulation of physics, at least in approximation, and physical interactions between objects, such as friction, and forceful collisions. In fact, all motion of karts and other game objects must be dealt with through the framework of the physics system. It is therefore integral to the game.

Libraries:

Contains any static or shared libraries required by the programming language as well as any SDK required by the system.

The Libraries submodule represents the various third party dependencies of the game, either provided as shared or static libraries by the operating system, or distributed with the game itself. These include one or more C++ standard libraries, the Irrlicht and Bullet libraries, OpenGL, and anything else that may be required.

Resources (Game Assets):

The resources subsystem represents any textures, pictures, meshes, xml files, audio files and such that may be required by the system. They are all formatted in such a way that the game engine may use them and will be, at some point, presented to the player. These are often buffered into the level.

Core Systems:

The core system of SuperTuxKart contains a variety of often used timers, containers and generally useful code. Features such as startup, and some memory management are controlled by this subsystem.

Platform Independence:

SuperTuxKart abstracts over many different multi-core processors of different operating systems. This allows for an independence layer separate over the operating system allowing it to perform well regardless of the operating system the user may have.

Hardware Abstraction:

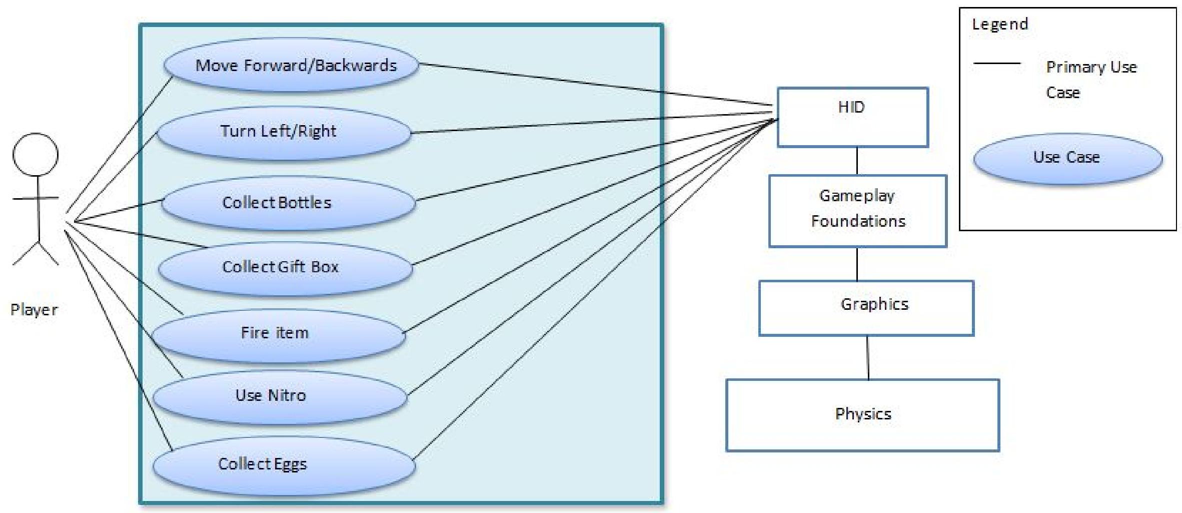
Any abstraction of the hardware to software interactions are located here (including drivers and such).

Hardware:

Hardware owned by the user. Often of many different variety and different brands. Requires a hardware abstraction layer to perform.

Player Use Case Diagram Explained:

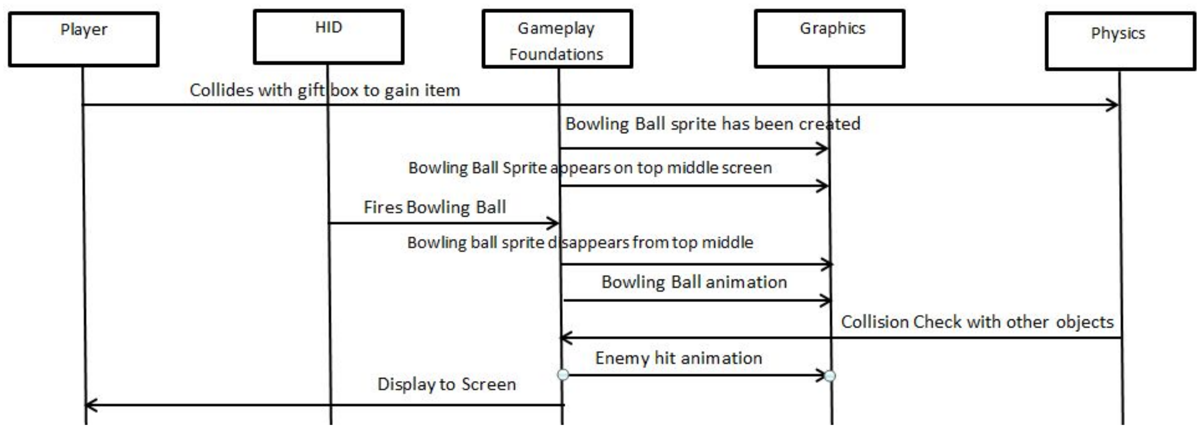
This is the Use Case Diagram for when the player wants to input a command into the game. The user inputs a command and a use case that corresponds to the choice is selected and is sent to the HID subsystem. Then these objects that were generated from actions done by the player are found in the gameplay foundations component. Then this data is sent to the graphics component to put this data where these objects will be seen and be moving. After that, it will go to the Physics subsystem where it detects collisions between the objects, takes gravity into consideration, decides how fast the projectile is moving depending on weight and much more. After that, we will wait for the next input that the user wants.



Example Sequence Diagram :

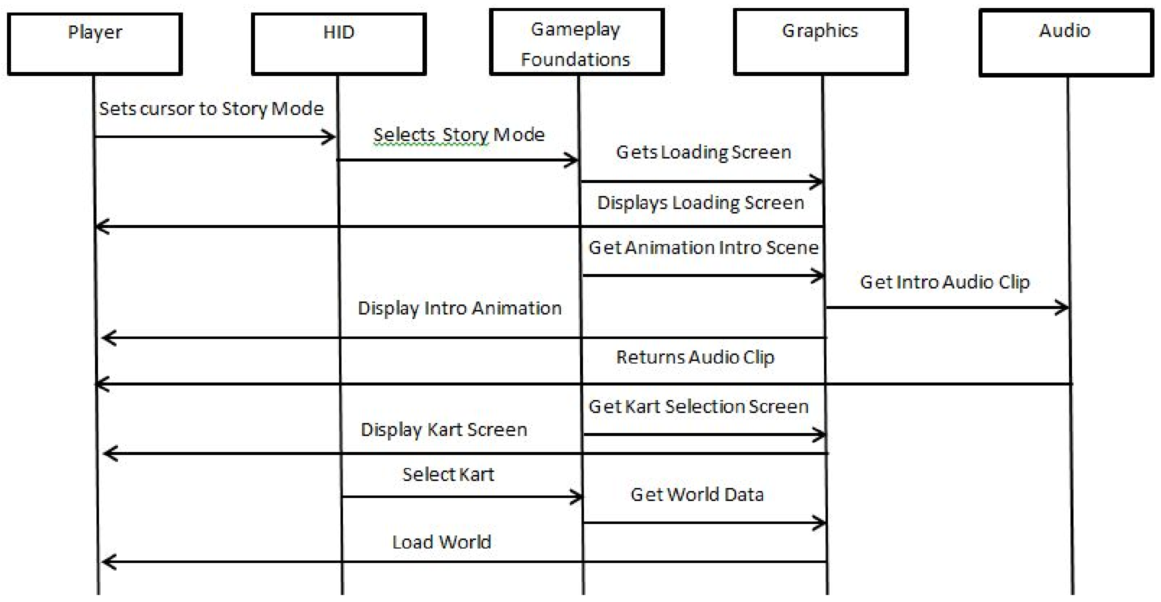
This sequence diagram is for getting an item (bowling ball) from the gift box and launching it forward and hitting an opponent. We see that in this sequence, we use the components other than the networking and game manager components. First we use collision detection to see if the player has hit the gift box to get an item or not. Then the item object is created from the gameplay foundations component and sent to graphics. The player has input to fire the ball and the animation is created. When the player fires the ball, the graphics for the sprite telling you the item you have will disappear. In order for the attack to be successful, we use the physics component to check if it has a collision with any of the things in the game. In this case, it does and the enemy hit animation is created and displays back to the player to see.

Retrieving a Bowling Ball from a Gift Box and Hitting an Opponent with it


Player
HID
Gameplay Foundations
Graphics
Physics
Collides with gift box to gain item
Bowling Ball sprite has been created
Bowling Ball Sprite appears on top middle screen
Fires Bowling Ball
Bowling ball sprite disappears from top middle
Bowling Ball animation
Collision Check with other objects
Enemy hit animation
Display to Screen

This sequence diagram is for selecting the story mode option for the first time. First, the user selects the story mode option and then a loading screen appears. For this to happen, the game gets its data from the Gameplay Foundations component and sends it to Graphics. Then a short introduction movie is found in the gameplay component while an audio clip is found in the audio component and is sent back to the user. Kart Selection Screen is retrieved from the gameplay foundations and the user inputs their choice of kart. Then the World data is achieved and loaded back to the user to see.

Selecting Story Mode and Kart Selection



Concurrencies

Concurrency can be identified in the architecture by looking at two subsystems which do not interact with each other, but share information with a common subsystem. Taking a look at the architecture, the following systems run concurrently:

- Audio and Graphics
- Physics and Graphics
- Gameplay Foundations and Graphics
- Gameplay Foundations and Audio
- HID and Physics, Graphics, Audio, and Gameplay Foundations

Most of the subsystems in the conceptual architecture run concurrently; they do not interact with each other (with the exception of Gameplay Foundations and Physics), as there is no need for any sharing of information between these systems, they are all managed by the main game manager (Game specific subsystems).

Development Team Issues

In 2004, due to team disagreements and fallout during the development of SuperTuxKart, the game was unplayable and unsupported for two years until it was picked up again in 2006 by another team.

Looking at the developer's blog, we can also identify some development issues during the switching on 3D engines in 2010:

- A total rewrite of the GUI was needed, which was very time consuming and the development was essentially making a brand new game.

- Because of the fact that the game is open source and free to play, the development team asked fans of the game to test it instead of hiring professional testers. This resulted in many, many bug fixes as well as a lack of professional feedback.

- After over a year and a half, the game was finally completed, but this long duration means that a new, potentially better 3D engine was being developed as Irrlicht was being implemented.

Lessons Learned

Over the course of creating this conceptual architecture we learned several things. We learned to think critically about the flow of information, making sure that dependencies are both absolutely necessary, and are sometimes not a direct link between all components involved.

Our team had many discussions about what the architecture should be. We settled on a layered style very quickly, but the strictness of this style, and what components lay within changed frequently. We had to learn how to merge multiple different hypotheses into a more cohesive, simpler idea.

The game being Open-Source helped us in our efforts, as there was already some documentation of how some elements of the game interacted with each other. We had to learn how to look at these elements and determine which part of the architecture they should fall under, and as a result what dependencies we should be seeing in our diagram.

The most important thing we learned was to take a step back and look at what we had produced, fixing errors, simplifying ideas, and adjusting the structure of our diagram to better display the importance of components and their relations.

Conclusion

In conclusion, we believe that a layered style of architecture best suits SuperTuxKart's design. The reference architecture and documentation on the game helped to give us a clearer view of the components of the system, and their dependencies.

The main component of SuperTuxKart is the Game Specific Subsystems component. It in turn is dependent on every other subsystem using them to create and interact with the game world. Our initial hypothesis was far more chaotic, but a closer look at the game revealed that the design was actually more streamlined than we first thought.

In the architecture all of the dependencies are one-way, so a change to any module will require minimal changes to those dependent on it. This allows for easier modification and development in an open source game, which is the goal of the game as a project.

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)