Need for Tux Assignment 1

Conceptual Architecture of Super Tux Kart

Morgan Scott 10149124
Daniel Lucia 10156114
PJ Murray 10160261
Michael Wilson 10152552
Matthew Pollack 10109172
Nicholas Radford 10141299

Table of Contents

Abstract

SuperTuxKart is an opensource three dimensional kart racing game. It offers a diverse set of game modes, race types, character, and game mechanics. SuperTuxKart in terms of gameplay is very similar to popular racing games like Mario Kart or Crash Nitro Karts.

We found that like most games SuperTuxKart adheres to a Object Oriented architecture style resulting in high cohesion and low coupling. Its major subsystems include Physics, Gameplay, Graphics Systems, Utilities, and Game Support Systems. The physics subsystem handles all the math based operations of the game including driving a vehicle and launching weapons. The Gameplay subsystem contains all the game specific rules and mechanics. The Graphics systems handles the rendering and animations performed by the game. This subsystem uses a third party graphics engine, called Irrlicht. The Utilities subsystem contains all the non-specific game components such as audio and the software development kit. Lastly the Game Support system includes all the components that affect the game and influence the gameplay.

The game surprising does not have a networking subsystem. This subsystem is still in development and currently its capabilities only allow for saving user game data that can be shared amongst users.

Introduction and Overview

SuperTuxKart is an open source three-dimensional kart racing game, that includes a varied selection of characters, race modes, and tracks.

This report will give a detailed description of the conceptual architecture of Super Tux Kart. It will explain the major subsystems and components of the architecture as well as discuss their relationships to each other. This document will focus on addressing the goals, requirements, evaluability, and testability of these subsystems.

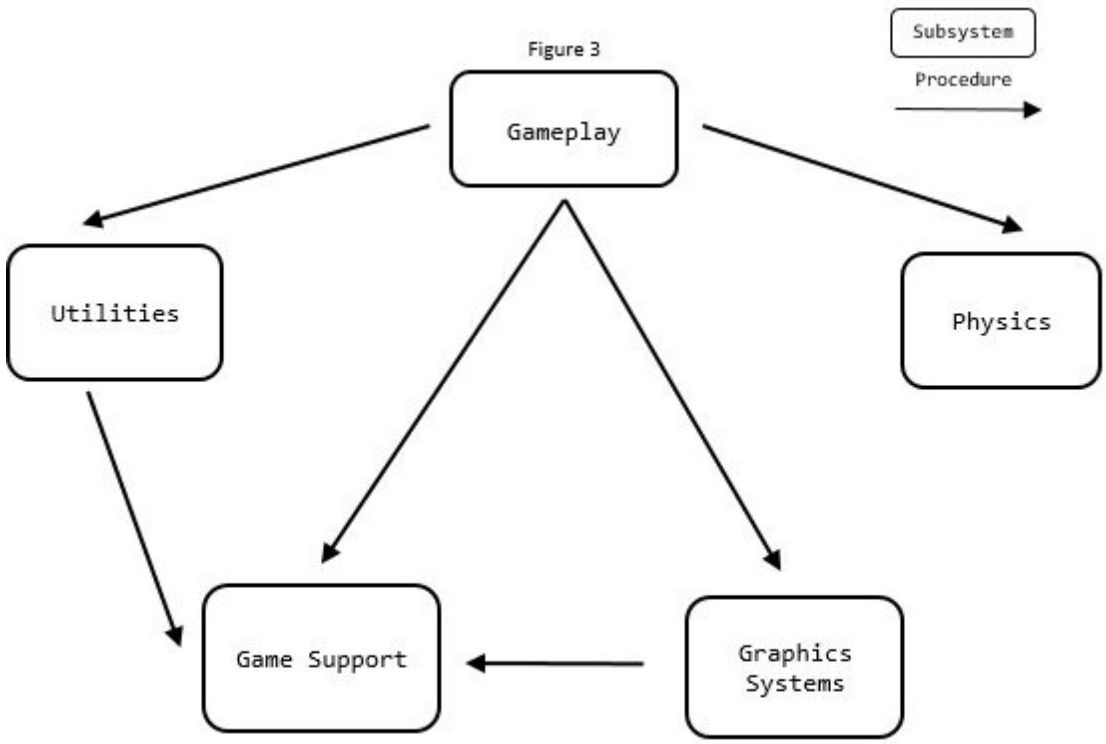
We will first explain our logic on how we derived our conceptual architecture. We will discuss the overall process, analysis and conclusions of the development of the conceptual architecture. Secondly, we will explain each of the major subsystems of the game in detail and discuss the relationships and communications them. This report will use various diagrams including sequence diagrams to demonstrate the actions of all the systems in response to user interaction. Thirdly we will explain how the overall system support future changes and alteration to the software. Lastly, we have included a lessons learned section

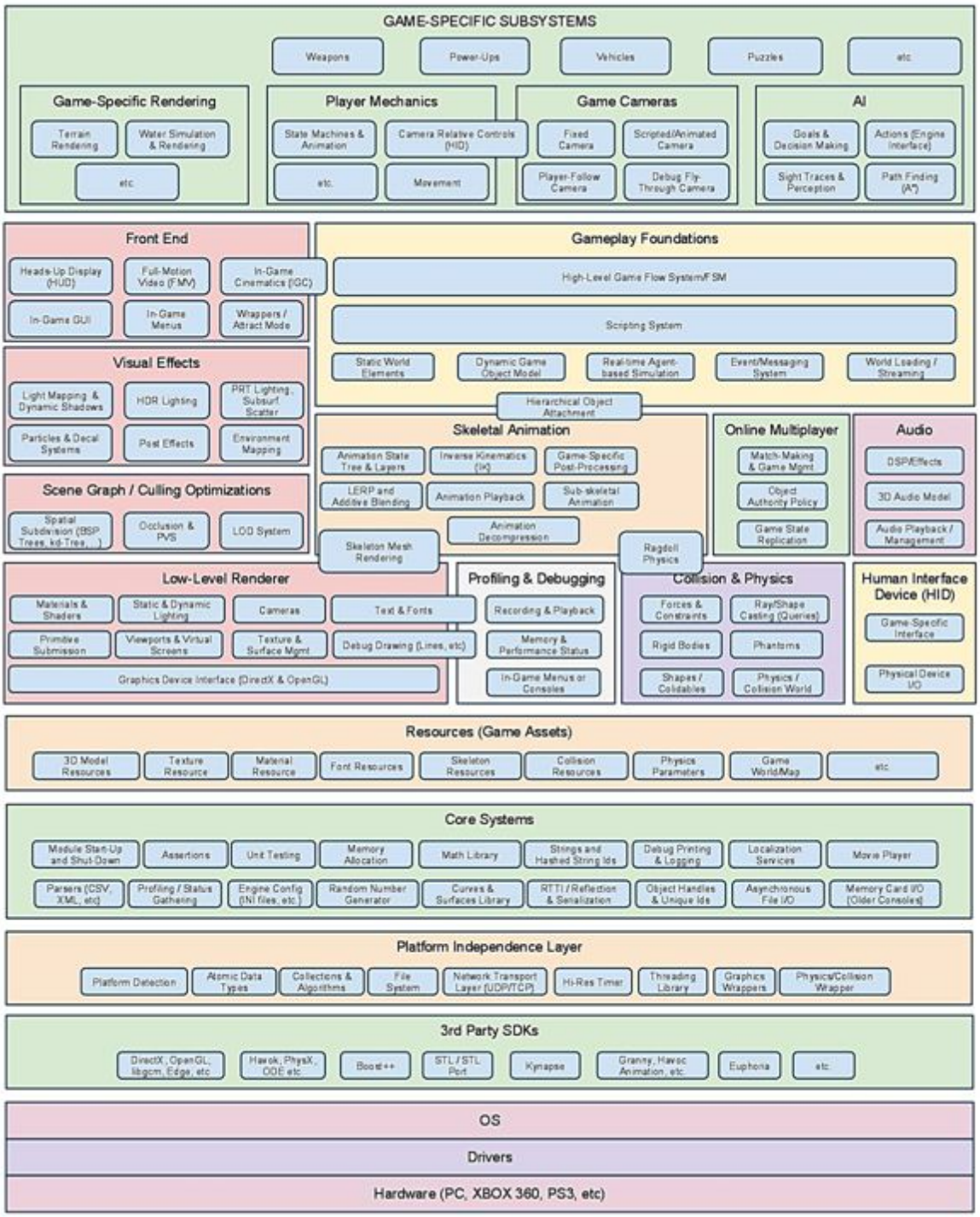
that contains beneficial information on how we would re-approach the conceptual architecture and lessons for the future.
Through our analysis of the conceptual architecture of SuperTuxKart, we have found that its architecture is of the object oriented style consisting of five major subsystems: Physics, Gameplay, Graphics Systems, Utilities, and Game Support System.

Derivation of Conceptual Architecture:

The conceptual architecture for SuperTuxKart was developed by first developing a reference architecture. This reference architecture was synthesis using Jeff Lander's and Matt Whiting's book of Game Engine Architecture. We used their runtime engine architecture diagram and provided explanations as resources for our reference architecture (see figure 1 below). With information, we decide on an architecture containing these seven major subsystems: utilities, graphics, physics and collisions, gameplay, networking, game UI, and game specific assets (see figure 2.)

Figure 1





From this reference architecture, we analyze the major modules of SuperTuxKart and decided which subsystems were relevant. From this we determined that the Physics, Graphics, Gameplay, Metadata, and Utilities were the best subsystems to describe SuperTuxKart (see figure 3). We decided our architecture style is object oriented. We determined this because components should not directly interact with low level subsystems. This style also makes sense because most game engines will

take on a layered style. Furthermore, this game was developed in C++ which is an object oriented language.

Alternate Architectures Considered

We considered using a couple different other architectures
Our first thought was repository, but found that our subsystems had only very direct interaction, so it would have made little sense to classify them all as one repository rather than breaking them up based on focus, and separating parts that had virtually no interactions.

We also looked at layered architecture, but found that the user interacts with many subsystems, and that there was no easy way to layer all the subsystems to limit access while maintaining efficiency. We could have layered our subsystems, but we found little to no benefit as we are not trying to protect data in any significant way, and we would have needed many indirect references to make it work.
Object Oriented was the middle ground between repository, which was too concentrated, and layered, which was too separated.

Figure 2

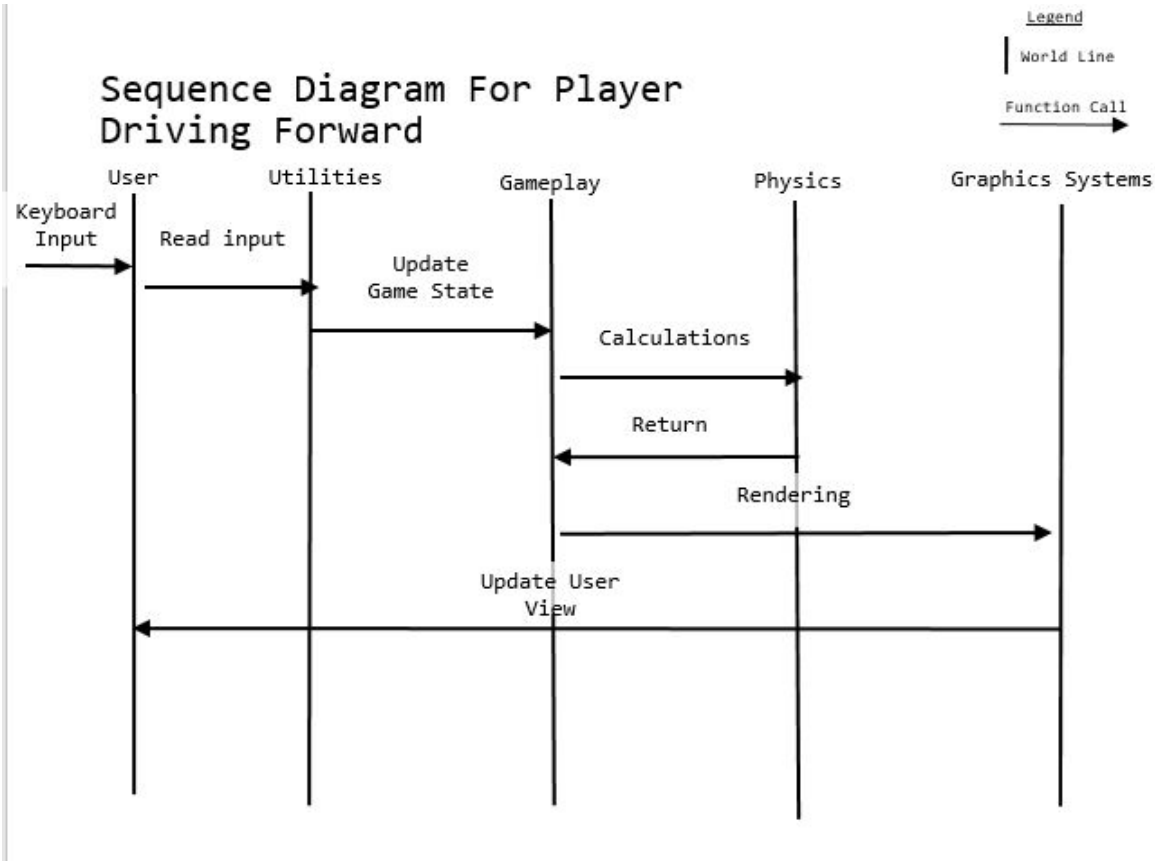
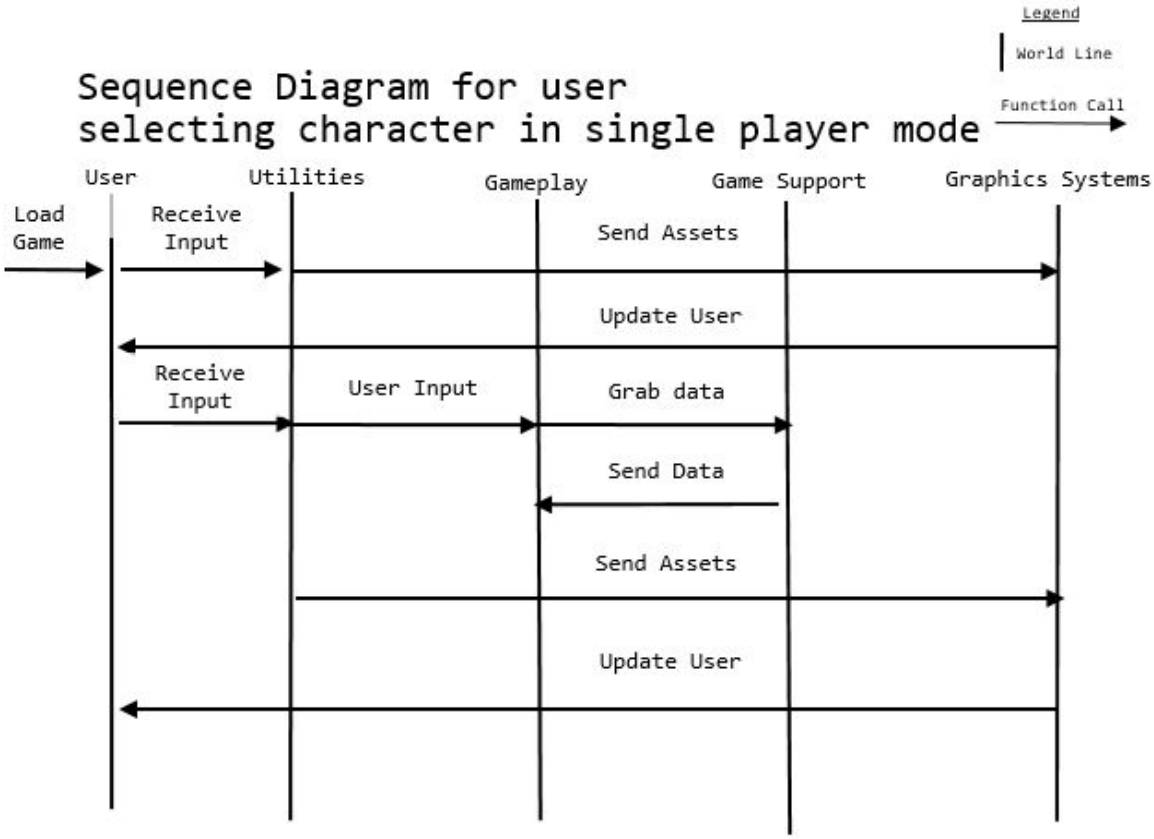


Figure 3



Modules

Game Play

The gameplay subsystem is responsible for handling how the player interacts with the game. It usually governs the rules and interface between the player and the world generated by the game (objects in game, challenges, goals and objectives etc...). We feel the Gameplay subsystem for supertux contains the the karts, track, modes, Items and race modules. The gameplay subsystem is also responsible for holding the game state that will be passed around to the other various major subsystems.

Karts: The kart module is responsible for handling all the kart information. This means it will be holding the kart models and the meshes as well as the properties of the karts. This will in short be managing all the karts on the track.

Tracks: the module handles all the track information, track objects, drivelines, and checklines. This would include information on the separate track objects, laps taken, and the terrain information.

Modes: The modes module governs all the different game modes offered by Supertuxkart. This would include logic for follow the leader mode, easter egg hunt mode, the soccer mode. Also handles the timings for all races (ie ready, set go) and countdowns. This module also handles running the actual game modes.

Items: The items module is responsible for the various collectables and weapons that you can find throughout the game. The items will holed the rules and behaviours of sed objects and let the gameplay subsystem know how to handle any items active in the game.

Race: The race module accumulates all the race and kart data from each race and records it appropriately. This will record the current high scores for the grand prix as well as the current rankings for karts during the actual race/mode.

Interactions Between Gameplay subsystem and other Major subsystems:

The gameplay subsystem is basically the governing subsystem that will communicate with all other subsystems to update the game rules and mechanics and use this information to update the game state.

Gameplay to Utilities- The gameplay subsystem will be passing in the input from the controller and using that input update the game state.

Gameplay to physics- The gameplay will pass the current game state as well as any objects that need to be updated to the physics engine so that it can calculate all the math done by the physics engine and update the game state.

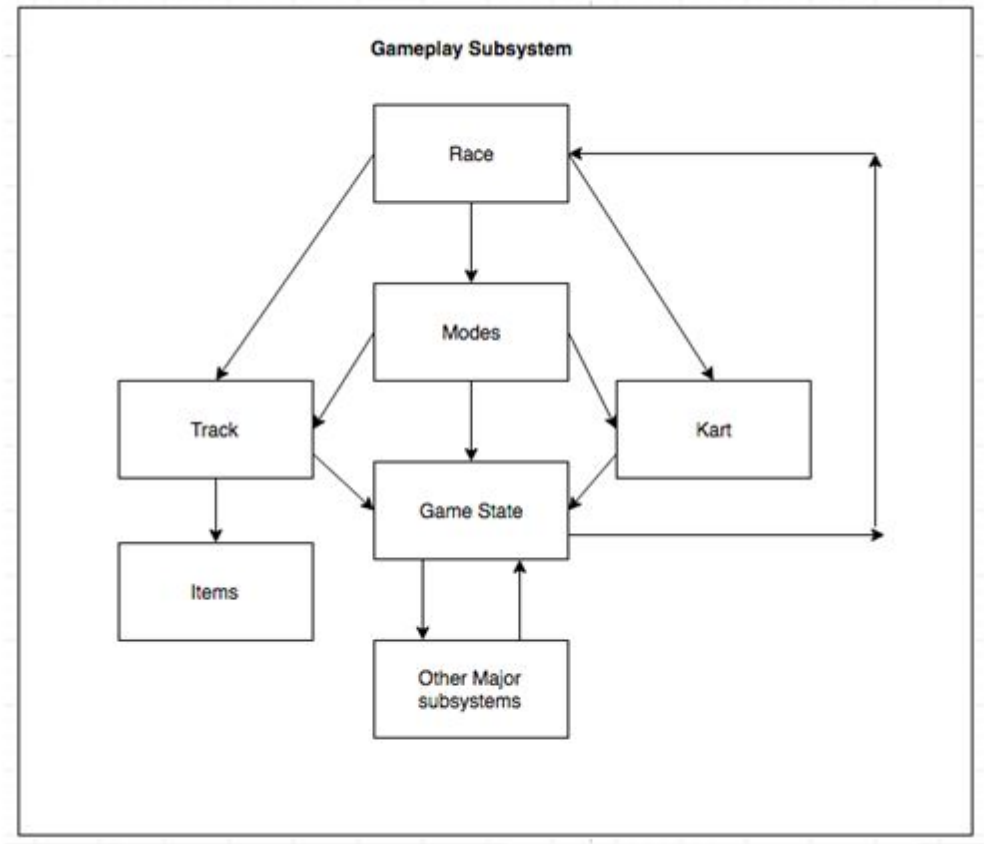
Gameplay to graphics- The game state can then be passed to graphics which will render the game based on the new game state. The rendered game state will be outputted on the screen

Gameplay to gameplay support - the gameplay will check with gameplay support for new content and rules that can be stored in gameplay and affects the various aspects of the gameplay subsystem

Interactions between the internal modules of the gameplay subsystem:

The Game state subsystem will hold the current gamestate of the game. The input from the controller will be passed in here from the other major subsystems. The game state will recognize new input and pass that to Race which will update the highscore and positioning data. The new data will cascade down through the other modules as shown in figure A updating the game state based on the new input and the rules in each respective module as stated above. Items is outside of track as the track will be holding all the karts ie the things the items act upon so they only need to be referenced. Once the new gamestate is created with the new controller input/ai input applied in karts it is saved to the Game state subsystem. This will then pass the gamestate to the physics engine to further update positioning of the objects in the state and then finally handed to the graphics system to be rendered.

(Figure A)



Graphics

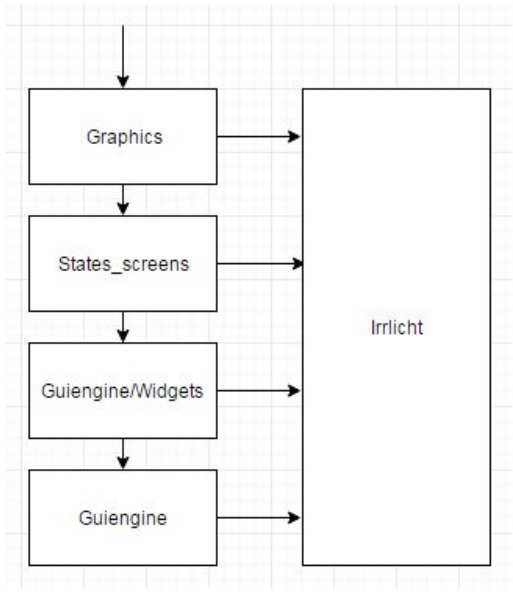
Graphics systems manages everything that is seen by the user at all stages of the game. This includes screens such as: main menu, stage select, and settings, and the actual racing visuals such as: karts, the track, and items. Without this system the

game would lack the important output required for the user to respond to the game mechanics. Graphics system heavily relies on a modified version of the Irrlicht to render platform independent graphics. By extension, graphics systems is middleware. Graphics systems is comprised of four subsystem: Graphics, Guiengine, Guiengine/Widgets, and States_screens. Each of these components rely on each other to build the context expected by the gameplay, and gameplay support systems.

How It Interacts Internally

Graphics systems has four major subsystems namely: Graphics, Guiengine, Guiengine/Widgets, and States_screens, and are defined as follows:

- Graphics: is the main entry point for almost all requests made to the graphics system. It manages the highest level of graphics interactions.
- Guiengine: encapsulates the main gui functions required to build and store gui elements. It interacts directly with irrlicht to accomplish it's tasks.
- Guiengine/Widgets: is simply a collection of predefined widgets presumably all unabstracted versions of the abstract widget class supplied by Guiengine.
- States_screens: manages creation of screens which are ultimately collections of widgets supplied by Guiengine/Widgets.



(Graphics Figure 1)

As seen in the diagram, each system talks to Irrlicht. The main entry point is labeled by the arrow at the top not starting from any system. This confirms graphics being the main entry point into the graphics system subsystem. In other words, it routes all commands to the appropriate cosystem.

How It Interacts Externally

Graphics is indirectly relied upon by almost all other subsystems. This is because graphics is the other major output medium beside audio. Since graphics is an output, it makes little to no requests outside its own subsystems. It primarily receives requests from Gameplay.

Graphics ← Gameplay: Graphics receives all kinds of requests from gameplay. This is due to gameplay being the major governing body of the whole architecture. Gameplay requests to graphics can be categorized in two ways: 2D and 3D. An example 2D request that gameplay would send to graphics could be: prepare the stage selection screen, then render it based on these unlocked stages. An example 3D request that gameplay would sent to graphics could be: prepare the world based on this stage, and with these karts, and these sets of inputs. An example 2D and 3D request that gameplay send the graphics could be: prepare the kart selection screen with these karts. These examples are higher level than what is truly being requested, and likely over simplify the data being sent back and forth. After each example graphics renders the request.

Graphics ← Gameplay Support: This interaction is not direct, as gameplay support goes through gameplay when talking to graphics, but is noteworthy as it supplies relevant context. Gameplay support manages things such as screen resolution and specific rendering settings such as antialiasing that graphics needs to know at launch. Graphics support also indirectly requests gameplay support when the gui captures the user changing the resolution on the settings screen.

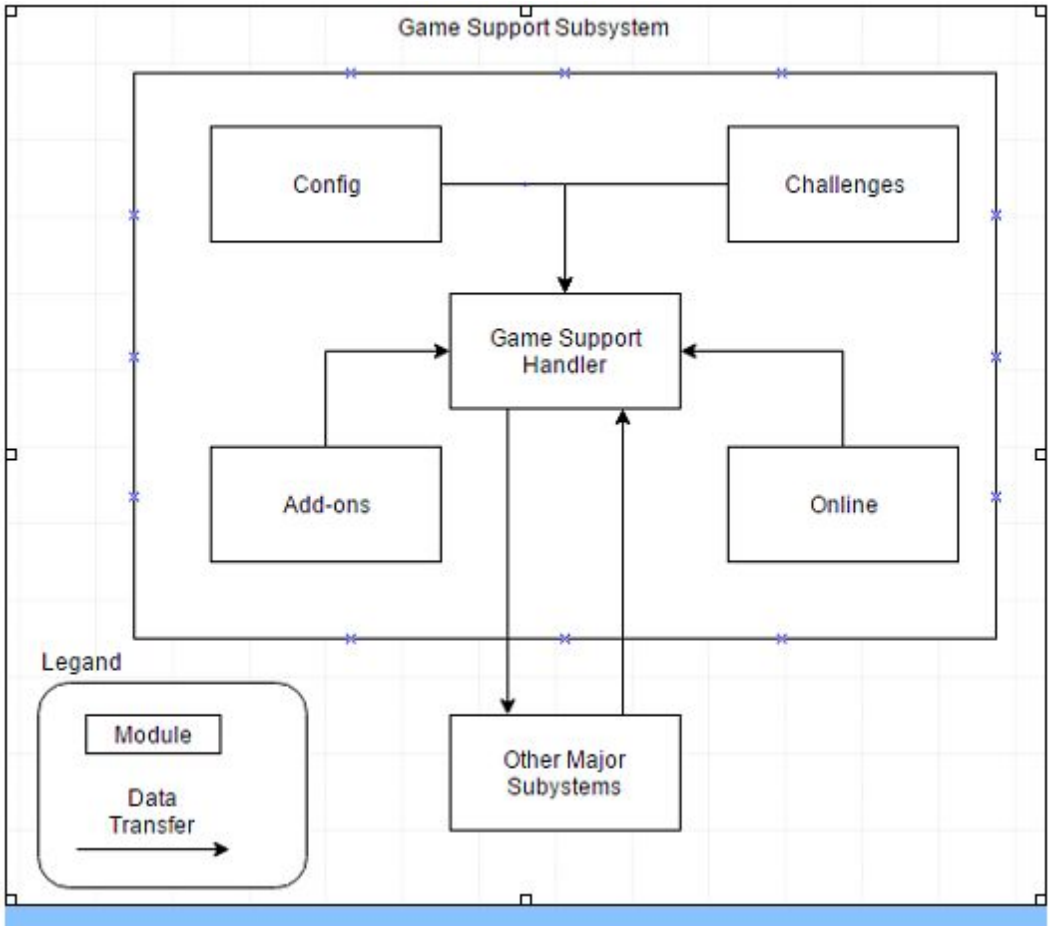
Game Support Subsystem

Game Support is a subsystem that is not entirely essential to the user's experience, but it can be of impact to the Gameplay of the game. It can enhance the experience for the user by implementing different features such as the configuration files, Challenges, add-ons, and an online mode.

The first module Config is very crucial to the user's experience. Config handles the user configuration which is not normally edited by the user. There are configuration files which the user does have control over and these consist of settings such as the resolution of the game, the graphics quality, and the key bindings. Another configuration file that the user has control over is the device configuration which are configuration files for the user's controller type, these files determine whether the user is using a keyboard or a gamepad. This setting will prepare the system to expect a certain type of input from the user, whether it is an accelerate action on the keyboard or on the gamepad. The Challenges module is a straightforward module in the sense that all it needs to do is provide the user with the challenge system. The games challenge system has control to lock different features depending on how

many challenges the user has completed. Features including but not limited to certain tracks, karts, and game modes can be locked until the user accomplishes certain objectives stated by the Challenge module. The next module in Game Support is the Add-Ons module. This module simply handles the downloadable content that the user has downloaded and uploaded to the game. The last module is the Online module. This module is currently in development and there is not actual online interface that allows you to face your friends in an online race. The only purpose this online module has is that it allows you to see the progress your friends have made in the challenge system.

When we derived the architecture for the game, the Game Support subsystem was connected to 3 other subsystems including Utilities, Gameplay, and Graphics System. Interactions between these systems are important for the user's experience. The interaction between Game Support and Utilities consists of the configuration files being requested by Utilities. Utilities requires the input settings as well as the audio settings that are stored in the configuration files. Gameplay would be requesting information pertaining to the challenge system so that the user can be shown what challenges they have completed and have yet to finish. Gameplay would also request data from the online system to output completed challenges of online friends. Certain configuration files would be of use to the Gameplay subsystem, these would include the rules and settings of the game. The last interaction with the Graphics subsystem is very simple, the data being transferred consists of sending the graphics and resolution settings. The Game Support Handler is the key to the interactions between these subsystems. It gathers the data requests from the other sub systems and retrieves the required data from the corresponding module located within Game Support.



Physics

The physics subsystem is responsible for handling all of the object interactions, collisions, and forces. It handles simulated friction, momentum, gravity and all the other forces needed to make the kart's handling feel natural and realistic. It also handles collisions of objects and kart control, as well as 3d animation. In our architecture, the physics subsystem contains the following modules: Physics, Karts/Controller, animation. The physics subsystem interacts directly with only two other subsystems, Gameplay and Utility, and interacts indirectly through gameplay with animation.

The physics subsystem's modules are outlined below:

Physics: This is the core of the physics subsystem, responsible for modeling collisions and object interaction. It also simulates forces such as friction and gravity, and other things like momentum.

Animation: This system manages the 3d animations that are grouped with the physics engine because of the reliance on physics. These animations are tied very closely to modeling interactions.

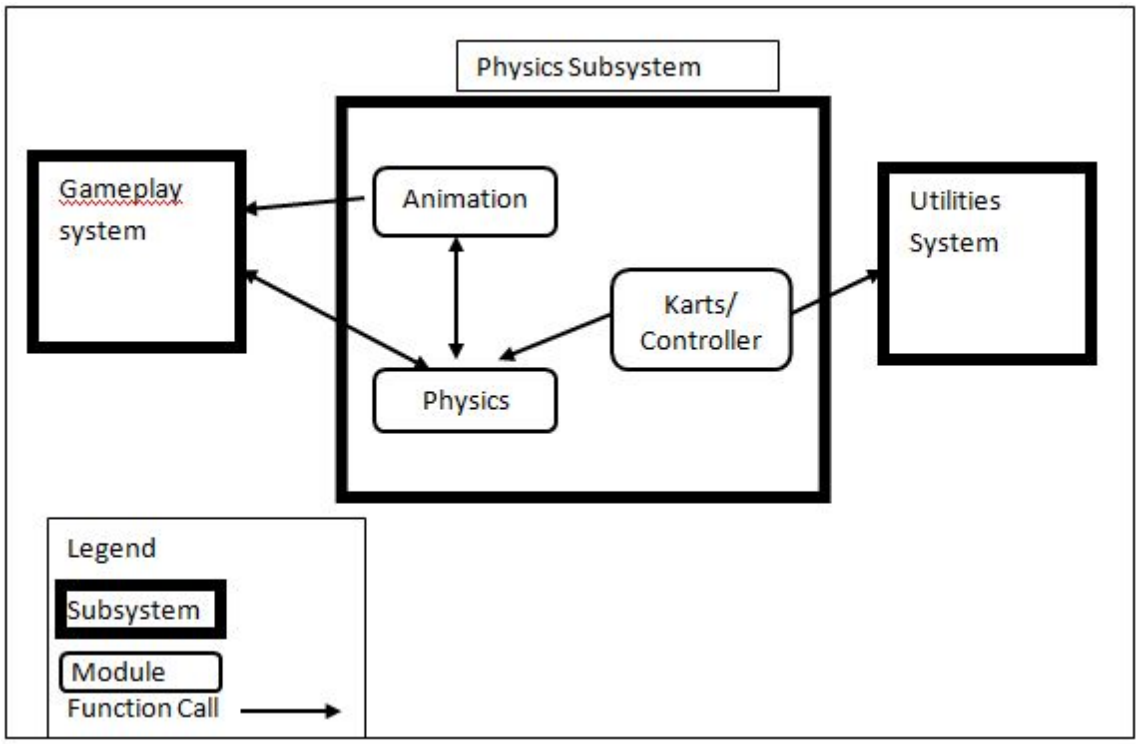
Karts/Controller: The physics subsystem manages turning the input from the controller via the utilities subsystem, and turns control inputs into physical kart behavior, managing thinks like acceleration, position, velocity and inertia to make the kart driving feel as smooth and realistic as possible.

Interactions between Physics Systems and other major subsystems

Gameplay: The physics subsystem provides information to the gameplay subsystem extremely frequently, and also receives it to process. The Gameplay system will call on the physics system for the processing of object collisions, as well as for obtaining the behavior of various in game objects with physical models. Physics manages acceleration, position and velocity of all karts and objects, which the gameplay system need to have access to in order to accurately represent the game state, and also to use in the UI. The physics subsystem will also indirectly pass animations to the Graphics subsystem via the gameplay subsystem.
Utilities: The physics subsystem handles the Kart/Controller module, which takes its input through the input module in the utilities subsystem. It convert controller input into actual kart behavior, which is passed on to the gameplay system, which holds a representation of the entire game state at all times.

Interactions within Physics subsystems

The physics module interacts with both other modules in the system. It take input from karts/controller which it uses in modeling the physics of the game and object interaction as well as all the vectors associated with each object.
It also interacts with animation for the animation of physical object interactions (3d animations of object collisions) which the physics system handles, and then indirectly passes to graphics, as its elements are more intertwined with physics than with the other elements in graphics.



Utilities Subsystem

Utilities is another of the subsystems that we derived from our architecture of SuperTuxKart. This subsystem is made up of small non essential modules that provide additions to the game and its subsystems. The modules within it include Input and audio. The input module manages the input taken from the user via their choice of controller, either a keyboard or a gamepad. Audio manages what SFX and music need to be played depending on the players place in the game.

Utilities interacts with two other subsystems Gameplay Support, Gameplay. The config modules from Gameplay support will supply what type of controller the user is using so that input will know what type of input to expect from the user. The audio module will interact with Gameplay so that based on the Gameplay state it will play certain SFX and music.

Lessons learned

- Learned to not look to deep into the code /documentary when trying to create a high level architecture
- Learned how general game architecture works for a variety of games
- Learned that it's very important to have an idea of how the game actually appears to the player before trying to figure out its architecture
- Learned that it's a lot easier to explain a high level architecture in diagrams rather than solely in words while presenting.
- Learned it's not always best to divide work up at the beginning before everyone has a general understanding of the system as this

Limitations

The limitations of our architecture are primarily caused by the object oriented layout. Object oriented architectures work well when data is managed synchronously. This is impractical with most games considering audio and graphics must be rendered asynchronously to avoid frame and audio tearing and stuttering. Though this may be solved by certain operating systems, SuperTuxKart relies on hardware and software abstraction layers. The architecture solves this issue by breaking from its intended object oriented layout when managing asynchronous calls.

While deriving the different subsystems for the architecture of SuperTuxKart, we noticed that there were a few modules within gameplay that were not crucial for the success of the game. These modules included config, challenges, add-ons, and online. After endless amounts of discussion, we initially decided that a separate subsystem would be created for these modules and it would be named Meta-Data, even though we knew that this term did not match the subsystems practical use. Following the presentation, it was clear that subsystem

had to be renamed. Following further discussions within the group and with the professor we resolved this issue and renamed Meta-Data to Game Support.

Conclusion

In conclusion Super Tux Kart is a 3d free to play, open source kart racing game. It uses an object oriented architecture to support its requirements. This system is optimally divided into 5 modules: Gameplay, Graphics, Game Support, Physics and Utilities. Our architecture provides a high level of flexibility while keeping modules that interact close together, allowing for easy data flow. This architecture's subsystems will allow for the game to play naturally, easily and efficiently, as well as supporting modification and online play, when they are implemented eventually.