Need For Tux

Super Tux Kart Concrete Architecture

Daniel Lucia - 14dvl@queensu.ca - 10156114
Michael Wilson – 14mw13@queensu.ca - 10152552
Nicholas Radford 13nr34@queensu.ca - 10141299
PJ Murray - 14cm36@queensu.ca - 10160261
Matthew Pollack – 13mbp@queensu.ca - 10109172
Morgan Scott – 14mjfs@queensu.ca - 10149124

TABLE OF CONTENTS

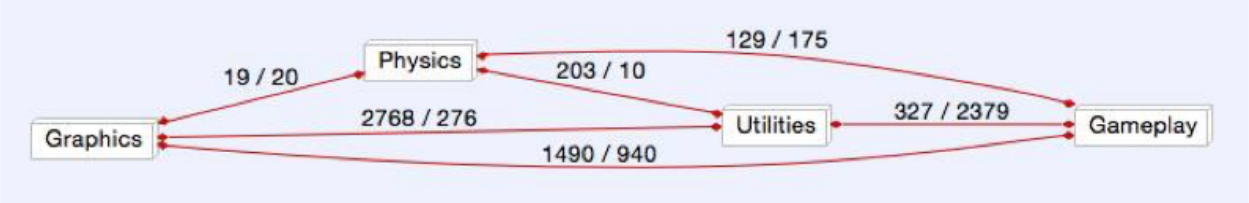
Abstract

SuperTuxKart is a complex piece of open source software developed over many years. In order to fully understand the project, a derivation process explains, in-depth, how the concrete architecture was picked. Each major group of classes are described in detail down to inner and outer dependencies. These findings lead to the conclusion, STK is an Object-Oriented architecture, with four major subsystems: Physics, Gameplay, Utility and Graphics Engine.

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 concrete architecture of SuperTuxKart. The concrete architecture was obtained using the Understand Tool which analyzes the source code. This document will delve in the architecture of a subset of the top-level subsystems of the studied software.
Through our analysis of the source code of SuperTuxKart, we have found that its architecture is of the OO tyle consisting of four major subsystems: Physics, Gameplay, Graphics Systems, and Utilities.
We will first discuss the derivation of the concrete architecture then the differences between the Concrete and Conceptual architecture will be discussed. This will be followed by the two sequence diagrams. Next, the two subsystems that we found impacted the runtime of the game the most will be discussed in detail, these were Gameplay and Graphics. There will be diagrams explaining each module and its interactions with other modules as well as other subsystems. Lastly we will reflect and discuss the limitations we faced while creating the concrete architecture for SuperTuxKart and then the lessons learned will be reviewed.

Derivation Process



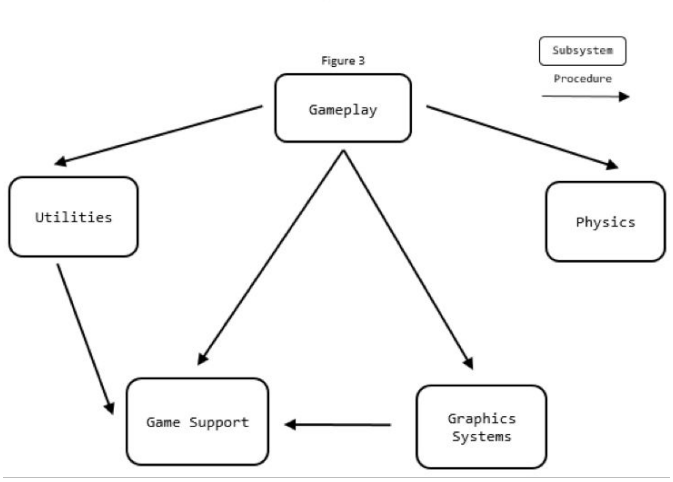
To come up with our concreate architecture we started with our conceptual architecture designed in assignment one. We started by putting our conceptual architecture into the understand software to have a look at the internal dependencies of our subsystems. After looking at the dependencies it was very obvious that our game support subsystem had next to no internal dependendencies. We decided to remove this subsystem as a result of our findings. We then had the problem of a few modules floating and we needed to figure out what subsystem we wanted to put them in. To figure out their placement we looked at their dependencies as well as delved into their code to make sure our placements of of the modules still fit on a fundamental level with our design. After addressing the removal of the game support system we started poring through code of various classes to see how we could reduce the coupling between major subsystems and increase the internal cohesion. We also throughout the process reevaluated our type of architecture but decided to stay with object- oriented

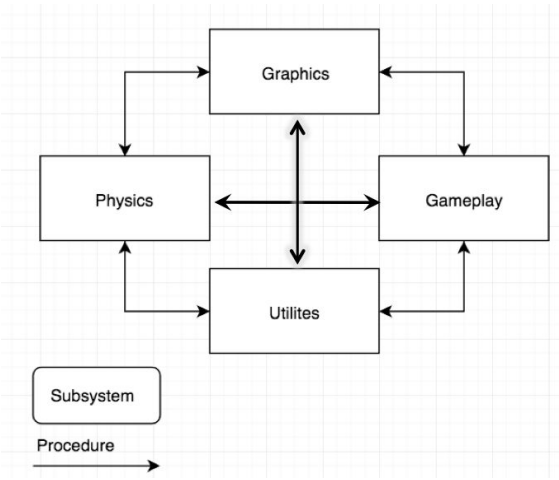
Differences between conceptual and concrete architecture

The major difference between our conceptual and concrete architecture is the missing game support system in the concrete architecture. We felt after looking at the code and “understand” that it was really not needed and decided to remove it. The other major difference was the number of dependencies. In our conceptual architecture we felt there were only a few logical dependencies between subsystems. We realized this was not true after looking at the code as there's a ton of unexpected dependencies all over

the place resulting in every major subsystem talk to every other major subsystem in our concrete architecture.

Figure 1

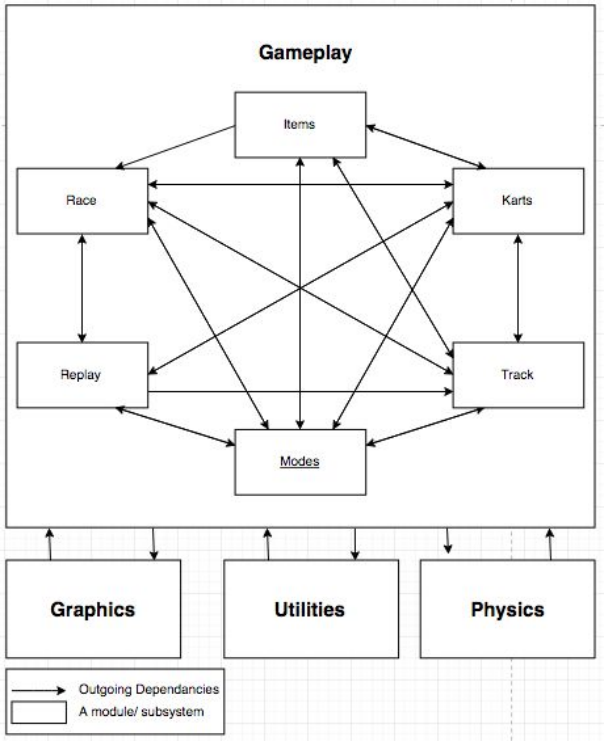




Gameplay Subsystem

Gameplay Description: 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.

Gameplay Subsystem Diagram



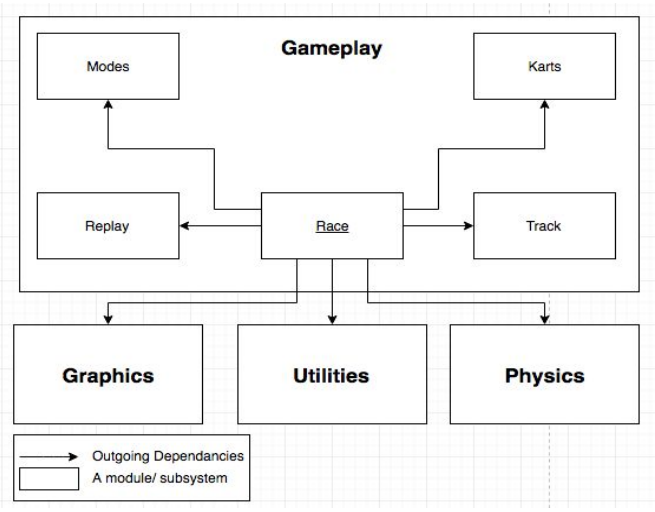
Race Module

Race subsystem of gameplay

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.

Race Subsystem placement:

In our first assignment we decided to put race module in the gameplay subsystem. After an extensive look at the code as well as the underlying dependencies between modules we still feel this was the right course of action due to the high dependency of the race module to the other modules located in gameplay subsystem.



Race module Diagram

The Diagram is a representation of all the outgoing dependencies from the race module to respective modules and subsystems.

Race subsystems relations to other subsystems:

Gameplay/race - Gameplay/karts: The race module needs to talk to karts so that it can record the race data as well as high score information for the grand prix/ individual races.
Gameplay/race - Gameplay/modes: The gameplay race module talks to the modes module to figure what game types to load into the grand prix as well as the available modes it can generate the grand prix from.
Gameplay/race - Gameplay/replay: The race module needs to pass its data to replay module so that replay can create a copy to be replayed at any point in time.
Gameplay/race - Gameplay/tracks: The race module needs to check if tracks are playable as well as holde the order of the tracks to be played in the grand prix.
Gameplay/race - Graphics: The race module needs to be able to pass its information out to graphics so it can be rendered.
Gameplay/race - Utilities: It needs to talk to the scripting engine so that the race module code can be executed.

Gameplay/race - physics: the race module talks to the physics engine very little and it's mainly there to make sure that race is holding the most accurate data about the polling positions in the race.

How the relationships changed between assignments:

Overall the race module dependencies are the same as we predicted in assignment with the major difference being the fact that it communicates with the replay model. Originally we didn't know of the existence of the replay module hence it did not fall into our scope of assignment 1. We also thought race talked to the items module but according to the code it is incorrect and was an oversight on our part in assignment 1.

Modes Module

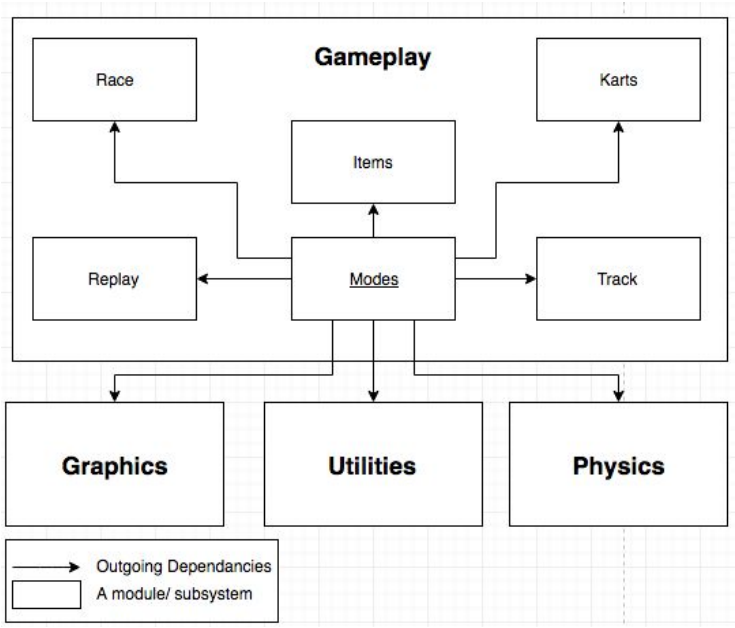
Modes subsystems relations to other subsystems:

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.

Modes subsystem placement:

Modes was originally in our gameplay subsystem and we've decided to leave it there. After analysing the code our assumptions made off of the documentation in assignment 1 held true meaning modes has a lot of dependencies on the internals of the gameplay subsystem as such we've decided not to move it.

Modes module diagram



The Diagram is a representation of all the outgoing dependencies from the modes module to respective modules and subsystems.

Modes subsystems relations to other subsystems:

Gameplay/Modes - Gameplay/items: The modes module talks to the items module to load specific items for the game type.

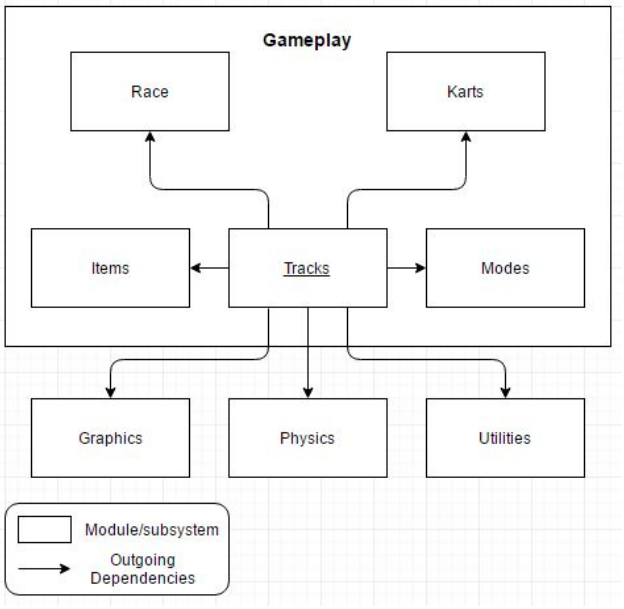
Gameplay/Modes - Gameplay/karts: The modes module tells the game what karts can be loaded for the specific game type.
Gameplay/Modes - Gameplay/race: The modes needed to be able to end the race and pass the subsequent data to the race module to hold the winner. Its also used by the tutorial world.
Gameplay/Modes - Gameplay/replay: The modes module needs to pass the replay module its data so replay can hold a copy to reply.
Gameplay/Modes - Gameplay/tracks: The modes module needs to know the track so it can select appropriate game modes as well as items and objects to populate it.
Gameplay/Modes - Utilities: The modes module need to call utilities to execute the script with the script engine.
Gameplay/Modes - Physics: The modes module holds a class that is used to render 3d cutscene that is partially handles by the physics engine.
Gameplay/Modes - Graphics: The modes module holds a class that it uses to render 3d cutscenes and this is also passed to graphics to aid in the actual rendering of the cit scene.

How the relationships changed between assignments:
The Modes module behaves exactly as predicted in assignment 1.

Tracks Module
The Tracks module is found in Gameplay and handles all the track information, track objects, drivelines, and checklines. This includes different information pertaining to the separate track objects, laps taken, and the terrain information.

Tracks Module Placement
In assignment 1 the Tracks module was placed in the Gameplay system. After analyzing the code with the help of the Understand Tool we decided that there was no reason to change the location of Tracks. Tracks is so dependent on the other modules that are located in the Gameplay subsystem that it would make no sense to have it anywhere else.

Tracks Module Diagram



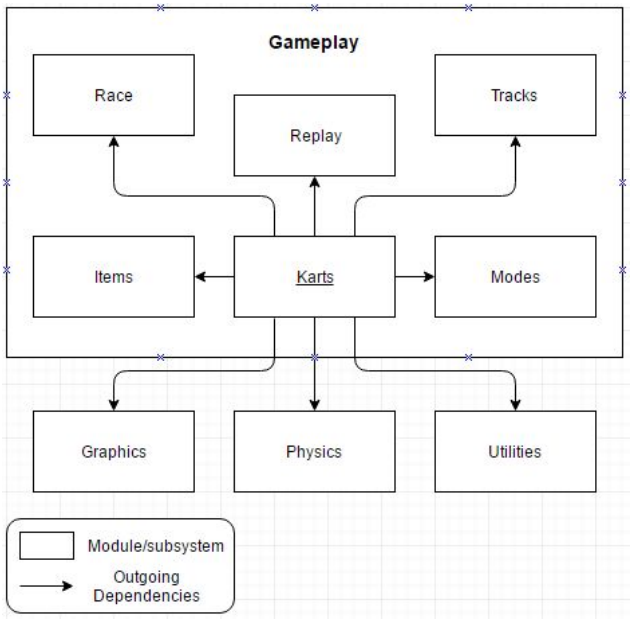
Tracks Interactions
The Tracks module depends on every other module in the Gameplay subsystem except for the replay module. It is also dependent on Graphics, Utilities and Physics.

Gameplay/tracks - Gameplay/karts: The Tracks module must send information about the terrain that the kart will be interacting with so that the karts module can properly respond.
Gameplay/tracks - Gameplay/modes: Tracks sends modes the track information so that it can prompt the user with the correct information in regards to what game mode can be selected.
Gameplay/tracks - Gameplay/items: The items module needs to know what items are going to be needed during the race, therefore tracks sends items the track information which includes the location of the items as well as the items available for that track.
Gameplay/tracks - Gameplay/race: Gives the Race module what tracks are available for the user to select.
Gameplay/tracks - physics: Tracks sends information about actions that occur on the track so that the physics engine can make calculations, determine the outcome, and produce the corresponding animation
Gameplay/tracks - graphics: Tracks sends the tracks information to the graphics subsystem so that the corresponding graphics can be outputted to the user. Requires the config files held within graphics so that the track is prepared with the correct player information and profiles which is then rendered using the guiengine.
Gameplay/tracks - utilities: CAlls the script engine to run the needed scripts for the tracks module.

Karts Module
The Karts module is found in the Gameplay subsystem and is responsible for holding all the kart related class files which includes the Karts properties.

Karts Module Placement
This module was originally placed in the Gameplay subsystem. Our group found there was no reason to move it due to its high dependencies to the other modules we have placed in the Gameplay subsystem.

Karts Module Diagram



Karts Interactions

The Karts module depends on all other modules found in Gameplay. This module also depends on all the other subsystems including Graphics, Physics, and Utilities.

Gameplay/Karts - Gameplay/tracks: The karts module requests upcoming terrain info so that the Kart can react properly to the environment.

Gameplay/Karts - Gameplay/modes: Karts requests the game mode and with this information it can send the modes module what karts will be available for the user to choose from for that specific mode.

Gameplay/Karts - Gameplay/items: Karts depends on items in when a Kart interacts with an item to determine what action will occur.

Gameplay/Karts - Gameplay/race: The karts module depends on the race module in order to figure out the amount of karts participating in the race along with the properties that will be used for each kart during the race.

Gameplay/Karts - Gameplay/replay: Karts needs to send the Karts properties along with the physics of the karts so replay can fulfill its job.

Gameplay/Karts - physics: Karts relies on physics because the karts module needs to know what the outcome of different actions on the kart will be during the race.

Gameplay/Karts - graphics: Karts depends on graphics because the karts module needs to know the properties of each kart in the race, therefore it accesses the config file for this information. Karts also relies on graphics to render these visuals in game.

Gameplay/Karts - utilities: Karts depends on utilities for the users input so that the kart will move accordingly.

Gameplay/Items

The items module handles all of the collectables and weapons that are within SuperTuxKart, this modules is dependent on many modules within gameplay such as karts, tracks, and modes as well as the Graphics Physics and Utilities subsystems.

How the relationship has changed from our conceptual architecture

- This module has not moved outside of gameplay from our first iteration of our architecture

Items' interactions with other modules and subsystems

Gameplay/items -> Gameplay/karts : items talks to karts to know how certain items interacts with certain karts

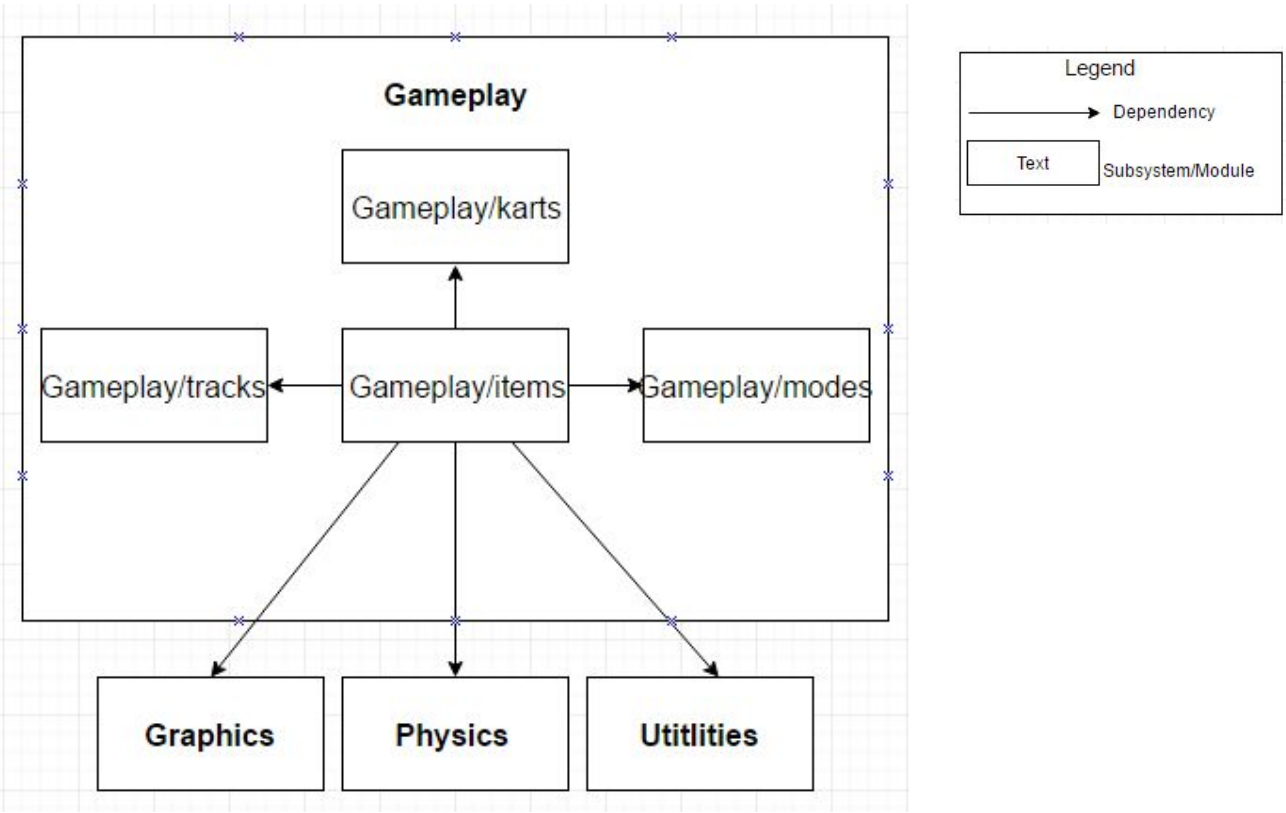
Gameplay/items -> Gameplay/tracks : items talks to tracks to know what certain items are available on certain tracks

Gameplay/items -> Gameplay/modes : items talks to modes to know what certain items available during certain game modes

Gameplay/items -> Graphics : items talks to graphics to show the items on screen when used

Gameplay/items -> Physics : items talks to physics to make calculations for how using and item can affect players.

Gameplay/items -> Utilities : items talks to utilities to get a few necessary libraries for items use

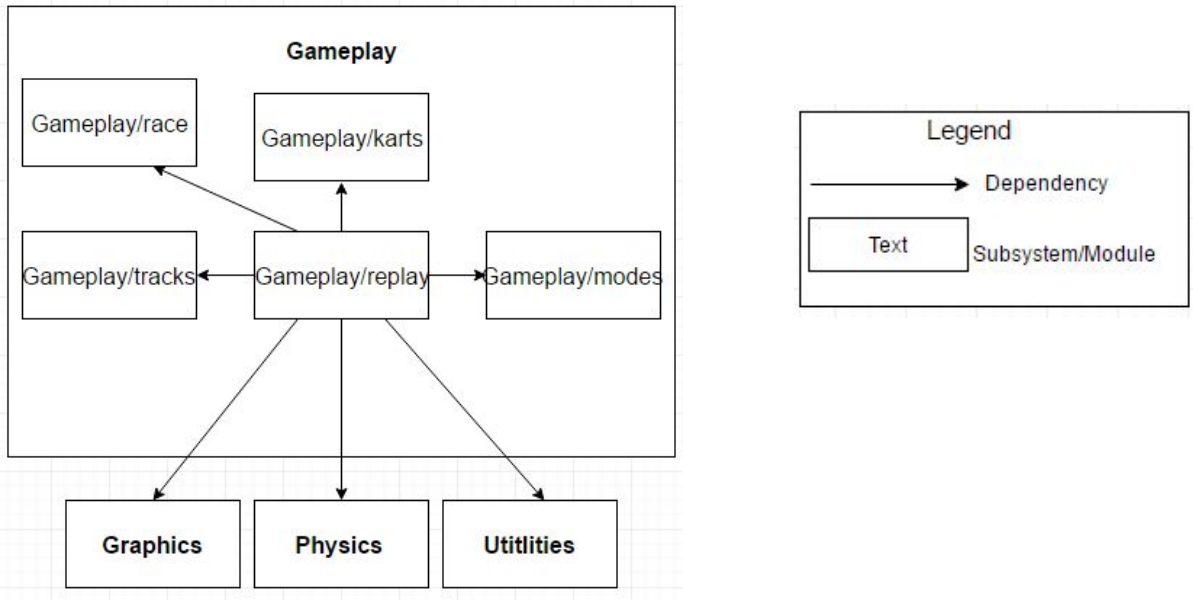


Gameplay/Replay

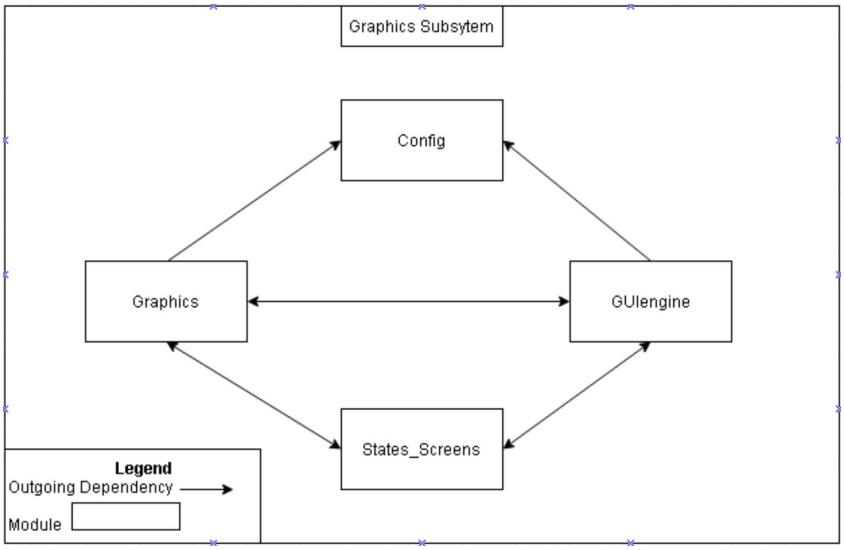
The replay module manages the user's ability to watch recorded races while also assisting with the ghost race mode. This module has a high level of dependency on many modules within gameplay and was therefore placed there. It relies on karts, tracks, race, and modes as well as the Graphics Physics and Utilities subsystems.

Replay's interactions with other modules and subsystems

Gameplay/replay -> Utilities : replay talks to utilities to get libraries needed to play a replay of a recorded race
Gameplay/replay -> Graphics : replay talks to graphics to play a replay of a recorded race
Gameplay/replay -> Physics : replay talks to physics to handle collisions and such in a replay
Gameplay/replay -> Gameplay/modes : replay talks to modes to play a replay of a recorded race done with a certain mode
Gameplay/replay -> Gameplay/race : replay talks to race to loaded saved race data for the replay
Gameplay/replay -> Gameplay/karts : replay talks to karts to play a replay of a recorded race with the karts that were used
Gameplay/replay -> Gameplay/tracks : replay talks to tracks to play a replay of a recorded race on the saved race track



Graphics Engine Subsystem



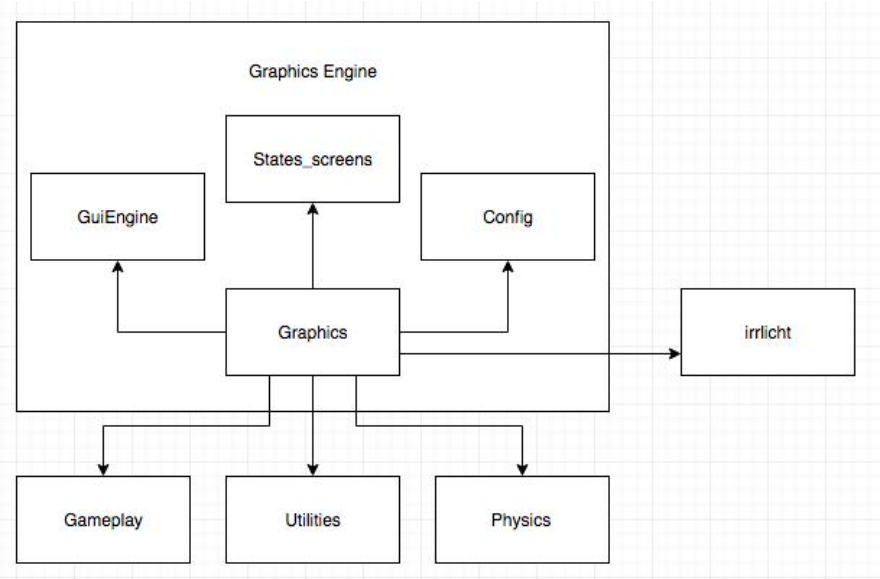
Graphics Module:

The Graphics Engine subsystem manages all high level graphics functions, but how are those functions mapped on a low level? The Graphics module under Graphics Engine handles this. SuperTuxKart uses a third party graphics engine named Irrlicht. Irrlicht was designed for easy, and lightning fast cross platform OpenGL contexts. The Graphics module acts as the middleman between STK and the low level graphics operations it ultimately relies on. Due to this interaction, the Graphics module provides irreplaceable functionality. Due to its importance, most of all STK interactions begin from Graphics.

The Graphics module has three neighbors: Config, States_screens and GuiEngine. Graphics depends on all of it's neighbors. Config holds important screen preference information. Graphics requires retrieving and storing this information in multiple instances, most notably when the OpenGL context is

first created. States_screens and GuiEngine are required when a screen resolution change is made. Both modules require unloading before then loading after. The process must be done due to Irrlicht' design. A confirmation modal is presented after the change. All these interactions can be found in the irr_driver file.

The Graphics module also depends on each other subsystem, Gameplay, Utilities, and Physics. Gameplay supplies items, karts and other race data. Utilities supplies important debugging and profiling functionality. Physics supplies the DebugDrawer which displays important physics debugging information on the screen. Gameplay requires unloading before and loading after a screen resolution change.



GUIengine

The GUIEngine module contains an AbstractStateManager class that you must override to use the GUIengine. This class uses a stack data structure to keep track of all the menus. It contains classic push and pop operations a stack would have. This would also contains ways to reset the entire stack or replace the top most stack. This class also has a enterGameState() method that will make the StateManager enter game mode. The AbstractTopLevelContainer class contains methods for getting and adding widgets. All this is done through a vector of widgets.

This is an abstract base class for both screen and Modal Dialog as well. The EventHandler class in the GUIEngine handles all the irrLicht, and GUI events. In game mode, all the input events are not handled by this class but are instead handled by the input module. In menu mode, however, this class will map all the input to game actions. This is done with assistance from the input module. After this all done the EventHandler class can now trigger and focus events. This class is effectively a wrapper for the irrLicht graphics engine.
ModalDialog in the GUIengine module is an abstract base class that essentially handles all the creation and events of the various modals used. A modal is a lot like a pop up or alert menu. Some examples of modals in Super Tux Kart, are the race pause dialog, vote dialog, and select challenge dialog. By overriding methods in this class, you can customize and create different modal dialogs. This class also allows for creation of modals through an XML file.

The GUIengine/Widgets module is responsible for handling all of the various widgets used in the GUI. It has a base class that is nearly abstract and several sub widgets that inherit from it. Once it assembles a widget it passes it to a SkinWidgetContainer to avoid having to calculate rendering every frame.

How it has changed from conceptual architecture
Widgets interacts with the Config to get necessary information, and config has been moved into the graphics subsystem from the Gameplay Support subsystem, which was eliminated. This means it does not have to interact with other subsystems as much.
Interactions with other graphics subsystems
States_Screens Widgets does not interact directly with States_Screens, but both are taken as input for GUIengine::Screen
Config Widgets gets base info on availability and graphics settings from config
GuiEngine Widgets provides it's rendered widgets to GUIEnging for use in screens
Graphics Widgets does not directly interact with Graphics

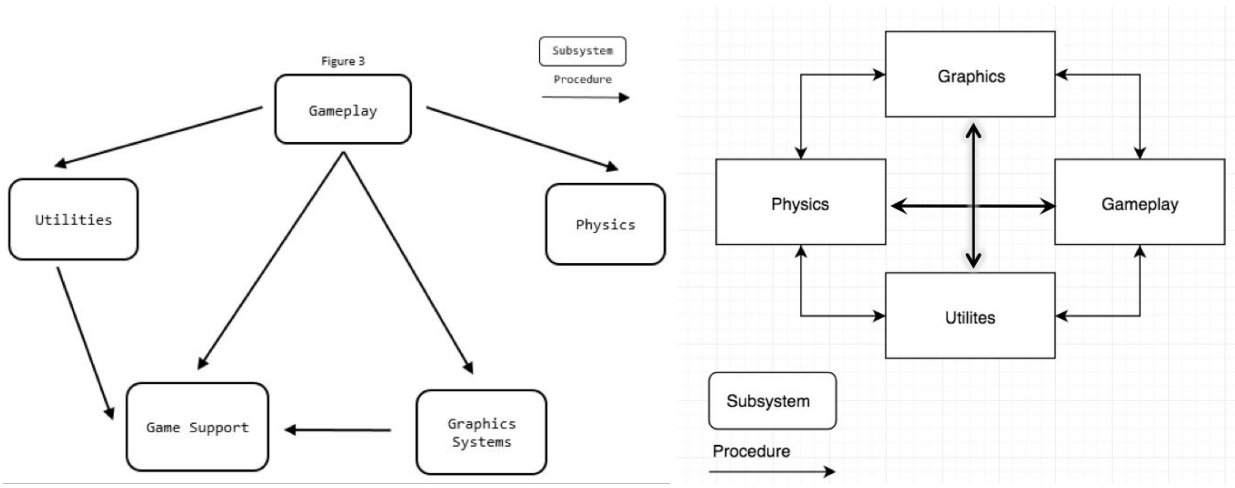
Config

The Config module includes all of the systems that provide information about settings. It manages graphics settings and similar, as well as player information, account preferences, progress and scores.
How it has changed from conceptual architecture
Config used to be in Gameplay Support, but when that was eliminated it was added to the graphics subsystem. Since most of the call to it came from there, this eliminated many inter-system calls and greatly increased efficiency.
Interactions with other graphics subsystems
States_Screens Config does not directly interact with States_Screens
GUIengine Config provides things like settings such as resolution and quality to GUIengine
Graphics Config provides various graphics settings and information to Graphics

Reflection

Our conceptual architecture was very similar to our concrete architecture. We found that there was no need for the Gameplay Support system, which had initially held all of our modules that did not directly affect the gameplay but that others relied upon. We found that we could greatly condense interactions between subsystems if we simply moved each support system in with the system it was supporting. After this was done, we found a few modules that interacted largely with different systems than expected. We were able to eliminate hundreds of inter-system calls by moving certain modules, such as the scripting engine, between subsystems. We also discovered one large interaction that we had missed in our conceptual architecture, but overall, our guesses were reasonably accurate based solely on the documentation.

Figure 1



As shown in the diagrams, we eliminated Game Support. Physics as a system interacted with many more modules than initially thought, as it contained information critical to Graphics and required information from utilities for kart control. It was a mistake in our conceptual architecture to assume based on documentation that most of the interactions would be perfectly one directional, as any subsystem that interacts with another will get a result from that, and that interaction will go the other way. Graphics interacts with utilities when we previously thought it didn't because many of the minor modules in utilities are essential for graphics, it contains many of the essential files.

Limitations

An Object-Oriented approach to the concrete architecture provides the best description of STK, but carries with it some limitations. Firstly it is extremely coupled. It's rare to see a module/system which isn't coupled to each of its siblings and parents. And even if it isn't, it's likely that it inherits from a class that is. The consequences of this include: that a new module needs understanding of each system, as it'll likely end up depending on each other system, and it's programmatically simple to place the a new module wherever.

Lessons Learned

We learned during the derivations process that Understand was not well suited for collaboration, because when transferring the udb files some of the mappings were lost. Moreover, some of our group members were not able to install the Understand on their machines, because of this we had to resort to sending multiple screenshots to these team members. We later learned, that Understand allows for exporting XML files that contain all the mappings. This made is possible and easier for the team member that got Understand to run to import the mappings and view the architecture. Moreover, we also learned that the Super Tux Kart class descriptions aren't detailed. We discovered that often more detail can be gathered from the class by looking at the documentation for the constructors of the class.

Conclusion

In conclusion after the use of the understand program we determined our concrete architecture of SuperTuxKart to be object oriented. This system is optimally divided into 4 subsystems now with game support subsystem being completely eliminated and its contents moved to other subsystems. Our new architecture has the subsystems: Gameplay, Physics, Graphics, 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.