# Concrete Architecture

Super Mario ... Oops I mean Tux!

Gabriele Cimolino, Jack East, Meryl Gamboa, Tyler Searl,
Matt Skoulikas, Stefan Urosevic

2017-11-13

# Contents

**Abstract** **3**

**Derivation Process** **3**
Grouping Sources in Understand . . . 3
Changes to Conceptual Architecture . . . 4
Considered Alternatives . . . 6

**Concrete Architecture** **7**
Subsystems . . . 8
Gameplay Systems . . . 8
Physics . . . 8
Engine Functions . . . 8
Core Systems . . . 9
Design Patterns . . . 9
Abstract Factory . . . 9
Template . . . 10
Sequence Diagrams . . . 10
Sequence Diagram: Loading a Level . . . 10
Sequence Diagram: Fireball Hit . . . 11
Subsystem Investigation . . . 11

**Reflexion Analysis** **13**
Divergences . . . 13
Comparison of Conceptual and Concrete Architectures . . . 13

**Lessons Learned** **14**

**Dictionary** **15**

# List of Figures

1 Understand Dependency Graph Using Original Conceptual Architecture . . . 4
2 New Conceptual Architecture Diagram . . . 6
3 Concrete Architecture Diagram . . . 7
4 Sequence Diagram: Loading a Level . . . 10
5 Sequence Diagram: Fireball Hit . . . 11
6 Understand Dependency Graph of the Gameplay Systems Subsystem . . . 12
7 Understand Dependency Graph Using Original Conceptual Architecture . . . 13

# Abstract

This report summarizes our exploration of the SuperTux source files in order to map these files to functionally distinct subsystems. Our point of entry to this task was the conceptual architecture which we had made in the first assignment. This conceptual architecture was revised to better suit how the game was actually implemented. This report describes the changes from our conceptual architecture as well as the steps along the way. The interactions between the concrete architectures subsystems are explored through an investigation of the design patterns employed by SuperTux, The logical path of a command was traced by following the original function call through the code base to its origin and then visualized with a sequence diagram. This report dives further into the inner workings and dependencies of a single subsystem of the full architecture. A thorough examination of game systems was done in order to shed some light on how all the components connect on a more specific level. Once the concrete architecture was decided upon, a reflexion analysis was performed to explain the divergences found between our conceptual and concrete architectures. At the end of the report is general reflections of the process itself. The final few points delve into how the derivation process went, and what issues arose when examining the SuperTux architecture.


# Derivation Process

## Grouping Sources in Understand

Our initial approach to creating a concrete architecture for SuperTux was to have Understand analyze the source files and then to assign these source files to the different subsystems we had laid out in our conceptual architecture. For some subsystems this process was straightforward; in fact, some subsystems in our conceptual architecture shared names with the folders found in the SuperTux source files. This was the case for the Audio and Editor subsystems; from these folders we were able to create complete and cohesive architectural components with minimal manipulation and fine tuning. However, other subsystems were not so simply identified in the source and often required multiple folders or files spread around the source code to be rounded up and included in the categories we created. The Core Systems component was one such subsystem that included multiple folders with largely unrelated functionality as well as certain files from folders that later became part of the Game Logic component. Physics experienced similar treatment and, since it is so closely related in functionality to Game Logic, it contains only the trigger system and the files responsible for collisions between GameObjects. This means that the functionality found in Physics is drawn exclusively from folders that were later included in Game Logic. As a consequence of this division, Physics is now entirely different than the Physics subsystem that was predicted in the original conceptual architecture.

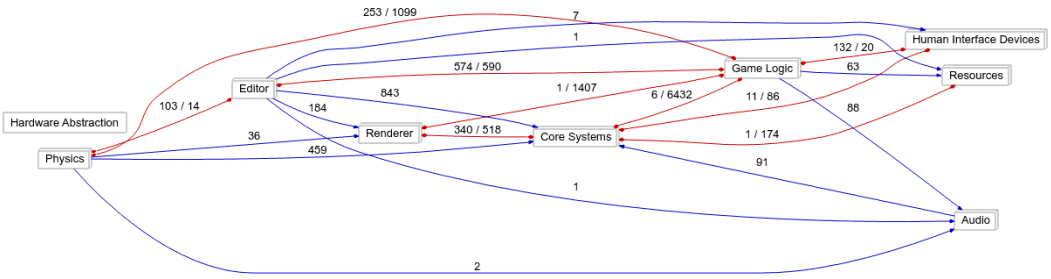


Figure 1: Understand dependency graph using original conceptual architecture

## Changes to Conceptual Architecture

The divergences between our expectations and the reality of the system were not limited to the Physics subsystem and, in fact, occurred at every layer of our conceptual architecture. The most disruptive of these were the absences of systems corresponding to the functionality of the Platform Independence Layer, Resources, and Events subsystems. The functionality that we expected each of these components to contain was not absent from the system but was instead integrated into other subsystems in ways that made separating the source code into the original subsystems illogical. For instance, our conceptual architecture featured a dependency cycle between the Game Logic, Physics, and Events components that made use of Physics system to generate a list of collisions for Events to handle and pass on to Game Logic in order to resolve. This sequence of events also occurs in the actual implementation of SuperTux, however it happens in the form of a codependency between the sources found in the Game Logic and Physics subsystems whereby Game Logic sends information about the collision geometry of specific entities to Physics and Physics determines if a collision has occurred. If Physics finds that a collision has occurred then the Game Logic component is fully capable of resolving the collision without any further reliance on other subsystems. In this case, the details of the implemented structure of the systems functionality is so significantly different from the architecture that we had imagined that changes needed to be made to our conceptual system architecture.

These changes meant that the functionality from the systems in the conceptual architecture which were absent in SuperTuxs concrete implementation were migrated into the systems that either required of contained that functionality. In the case of the events system this just meant that collision detection was handled directly by Game Logic and Physics. The absence of a system that handles the collection and availability of the necessary game assets from the file system meant that the functionality of the Resources component from our conceptual architecture was handled in some other way. Upon further investigation of the source files associated with the subsystems that we thought would require the services of Resources, we determined that each of these systems handles its own resource management in a way that is suitable for their respective functionality. Due to our findings, we believed that it was best to remove the Resources subsystem from our revised conceptual architecture since its responsibilities were being performed by other parts of the architecture. There was, however, one aspect of the functionality of Resources that was not suitably covered by other subsystems but corresponded directly to a folder found in the SuperTux sources; this was the managing of the games addon content. Because SuperTuxs addon content is distinct from the content

included in the game by default, since it is acquired differently and the game therefore cannot be certain that the content is still available to the game, it needs to be handled completely differently. These requirements led us to the creation of a new conceptual subsystem, which we called Addons, that deals with managing this extra content and has many of the same dependencies as the other subsystems that deal with system resource management.

This divergence between how we had envisioned resource management and the reality of the systems approach to the problem meant that the functionality found in other aspects of our conceptual architecture could be handled in the sources in ways that were not possible before. This is what we found when we began inspecting the source files related to the system functions that we expected to require an abstracted interface to deal with the computers hardware, as well as even earlier when we began grouping folders in Understand and found no systems or individual files that we could assign unequivocally to the Platform Independence Layer present in our original conceptual architecture. Again, as in the case of the Resources component, it was not as though SuperTux did not do any hardware abstraction, only that the functionality that we expected this system to provide was adequately handled by its related subsystems. We found that SuperTux relies largely on three systems that perform these responsibilities: OpenGL, OpenAL, and PhysFS. The last of these is a file system utility and its functionality, although it does provide platform independence, is far closer to the file system service that we expected to find in Core Systems, leading us to group the PhysFS related sources into that subsystem instead. OpenGL and OpenAL are video and audio utilities respectively and the primary drivers of the systems A/V output. Because SuperTux uses only these systems for rendering the game and playing audio, no further abstraction is required and so they are referenced explicitly in the source files related to the Renderer and Audio subsystems in our conceptual architecture. The developers decision to have video and audio systems reference the OpenGL and OpenAL APIs directly made it impossible to isolate this functionality in each system and so we removed it from our conceptual architecture.

At this point we had fully grouped the source files into their appropriate subsystems and were able to further group these subsystems into higher level systems by noticing patterns in their dependencies. In this way, the components related to gameplay modes, the Game Logic and Editor subsystems, were placed together to form the Gameplay Systems component. As well, the Renderer, Audio, Addons, and Human Interface Devices subsystems were grouped together, since they are all related to resource management and system IO and therefore had similar dependencies. Because none of these systems depend on the others in their group, we believed it was obvious to form the Engine Functions subsystem as a layer which only had interactions with systems outside of itself. The Core Systems and Physics subsystems were left alone since their functionality is distinct within the system and encompasses several related but different types of responsibilities. Our next task was to determine which dependencies were necessary to the typical operations of the game and which were unnecessary and possibly erroneous. We identified the edges that we considered to be required by cross referencing some combination of our understanding of how the game works, the dependency graph generated by Understand and the number of references to each other system that it showed, and our original conceptual architecture. Ultimately, this led us to an architecture that was in many ways similar to our original conceptual architecture but different in ways that addressed the subsystems that we had identified as absent from the concrete architecture. As well, by this point we had reconsidered the architectural style of the overall system and now considered it to be entirely object oriented but with layered characteristics, since it groups and segregates similar functionality that most often only requires access to the layers above and below.

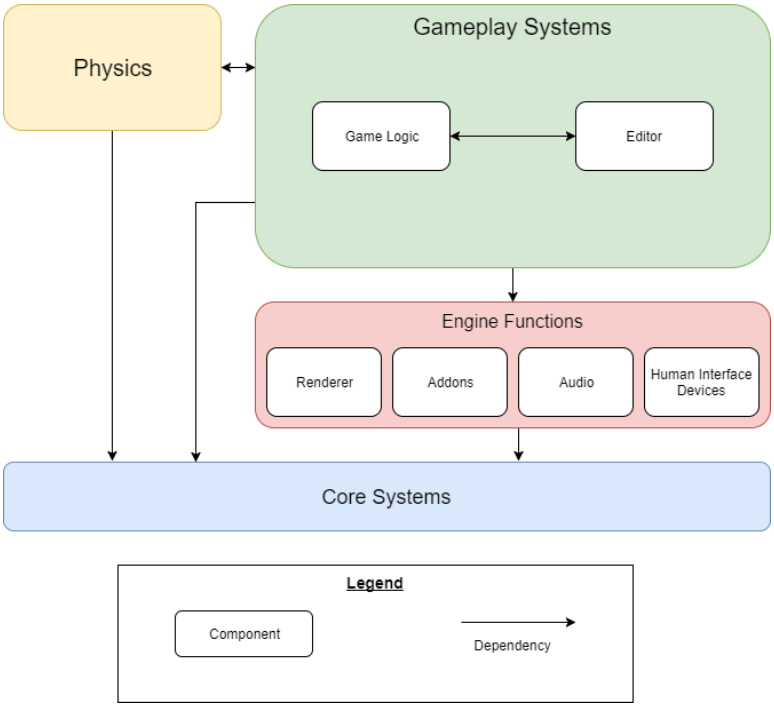


Figure 2: Our new conceptual architecture with the subsystems from our original conceptual architecture appropriately grouped

## Considered Alternatives

The absence of the dependency cycle formed by the Game Logic, Physics, and Events subsystems from our original conceptual architecture in our new conceptual architecture made moving Physics into Gameplay Systems as a subcomponent an appealing possibility. Since both Game Logic and Editor are codependent with Physics, in essence, this is what is occurring while the game is running. However, we decided against this option for two reasons. The first reason was that, from the perspective of Gameplay Systems, the newly formed subsystem would be less cohesive if it contained two subcomponents that controlled separate game modes, and relied heavily on each other during play, and one that was only a utility to be used by both. In this scenario the physics subcomponent would lose its significance and would be more cohesive as a subcomponent of either Game Logic or Editor. The second reason why Physics was not merged into Gameplay Systems is because of the divergence created by how we believe the Physics subcomponent should behave and the way in which it was implemented. In our concrete architecture, Physics depends on Engine Functions because the trigger system, which detects if Tux is in close proximity to GameObjects such as doors and buttons, creates a prompt about these events for the user by calling Renderer and Audio directly. We do not believe that these prompts should be implemented differently than similar systems and that a better solution would require only the dependency from Physics to Gameplay systems, using either new GameObjects for the prompts or the menu system. If Physics were to become part of Gameplay Systems the meaningful absence of this dependency in our conceptual architecture would be obscured by Gameplay Systems reliance on Engine Functions.

The addition of a hardware abstraction component, as featured in our original conceptual architecture, was also considered; however, due to how hardware abstraction is handled by each subsystem, this new subsystem would need to be created from a subset of the source files which were already mapped to Engine Functions and Core Systems because they are integral to the operation of each system. A new Hardware Abstraction subsystem would therefore significantly increase coupling in the concrete architecture since these sources are referenced directly by all of the subsystems that depend on the two from which they would be drawn. For this reason it made more sense to not begin picking apart highly cohesive parts of the architecture to force the inclusion of something that did not exist in that form.

## Concrete Architecture

The only architectural style featured in our concrete architecture is one which did not make an appearance in our original conceptual architecture: the object oriented style. Although this change in our understanding of the system had profound effects on the way that we understood the interactions between subsystems it did not require any modification to the dependencies between them. For this reason, many of the subsystems of our concrete architecture have dependencies similar to those that they had in our original conceptual architecture; however, the way in which the dependencies are used means that the games typical operation is almost entirely different.

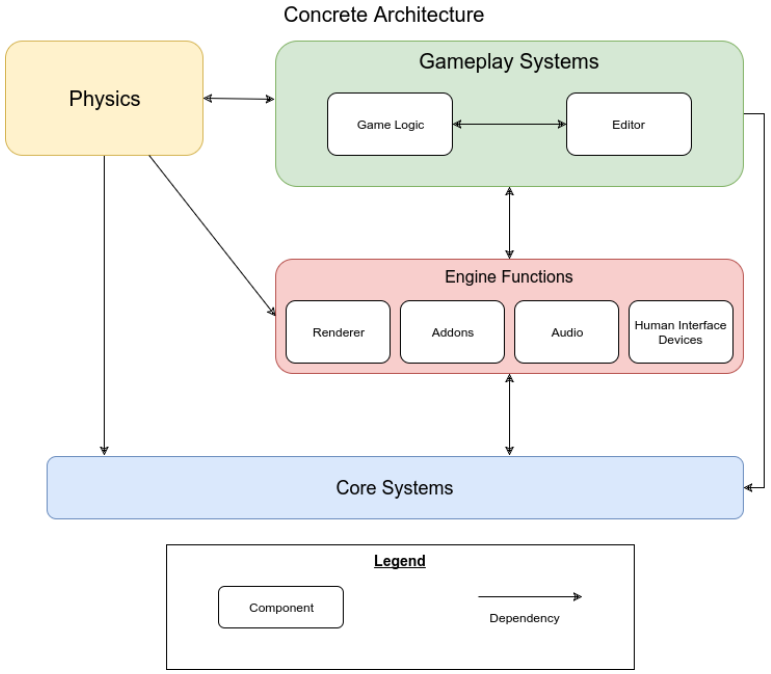


Figure 3: Our finalized concrete architecture which takes into account the divergences between our two previous conceptual architectures

Because SuperTux makes such consistent use of the object oriented style, most of the interactions between the architectures subsystems can be explained as either storing an object which

controls a subsystem or storing and passing along an object which another subsystem can act upon. An example of two subsystems which have similar responsibilities but use one of these two different methods to achieve its purpose is the case of the Renderer and Audio subsystems. The typical operation of the Audio subsystem works by having the current game mode subsystem, either Game Logic or Editor, store a reference to an object called a SoundManager. These systems control the entirety of the audio subsystem by sending instructions to this singleton object so that it can control the rest of the audio system, using its much more intimate understanding of its implementation details. In contrast, the Renderer system typically works by having the subsystem associated with the current game mode build a DrawingContext object out of all of the entities in the scene. This object then identifies the object which is currently responsible for drawing to the screen and has it operate on the drawing requests store in the DrawingContext object. Although in both cases an instance of an object is used to interact with a separate subsystem, the use of the object oriented architectural style allows for flexibility in the interactions between these subsystems such that very different approaches, such as these two, can be used for similar problems. In similar ways for each dependency, the implementation details of each portion of the games functionality are abstracted and intentionally obscured by sharing these sorts of objects between components.

## Subsystems

### Gameplay Systems

The Gameplay Systems subsystem includes only two subcomponents, Game Logic and Editor, which are codependent. From the user's perspective these two systems might seem very different but from the perspective of the system architecture they have all of the same dependencies and can be considered to be functionally quite similar.

During typical play, the Game Logic subsystem is the game's control centre. It requires Physics for collision detection, Engine Functions for input and output from and to the player, and Core Systems for access to system resources as well as other runtime system operations. Game Logic manages all of the entities in the current scene as well as the menu systems. Within Gameplay Systems, Game Logic is codependent on the Editor because it requires functionality related to manipulating and getting information about GameObjects.

The Editor subsystem has all of the same dependencies as Game Logic. In addition, it depends on Game Logic for access to the GameObject files for all of the entities that can be created in the editor.

### Physics

Physics is responsible for determining if a collision has occurred as well as creating prompts for the user whenever Tux enters a trigger area. In order to achieve these functions it depends on Gameplay Systems for information about the GameObjects that might collide, Engine Functions to report trigger events to the player, and Core Systems for access to the math libraries required to determine if a collision has occurred.

### Engine Functions

The Engine Functions subsystems contains four independent subcomponents with similar responsibilities and dependencies but no dependencies between them. They were grouped into this larger system since they are used in similar ways by the subsystems around them. The

subcomponents contained in Engine Functions are the Renderer, Audio, Addons, and Human Interface Devices components.

Renderer is used by Game Logic and Editor to display GameObjects during gameplay. It depends on Core Systems for access to the file system which it uses to gather the sprite data to be rendered. It is also used by Physics to display prompts to the user about trigger events.

Audio's dependencies and uses are similar to Renderer in that it is used by Gameplay Systems and Physics and depends on Core Systems. It is controlled directly by the subsystems that depend on Engine Functions, via an instance of the SoundManager class, to play the sound assets when relevant gameplay events occur

Addons manages the add-on content available for SuperTux. Since the game can never be sure if such content is present, or still present if it was previously available, Core Systems depends on Addons to provide a list of available content whenever engine configuration is performed. With the exception of this atypical dependency, Addons typical operation is much like the resource management functionality found in the Renderer and Audio subsystems. For this purpose it depends on Core Systems and is used by the Gameplay Systems when addon content is requested by the user.

Human Interface Devices is responsible for collecting input from the user and making it available to the Gameplay Systems. It does so in two ways: Gameplay Systems can request new input from Human Interface Devices or, if the user is in a menu screen, Human Interface Devices can tell Gameplay Systems to process input. In order to gather input information Human Interface Devices makes use of Core Systems and SDL.

#### Core Systems

Core Systems houses various utilities that the rest of the system needs to access like resources, getting input, and doing certain calculations. It contains the SuperTux math libraries, as well as different file reader objects and the PhysFS library files required by them. It is used by Gameplay Systems to gather information about gameplay resources, such as levels and player progress, as well as to allow the editor to create new resources. In the same way, it is used by Engine Functions, and the Renderer, Audio, and Addons components specifically, to gather game assets from the file system and by Human Interface Devices to get user input. Physics depends on Core Systems for its math libraries which are necessary for computing collision detection. Other than engine configuration, Core Systems has no actionable intentions during runtime. Its function within the system is most often to accommodate the operation of the systems that depend on it.

## Design Patterns

Since SuperTuxs implementation is entirely object oriented, there were many instances of object oriented design patterns within the sources. These are two of the patterns which we identified and found interesting because of how much impact they have on what SuperTux is capable of doing.

#### Abstract Factory

An abstract factory works by creating instances of concrete class objects that inherit from the same abstract class. This design pattern can be useful in many ways since it allows for a single instance of a class to generate many types of objects that inherit from the same parent class. This means the same code can be used in multiple computational contexts by passing the context dependent details along to the abstract factory when generating the required objects.

SuperTux makes use of this design pattern in the ObjectFactory class. It creates an instance of a factory for each type of GameObject in the game and then uses these factories when a request for an instance of the concrete class that each instantiates is received. This functionality is useful for managing the GameObjects in the game during gameplay as well as while editing a level.

### Template

Contained within the supertux folder is a pair of classes named LevelTransformer and FlipLevelTransformer. LevelTransformer is an abstract class that defines functions related to transforming either a level or a Sector. FlipLevelTransformer is a concrete class that inherits and overrides these abstract function definitions, providing the implementation details of the type of transformation it is meant to perform. In the case of FlipLevelTransformer it can be used to invert the textures of all GameObjects in the level or Sector. Although there are no other instances of concrete classes that inherit from LevelTransformer, this design pattern has the potential to create similarly profound changes to SuperTux levels by overriding these two abstract functions and applying them during play.

## Sequence Diagrams

### Sequence Diagram: Loading a Level

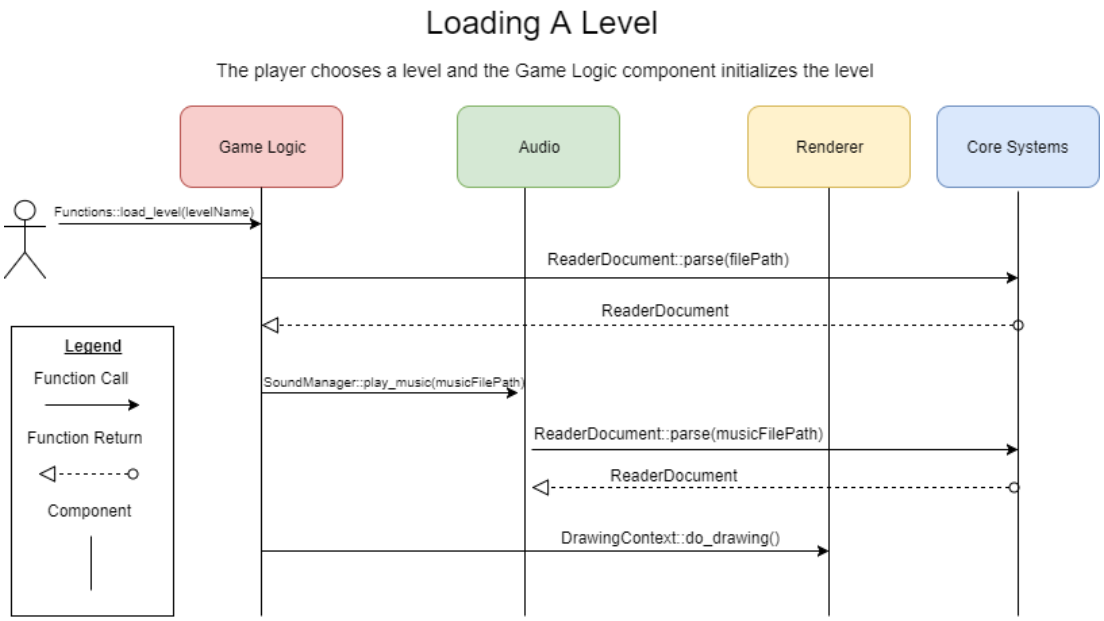


Figure 4: Sequence diagram of the game's process for loading a level, gathering the necessary resources and displaying the level to the user.

This diagram visualizes the sequence of function calls which are performed between three of our architectures high level subsystems and four of these subsystems subcomponents. When Game Logic receives a request to load a level it gathers the level information from Core Systems.

The level information is returned to Game Logic in the form of a ReaderDocument object which can be used to determine how the level should be loaded. Game Logic then requests that the SoundManager play the levels music, a file which is also gathered from Core Systems. Once again, Core Systems returns a ReaderDocument which contains the relevant information about the music file to be played by the SoundManager. Finally the Game Logic component builds a DrawingContext item which contains information about all of the GameObjects in the level information. It then requests that the DrawingContext perform the rendering of these GameObjects, which it does with the help of other objects from Renderer.

**Sequence Diagram: Fireball Hit**

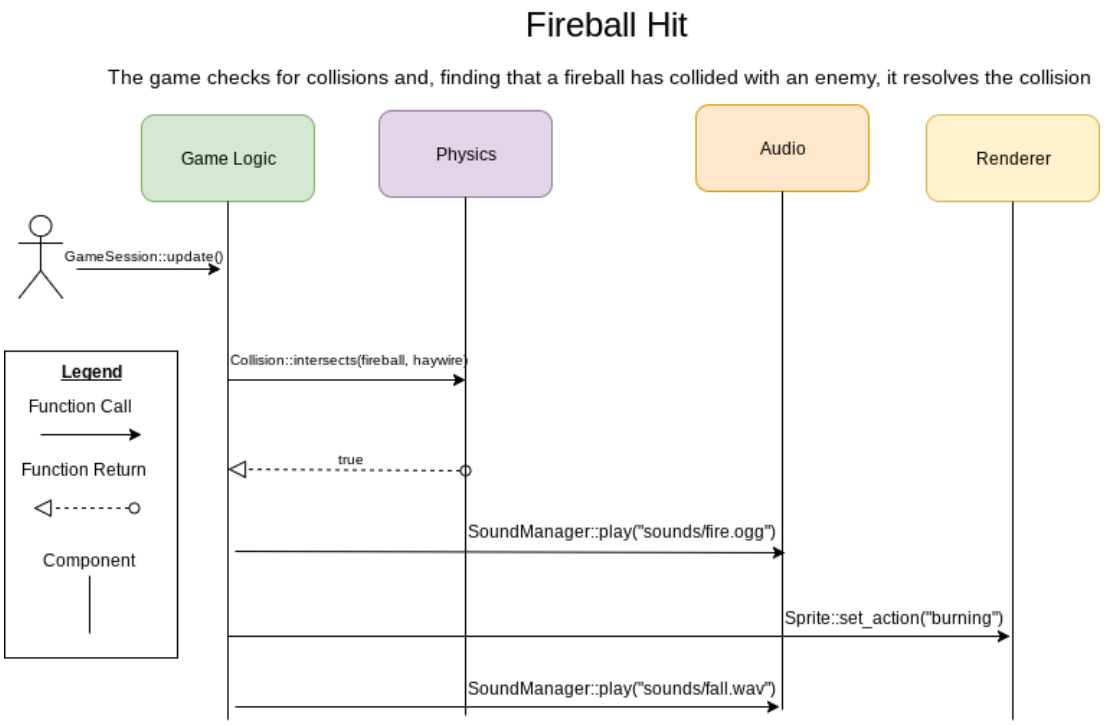


Figure 5: Sequence diagram of the game's response to a collision between a fireball, thrown by Tux, and an enemy.

Fireball Hit details the function calls between Game Logic, Physics, Renderer, and Audio as it detects and resolves a collision between a fireball, which is a type of Bullet object, and a BadGuy. When the GameSession receives a request to update, it uses Physics to check each pair of GameObjects in the current Sector for a collision. In this instance Physics finds that a Bullet and a BadGuy have collided and so it returns the value true, informing Game Logic of the collision. Game Logic informs the Bullet of the collision and then informs the BadGuy as well. The BadGuy determines that it has been hit by a fireball and so it requests that the SoundManager play the fire sound and the Renderer change its sprite to the burning sprite. The BadGuy then determines that it dies as a result of the collision and so it sends another request to Audio to play the fall sound.

## Subsystem Investigation

The subsystem we have chosen to investigate is Gameplay Systems, which contains the Game Logic and Editor components. Within our original conceptual architecture, we had separated these two systems into separate high level subsystems and expected a one way dependency from Editor to Game Logic, believing that the editor would operate like regular gameplay but with the ability to manipulate the level. This way, functionality that is already available in Game Logic could be reused such that the way in which a level is processed and presented to the user is uniform among the two game modes. This turns out not to be the case, although there is seemingly no reasonable justification for this divergence other than the possibility that some functionality which was implemented for use with the editor became useful enough to become ubiquitous in Game Logic as well. One might come to the conclusion that since the codependency between these two subsystems was not anticipated, and because we still do not believe that it should be there, then it should not be featured in our revised conceptual architecture. The issue with following through with this possible change is that references from Game Logic to Editor are more common than references from Editor to Game Logic. And so, it is now our belief that this quirk of the system is too much a part of how SuperTux actually works to be disregarded as a lazy workaround.

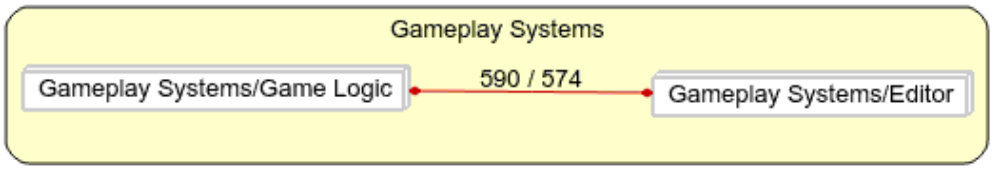


Figure 6: Understand dependency graph of the Gameplay Systems subsystem.

The many dependencies from Game Logic to Editor are due to only a small portion of the functionality in Editor being used many by types of things in Game Logic. The most common offenders are objects that inherit from GameObject calling functionality from the ObjectOption and ObjectSettings objects found in Editor. From their uses, it appears that these objects were initially created as part of the system that the editor uses to create new instances of GameObjects to be added to a level. It could be that this functionality became necessary as part of the Game Logic system as well and, instead of reworking the objects that were repurposed, they were used in the same way in both systems. Another typical use of Editor by Game Logic is a check, which can be found in many of Game Logics objects, to see if the current game mode is the editor or normal gameplay. This is done by calling the is˙active function in the games instance of an Editor object whenever distinguishing the current game mode. Again, ideally these systems should have no knowledge that an editor exists at all, especially during regular play.

The Editors uses of Game Logic were expected and found to exist largely in the ways we expected. These convergences included uses like the Editor using object definitions in Game Logic to build the systems required to create and manipulate these objects, as well as its use of the GUI system to display menus for the user. However, Editors use of Game Logic means certain quirks of Game Logics operation are inherited by Editor. One such quirk is the way in which Game Logic manages input, particularly for use with the menu system. The

codependency between Human Interface Devices and Game Logic means that Editor also needs to handle its own input by using Human Interface Devices. Where this becomes troublesome is in Game Logics menu system, which has Human Interface Devices tell it when to process input. Therefore, in order for Editor to get menu related input it needs to use to menu system to manage an EditorMenu object. Editors use of the menu system in this way should be encouraged, since it is reusing Game Logics functionality, but Editors use of Human Interface Devices directly as well means that SuperTux handles user input in three different ways across two game modes.

# Reflexion Analysis

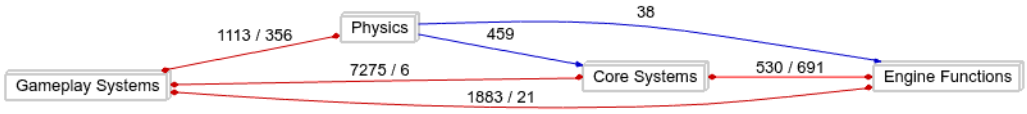


Figure 7: Understand dependency graph using our new conceptual architecture, featuring the divergent dependencies

## Divergences

Throughout our analysis of the source code of SuperTux, we ran into quite a few dependencies on other systems that were a surprise to us. Some of them were things we overlooked due to not knowing exactly how SuperTux was made and others were unexpected due to how they were used. The divergences we found between our revised conceptual architecture and our concrete architecture were the dependencies from Physics to Engine Functions, Core Systems to Engine Functions, and Engine Functions to Gameplay Systems. These divergences need to be examined more closely in order to be explained.

Physics use of Engine Functions is due to the trigger system which prompts the user about Tuxs proximity to trigger objects. We believe that these prompts should have been implemented using pre-existing display and audio methods instead of calling Renderer and Audio directly.

Core Systems dependence on Engine Functions is due to two parts of the engine initialization operations that SuperTux performs, setting up Renderer, Audio, and Human Interface Devices and also checking which addons are available. We did anticipate that Core Systems would be responsible for engine initialization, however, we expected it to only provide system information to the subsystems that depend on it and not to reference them at all. Ideally engine initialization would have no knowledge of the systems that make use of the Core Systems component.

The dependency from Engine Functions to Gameplay Systems is due almost entirely to Human Interface Devices controlling the menu system in Game Logic directly instead of the menu system, or another more suitable system, querying Human Interface Devices about input events. There is also a single dependency from Renderer to Game Logic which is due to the Font objects inclusion of the source file responsible for the Screen object. This reference goes unused by the game in its current state.

### Comparison of Conceptual and Concrete Architectures

Although there were many divergences between our original conceptual architecture and what we found when we began to examine the SuperTux sources, we were surprised by how much of the reality converged with our expectations. Our revised conceptual architecture is significantly different than our original, in their styles and in their interactions between subsystems, but many of the dependencies are the same. The similarities that they share made recreating the conceptual architecture, in terms of what we expected before, straightforward. It is because of this familiarity with the ways in which the subsystems interact that we feel justified in our disagreement with some of the dependencies found in the concrete architecture. All of the divergences between our revised conceptual architecture and the concrete architecture are due to atypical uses of systems where better ways for them to be used already exist in SuperTux or are possible with modification to only the subsystems directly involved. It is for this reason that we have not made further modifications to our conceptual architecture to include these divergent dependencies.

## Lessons Learned

During the revision process of our conceptual architecture we quickly learned that having a hybrid of three architecture styles was inefficient and would be difficult to follow for a development team. While analyzing our architecture we also realized that we needed to increase the cohesion of our conceptual architecture by combining subsystems. Our team decided to take inspiration from other groups and investigate an object oriented design. After reworking the conceptual architecture we learned that by taking more time to plan out the high level subsystems and focus on high cohesion we could create an architecture that was much easier to follow and thus easier for the development team to stick to.

Another important lesson we learned was that no matter how cohesive the conceptual architecture is there will be unexpected dependencies in the concrete architecture. Our team did our best to increase cohesion but with the software being open sourced it would be very difficult to enforce the dependencies created in the conceptual architecture. It would be quite easy for a new developer to start working on the software and ignore the proper flow of data and instead add a new dependency that suited their needs. By looking through the code and finding these unexpected dependencies our team learned that cohesion in the actual implementation was most likely going to be higher and that some dependencies might need to be added to make the developers' lives easier. In assignment 1 our team learned that using the sequence diagrams to verify dependencies in the conceptual architecture was very useful and once again we found the sequence diagrams to be a useful tool while analyzing dependencies. When we created our concrete architecture and found a few unexpected dependencies we found that using the sequence diagrams really helped to explain why some of those dependencies existed.

Once again the biggest takeaways from this experience has been to fail faster. While analyzing the conceptual architectures of other teams and taking inspiration for our revised architecture we learned that we needed to ignore our previous assumptions and look at the architecture from a different perspective to come up with a more accurate architecture. We learned this lesson in assignment 1, however, with the different aspects of software architecture looked at in assignment 2, we once again learned that it can be worthwhile to stick with an idea but it is also important to recognize when we are chasing a dead end and start fresh.

## Dictionary

Resource/Asset - A prefabricated file containing the data required by the game. Possibly a sprite sheet or sound file.

Sprite/Texture - A sprite or texture refers to an individual image file associated with a SuperTux GameObject. These resources might be organized into a collection of similar and related resources forming a sprite sheet.

SDL - Simple DirectMedia Layer: an external utility used by SuperTux to handle input from and output to the user.

Source code/Sources - The files from which the SuperTux game is built. Reading the files related to an aspect of the games operation can give some insight into the games implementation details.

Sector - A Sector is a SuperTux object that allows Game Logic world space to be divided into independent subspaces. Any individual Sector might represent a portion of that world space; possibly screen space or a secret area that only requires Game Logic computations when it has been discovered by the player.

GameObject - The abstract class from which all entities that might appear in an instance of the SuperTux game inherit. This includes things such as BadGuys and SecretAreaTriggers.