Conceptual Architecture

Super Mario ... Oops I mean Tux!

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

2017-10-23

Contents

List of Figures

Abstract

Our conceptual architecture for the game SuperTux makes use of three architectural styles: layered, implicit invocation, and repository. Inspired by the runtime engine reference architecture[1] we set out to identify and group functionally distinct components that might be required to create a game like SuperTux. Having considered other architecture styles, such as object oriented and client/server, we determined that a mostly layered architecture would allow functionality to be further abstracted at each layer so that functionality at the top level could happen entirely in terms of game logic. This led to the development of an implicit invocation and repository hybrid architecture for communication between these high level components.

This report explains and justifies these decisions, providing detail about the functionality of each component in relation to the others. It also includes a summary of the path we followed while creating this conceptual architecture as well as some of the pitfalls we encountered and the lessons we learned as a consequence.

Development

SuperTux is an open-source computer game, inspired by classic 2D side-scrolling platformers from the 8-bit era[2], that started development in 2003[3]. The game is officially supported on the Windows, OS X/macOS, and Linux platforms[4]; and like many such games it relies on technologies like OpenGL, OpenAL, and SDL2[2].

SuperTux was started in 2003 by Bill Kendrick[6]. Since then, many developers have joined the development team and many of them have made contributions solely as open source developers[2]. Communication between developers takes place mainly via IRC and a mailing list[5] and because SuperTux is an open source project, with a variable group of developers working on it, there have likely been many issues in the organization of development roles and the division of responsibilities. While some team members are credited as major contributors, like Ingo Ruhnke who created many of the game's visual assets[6], it is often the case with such projects that most contributors will only make minor contributions, or adjustments to preexisting code, without any prior communication with the development team. This method of cooperation could drag out development time far beyond the time that would be required by a smaller dedicated team to create the same game, due to redoubled efforts and differing opinions about the direction the project should take.

Derivation Process

Reference Architecture

Our point of entry for the derivation of this conceptual architecture was the runtime engine reference architecture found in Jason Gregory's *Game Engine Architecture* [1]. Using it as a reference, we identified subsystems and functionality within the diagram which would also be necessary to create a game like SuperTux. Among the chosen items were highly abstracted systems like resources, physics, audio, and human-interface devices. These were systems which any program that can be called a video game would need to have in some form, and so it simplified our initial approach to creating an architecture for SuperTux by generalizing the problem to that of creating a simple game architecture.

Once we had chosen all the components which would be required to make something like SuperTux we began specifying the functionality that each of these subsystems would need. For instance, SuperTux can accept inputs from several sources and abstract them into game system inputs, a process which would require the ability to identify and manage these multiple sources of input. The functionality of the human-interface devices component was therefore identified to be device management, to handle multiple devices, and player-IO (where the O in player-IO would be a possible rumble feature). Defining how these systems would work and how they would work together led us to the next step in our derivation process, separating and reorganizing components, and considering other functionality which was more specific to SuperTux.

So far, this is how we had conceptualized our architecture.

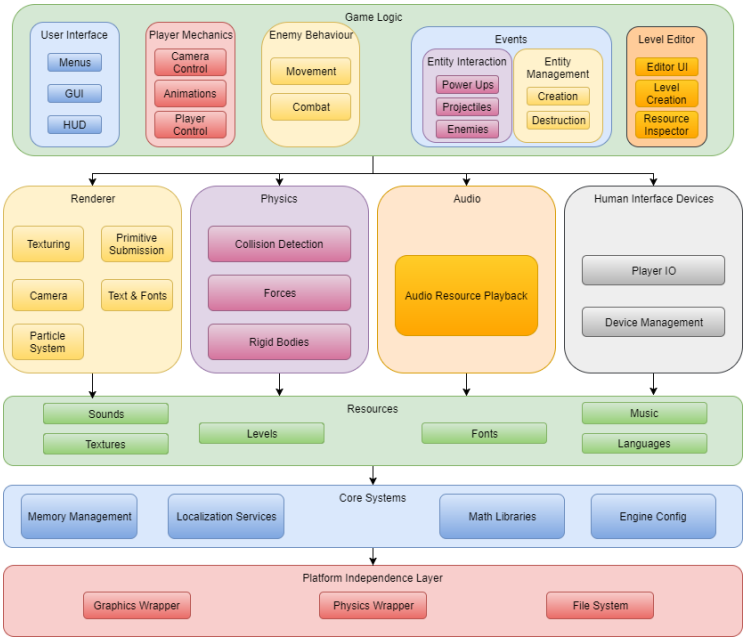

Figure 1: First draft of our conceptual architecture; strict layered and clearly influenced by Gregory[1]

Distinguishing and Reorganizing Components

Not long after completing this iteration of our conceptual architecture it became apparent that some of the functionality of SuperTux, at least in the way that we understood it, was not well suited to a software architecture that was so strictly layered. In many possible use cases for the game it would be necessary for information and instructions to flow between components and to not only have systems above tell systems below what to do. This led to major reorganization of many of the subsystems of our first attempt, and in many cases meant separating the functionality of some systems into more cohesive systems themselves. This was the case for sub-components of the draft architecture such as the events system and the level editor since they either provide functionality to or require functionality from the

game logic component but are distinct systems that require access to components outside of game logic. Similar gameplay functionality components, like animation and the user interface, would also require access to information from outside the scope of the game logic component however, unlike those subsystems which were moved outside, the information that they require could be gathered and stored within game logic and therefore they would not need to be aware of components at different layers.

This revelation about the coupling of certain components was prompted by our observations about the way that the game works. There are only three major sources of change in the game that would require updating the game state, counting only those changes which are meaningful to the player. These events occur when the player character Tux moves, prompted by the player's input, when something other than Tux moves, such as an enemy patrolling its area of control, or when two things collide, such as Tux and an enemy or an enemy and a fireball. When any of these events occur the game state must change to reflect the game's acknowledgement of the event and its resolution. This would require communication between the game logic component and the physics component which would provide the necessary information to the involved subsystems. Considering how this could be achieved led us to redesigning our conceptual architecture in a way that facilitated this type of communication, ultimately causing us to integrate elements of the implicit invocation and repository architectural styles.

Redesign

Taking our ideas about the flow of information into account, we repositioned three subsystems and restructured the game logic component to function as a self contained repository style architecture. The systems that were moved include the physics system, the events system, and the level editor.

We placed physics on the same vertical position in our diagram as the game logic component to show that, despite game logic depending on physics in the same way that it depends on the members of the physics component's former layer cohort, its functionality is fundamentally different than those other systems since it depends on the events system, which in turn depends on game logic. This dependency cycle was required to allow the broadcasting of information about collision events from the events component to the subcomponents of game logic by means of implicit invocation. This change meant that the events system no longer functioned like other subsystems in game logic and needed to be reorganized. Moving events outside game logic made both more cohesive, as did the removal and replacement of the level editor. Although editing a level requires access to game logic in order to display the level being edited in terms of the game itself, since objects in the level need to be rendered and game entities need to behave like they would during regular play, the creation and retention of the generated game map requires access to the functionality found in the resources component, which happens to be two layers down from game logic. This meant that a strict layered architecture was no longer feasible nor required; and so our conceptual architecture experienced further modifications, leading us to our final result.

Considered Alternatives

Given that SuperTux is a C++ program, and that C++ has powerful object oriented features, introducing elements of an object orient architectural style into our architecture was an attractive option. Although SuperTux almost certainly does include elements of object

oriented programming in its implementation, we believe that these implementation details were more of an aspect of the design of the subsystems than part of the overall system architecture. It was for this reason that we disregarded the object oriented style in the creation of our conceptual architecture.

Another such style that we ruled out as too implementation specific was the possibility of contextualizing the game as a client in a client/server architecture where the server would be the remote server which serves SuperTux with custom maps and additional languages. This style seemed to us to be wrong in most uses of the game, seeing as how the player is never required to access the server since it is not an integral part of typical gameplay. The role of the server was relegated to that of a secondary actor in only a few possible use cases, and therefore does not factor into the high level view of the game's architecture.

Concurrency

As we have envisioned it, our conceptual architecture makes no use of, nor has no need for, concurrent execution of instructions. In our reasoning, and in our descriptions of the functionality of our total architecture, instructions happen sequentially as information flows between different portions of the system. While it is almost certainly the case that, at the lowest level, graphical, auditory, and other computational type instructions would be separated and executed using specialized computational units, these details were far too implementation specific to factor into the overall conceptual architecture.

Conceptual Architecture

In terms of architectural style, our conceptual architecture experienced some significant changes, from the initial influence of the reference architecture to the redesign that introduced two new styles to the overall architecture. These changes brought the total number of stylistic choices up to three: layered, implicit invocation, and repository. As we have conceptualized it, the lowest level subsystems, these being the hardware abstraction layer, core systems, and resources, are organized in a strict layered configuration with game domain specific components above forming a dependency cycle through which information about physics events could be made available to the subcomponents in the game logic system via implicit invocation. Within game logic the subsystems would be organized in a self contained repository style with the current game state as the blackboard which the different subsystems, such as the animations system and player mechanics system, would modify when relevant events occur. Altogether, this is how we've conceptualized it visually.

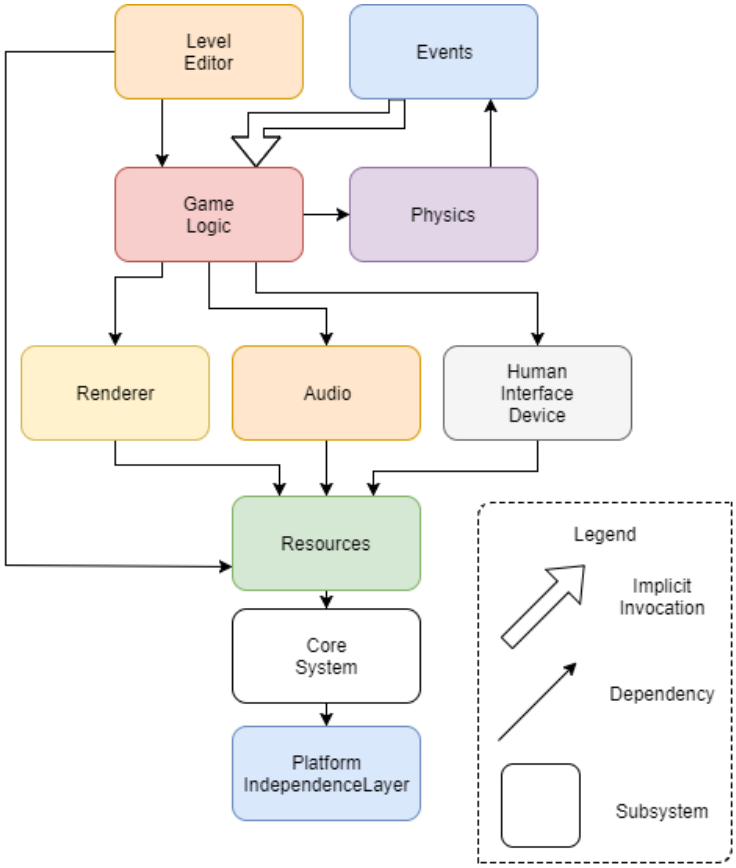

Figure 2: Finalized conceptual architecture (aspects of the game logic component are not visualize here for the sake of simplicity)

Subsystems

Platform Independence Layer

Holds all of the platform specific functionality that higher level components need not be concerned with. This system would need to provide access to a graphics rendering system for the platform on which the game is being run, as well as a way to interact with the operating system's file system.

Core Systems

This is where more abstracted but similarly vital system functionality would reside. The core systems component would provide math libraries for use with both a graphics processor, for graphics and physics calculations, and the central processor. It might also provide more efficient ways of using system memory, geographic localization services, and engine configuration features.

Resources

Resources holds all of the game assets, like textures and sounds, which SuperTux uses during play. This would require the functionality of a resource manager to collect the relevant data from the lower level layers and to provide the systems higher up access to these resources.

Renderer

The renderer would provide abstracted access to lower level rendering systems. It would need to accept instructions in terms of game logic and associate their arguments to game assets in the resources component, such as textures and particle effects. Other functionality would include an abstracted interface for affecting the camera system present in a lower level renderer.

Audio

Like the renderer, the audio component would receive instructions related to the playback of audio resources and associate the arguments with assets found in the resources layer.

Human Interface Devices

This component would be responsible for gathering input information from the machine, via the layers below it, and abstracting those inputs to correspond with multiple devices in terms of game inputs. For instance, a player might use either a keyboard or a gamepad to play within the same session; the human interface devices system would need to interpret these sources of input as different devices corresponding to the same controls.

Game Logic

We've envisioned the game logic component as a repository style system with a game state subcomponent that is the repository's blackboard. It depends on the renderer, audio, and human interface devices systems for receiving input and sending out display and sound requests from the lower level systems. As well, it depends on the physics system, sending it the current game state at the beginning of each update step, such that collisions can be detected and then sent to the events system to be broadcast back to the game logic component. The game logic system's subcomponents would then receive the collision events for which they are registered and make changes to the game state component to resolve the collision.

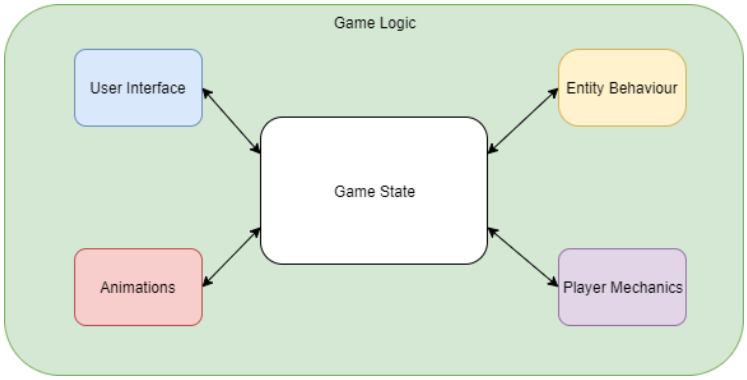

Figure 3: Our vision for the inner workings of the game logic component, arranged in a repository style architecture

Physics

The physics component is responsible for detecting collisions that occur between the different entities in game space. Once it has determined where collisions have occurred it passes the set of collision information to the events system to be made available to the subcomponents of the game logic system. In addition, it would be responsible for making sense of the physical properties of objects within the game, like the rigidbodies of enemies and the level geometry, and applying physics forces to these objects, such as when Tux performs a quick turn in the opposite direction and continues sliding along his previous heading.

Events

Once the physics system has finished its computations that information needs to be sent back to the game logic system so that the game state can be updated. This is the role of the events system. It works by making the information about each collision available to the subcomponents of the game logic system via implicit invocation. These subcomponents can register an interest in certain types of collisions which it might have some role in resolving. For example, the animations system would be interested in collisions between enemies and Tux's fireballs since the enemy's animation would be required to change to signify that the enemy has died as a consequence of the collision. However, collisions between enemies and other enemies are likely of no interest to any of the subcomponents in game logic, and therefore they would not need to be notified of such events.

Level Editor

Of all systems the level editor's functionality is entirely unique. This component allows the user to create their own custom SuperTux levels and even to make these levels available to other players online. Although it would require access to game logic in order to make the information about the level being created comprehensible to the creator, it does not need the level to be fully playable while it is being edited. It does however need access to the resources component in order to make the level available for later use as another game asset.

As a consequence of this dependency, the level editor most significantly violates the layering structure of the architecture.

Information Flow

The following diagrams illustrate the flow of information within our conceptual architecture for SuperTux. First is a use case diagram which demonstrates the intentions a user might have during play, as well as the possible roles that they might take on when experiencing different parts of the game. Next we have prepared three sequence diagrams, two of which detail the process that the overall architecture would use to resolve events that require the involvement of several larger architectural components. The last sequence diagram takes place almost entirely within the game logic component in order to demonstrate the acceptance and resolution of the same event as the previous sequence diagram from the perspective of the repository's subcomponents. This diagram will clarify the means by which the game logic system modifies the game state during play.

Use Case Diagram

During typical use, a player has one of two sets of intentions. In the diagram, these sets are organized with gameplay type uses above and resource management type uses below, which are easily distinguished by their connection to the secondary actor named "Server." Although these two types of actions are entirely distinct, we have contextualized them as belonging to a single user type named "Player" because any player could have either set of desired actions during a single gameplay session, albeit not at the same time.

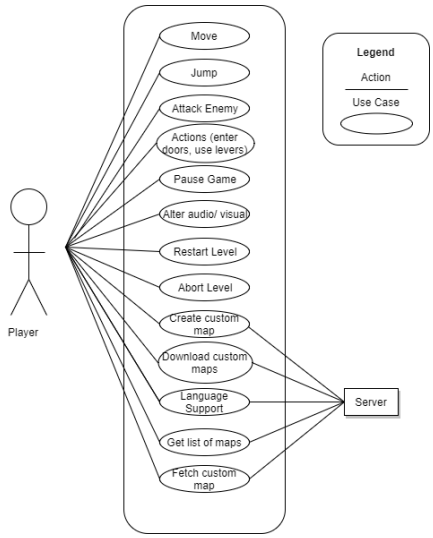

Figure 4: Use case diagram detailing how the user and the file server that stores custom levels both interact with SuperTux

Gameplay actions, those above, are all somehow related to changes within the game state during normal play. Some are highly abstracted, such as the "Actions" action which

is entirely context dependent, while some correspond directly to functionality found in one aspect of the overall software architecture, such as movement.

Resource management actions, found below with a connection to the "Server" actor, are comprised of those actions which alter the resources available to the game logic during play. They are all related to the acquisition, creation, or publication of custom levels (interchangeably referred to as maps) or access to non-standard language resources for the rest of the game.

Sequence Diagram: Loading a Level

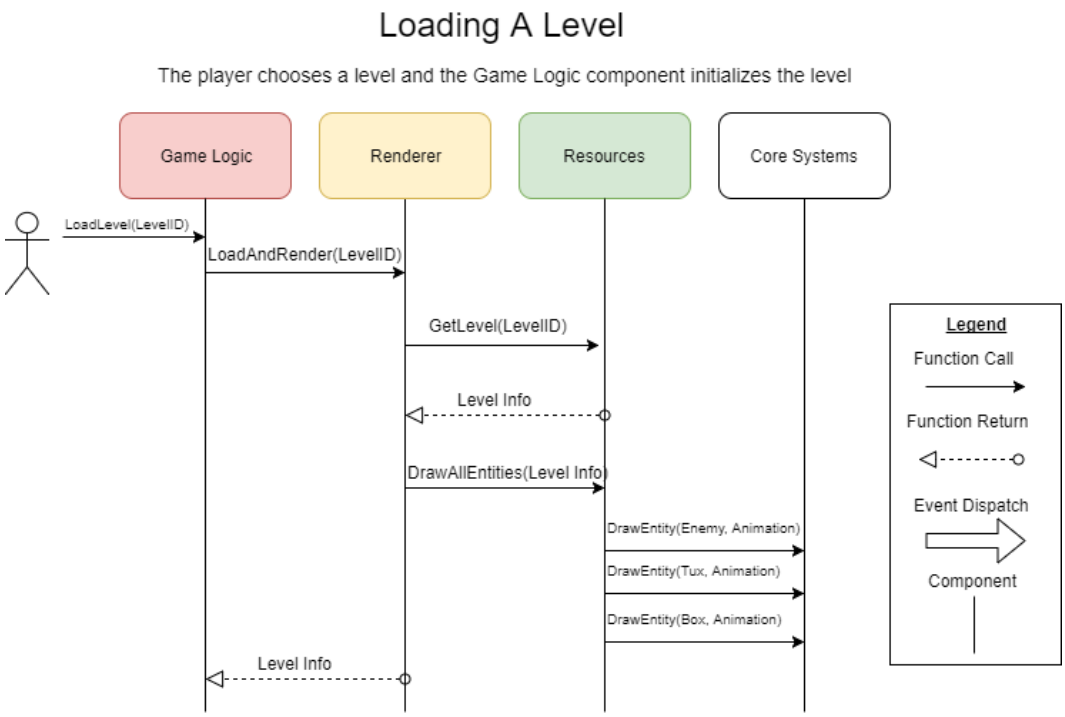

Figure 5: Sequence diagram of the game's process for loading a level, gathering the necessary resources and displaying the level to the user.

In this diagram, the game logic component, using its understanding of the menu system, interprets the user's input as a request to load a level and begins the sequence. It then sends a request to the renderer to serve it the information that it requires about the level being loaded and to render all of the game entities therein to the display. The renderer does this by requesting that information from the resources component and then telling resources to have each entity drawn, a request which is then pushed down through the architecture via core systems. After everything has been drawn, information is sent back to the game logic component and, since all of the required information is in its proper place, gameplay can begin.

Sequence Diagram: Fireball Hit

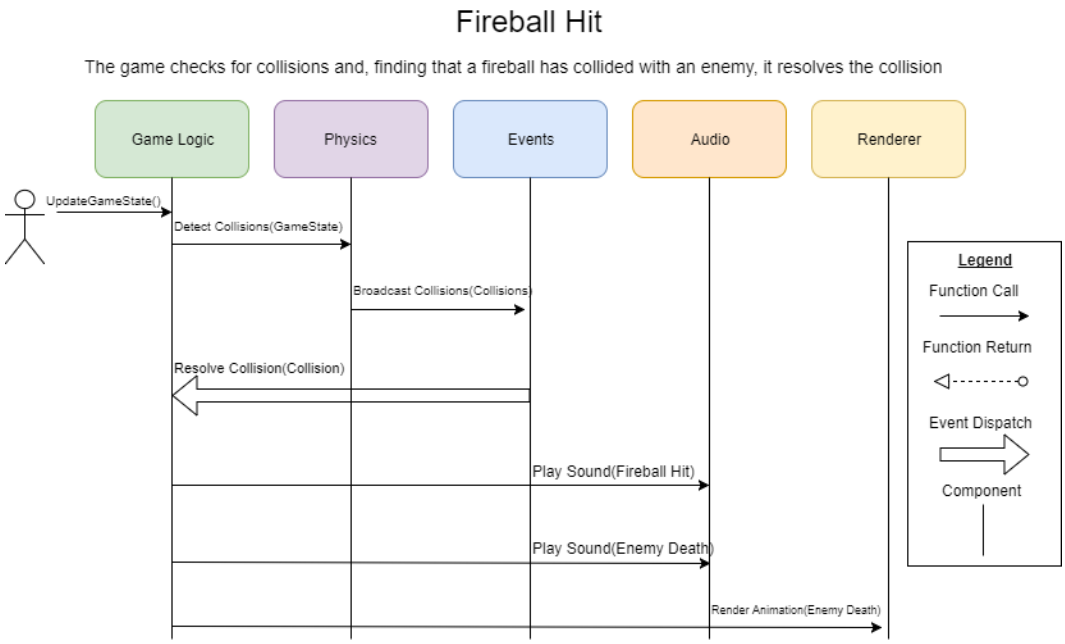

Figure 6: Sequence diagram of the game's response to a collision between a fireball, thrown by Tux, and an enemy.

This process is initialized when the game logic component decides that it is time to update the current game state. At this point it sends the result of the previous update's resolution to the physics component where collision detection can occur. Once the collisions have been detected, the set of all the collisions that occurred is passed to the events system to be transmitted back to the game logic system. Since the subcomponents of game logic have a registered interest in events of this type, they resolve the collision and update the current game state. These changes are then sent to the audio and renderer components, in the form of requests for resources to be used, so that the player can be notified of the change in the current state of the game.

Sequence Diagram: Fireball Hit (Game Logic Resolution)

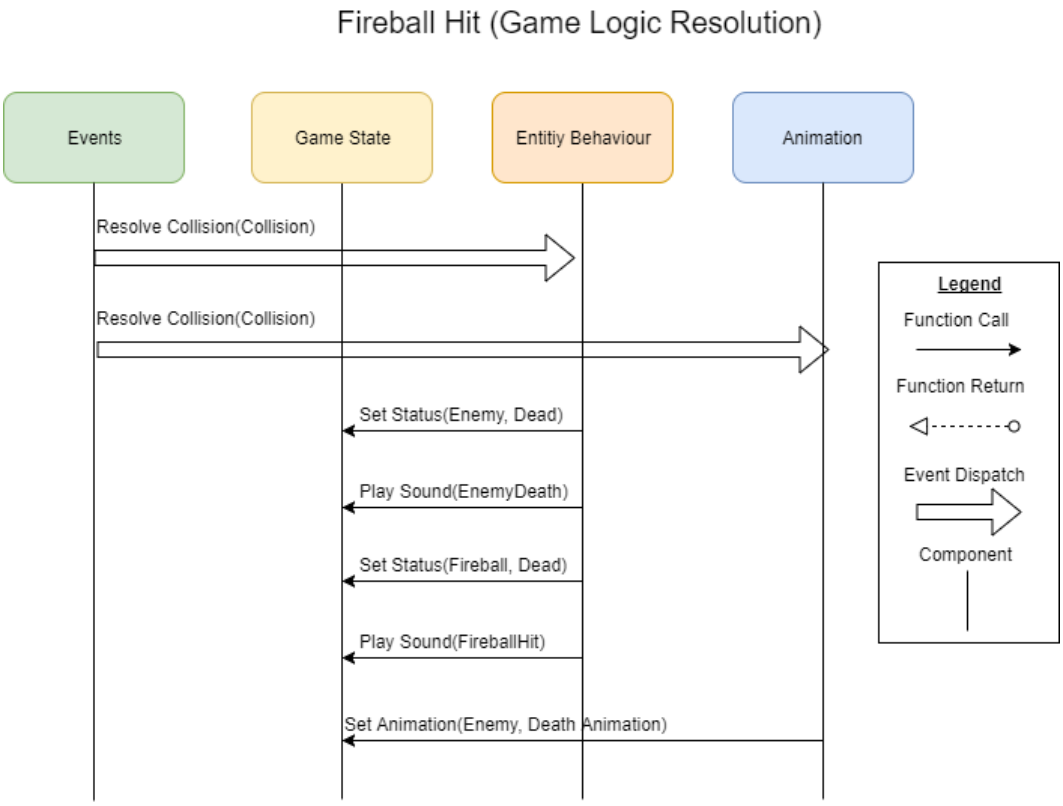

Figure 7: Sequence Diagram: Fireball Hit (Game Logic Resolution). Just like the previous example only from the perspective of the game logic subcomponents as they receive information about the collision and modify the game state

This sequence of requests would occur during the broadcasting of the set of collisions that occured in the previous example, but before the game logic component begins telling the audio and renderer components to make these collisions known to the user. The concerned systems in this example would be the entity behaviour and animation subsystems of the game logic component. Events informs both systems of the collision and each, in turn, modifies a portion of the game state for which they are responsible. The entity behaviour component does so by changing the status of entities and requesting that the sounds associated with the collision are played. The animation component modifies the current animation of the enemy that was hit, however no animation update is required for the fireball since it disappears after the collision. Once all the collisions are resolved the game pushes the new game state the other subsystems, as it does in the rest of the previous sequence diagram.

Lessons Learned

During the derivation process there were many times when we thought we had reached the end of the process only to find that, after only a small amount of scrutiny, there were sections of our architecture that were incompatible with either the rest of the architecture or with the way that we envisioned the game's operation. A previously mentioned instance of this occurring was during the early stages of derivation when it became apparent to us that the flow of information could not always move in the same direction and as a consequence a strict layered architecture would not be a reasonable choice of architectural style. As well, there were other points during derivation, particularly once we began defining use cases and tracing sequence diagrams, when our understanding of how the game worked meant that the functionality of a component, at least in the way that we believe it should function, did not make sense in relation to the functionality of other components.

For example, in the first draft of our architecture we had listed animation as part of the functionality of the player mechanics system. However, once we considered how the the game worked it became clear that Tux is not the only entity in the game that has animations and we became uncertain as to how exactly an animation component would function. This led us to consider the concept of the current state of the game, and ultimately to adopting elements of the repository architecture style into our conceptual architecture.

The biggest take away from the experience has been something that we are constantly relearning during our studies, that is to fail faster. Quickly discovering that your assumptions or beliefs are incorrect keeps a project on the right track and wastes less time chasing down dead ends. We found this to be the case several times over during this portion of our inquiry into the software architecture of SuperTux, and it is a lesson we intend to keep in mind as we continue into the next portion of this journey.

References

[1] Gregory, Jason *Game Engine Architecture*. A K Peters, Wellesley, Massachusetts, 2009.

[2] Karkus476 "About." Last modified January 9, 2016. https://github.com/SuperTux/supertux/wiki/About

[3] SuperTux Website "Home." Accessed October 18, 2017. https://supertux.org/

[4] SuperTux Website Downloads Page "Download." Accessed October 18, 2017. https://supertux.org/download.html

[5] Teufel, Max "Contact." Accessed October 19, 2017 https://github.com/SuperTux/supertux/wiki/Contact

[6] Wikipedia "SuperTux Wikipedia Page." Accessed October 19, 2017. https://en.wikipedia.org/wiki/SuperTux

Dictionary

Component/System - A distinct and separate functional element of an architecture. Used as a method of grouping similar functionality to reduce coupling by minimizing connections to other components.

Subcomponent/Subsystem - A functionally unique portion of a system, be it a component of the architecture or a component of another component. The terms component/system and subcomponent/subsystem might be used interchangeably for the same part of the architecture depending on the perspective from which it is being discussed.

Dependency - A relation between components. A component depends on another component if it requires a service from that component. For instance, I depend on the pizza guy for pizza.

Reference Architecture - A collection of components, and possibly dependencies, that are typical of a type of software or problem domain.

Layer - A portion of an architecture composed of a number of components which have no dependencies higher up and only depend on systems one layer beneath.

Collision - Generated by the physics component, a collision occurs when two entities within the game overlap in game space.

Map/Level - Used interchangeably, a map or level is the collection of initializing information about all of the entities in an instance of play