DOOM 3 BFG
CONCRETE ARCHITECTURE

BASEMENT GURUS

Isaac Chan

Daniel Elmer

Matthew Sherar

NOVEMBER 13, 2015

Table of Contents

Abstract

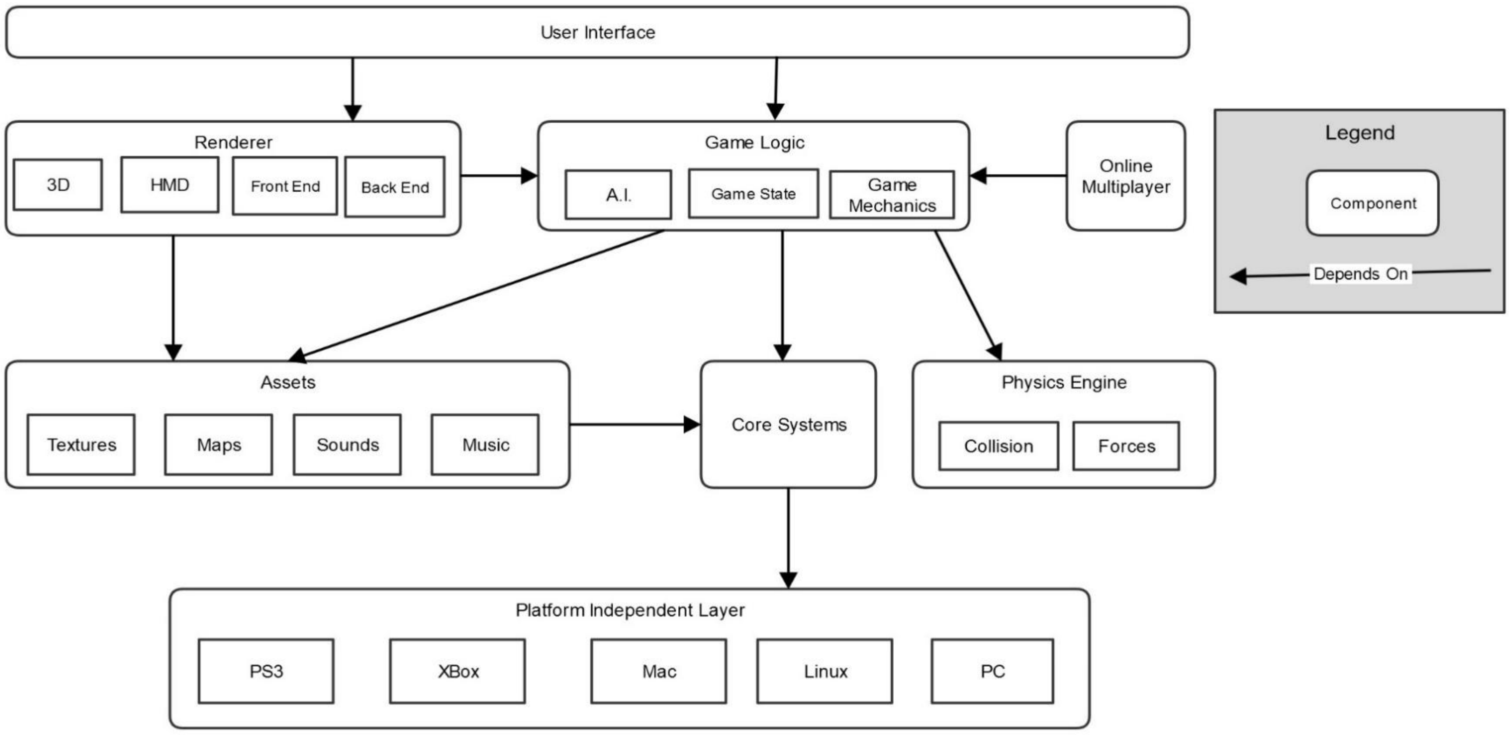
This report provides an overview of the concrete architecture for Doom 3 BFG. A concrete architecture for Doom 3 BFG Edition was developed using our refined conceptual architecture as the basis for the derivation. We will be taking a look at any discrepancies between our conceptual architecture and the newly derived concrete architecture and reasoning behind the unexpected discrepancies discovered. Following that are the alternative architectures we considered during the derivation process as well as the concurrency in relation to the game. A closer look at the Game Logic Subsystem will then be discussed providing a sequence diagram to demonstrate the flow. We will end with the lessons we have learned from deriving the concrete architecture as well as the limitations we had come across during the process.

Introduction

This report gives an overview for the derivation process of the concrete architecture for the game Doom 3 BFG including alternatives. In the analysis of our concrete architecture we will discuss the overall architectural style, noteworthy design patterns, in addition to unexpected and expected dependencies. This will include an updated sequence diagram of a use case to illustrate the concrete architecture rather than the conceptual from the first report. We will go in depth on explaining why certain unexpected dependencies were logical or illogical noting some downsides and benefits. A reflexion Analysis between our concrete and updated conceptual architecture will done, with particular attention to the Game Logic and Renderer Subsystem. The report will summarize concurrency as it relates to this game, ending with Game developer possible limitations and lessons we've learned through this report.

Derivation Process

We began our derivation process by first re-accessing our conceptual architecture. Since the conceptual

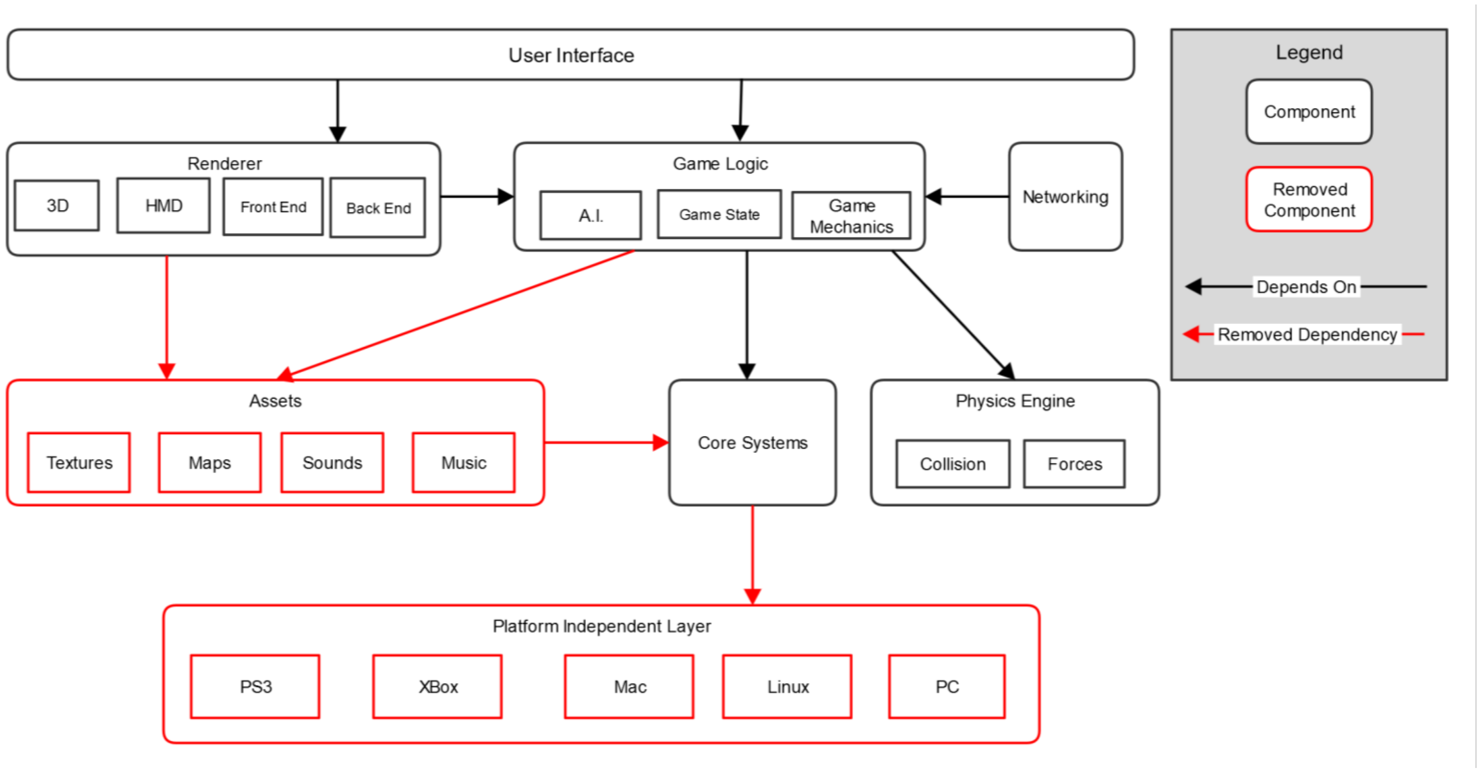


architecture was the basis for our concrete architecture we decided to simplify it and remove any unnecessary subsystems.

From here we began to analyzing the source code. The program Understand was used to derive the concrete architecture. First we had to create our conceptual architecture on Understand. This allows us to see the dependencies between files within Doom 3 BFG.

After creating the architecture on Understand we began analyzing the dependencies. Most of the dependencies did match our conceptual architecture but there were also quite a few dependencies that we did not expect. We referred to the source code to determine the reason for these unexpected dependencies. After examining the unexpected dependencies we managed to successfully derive a concrete architecture.

Revised Conceptual Architecture

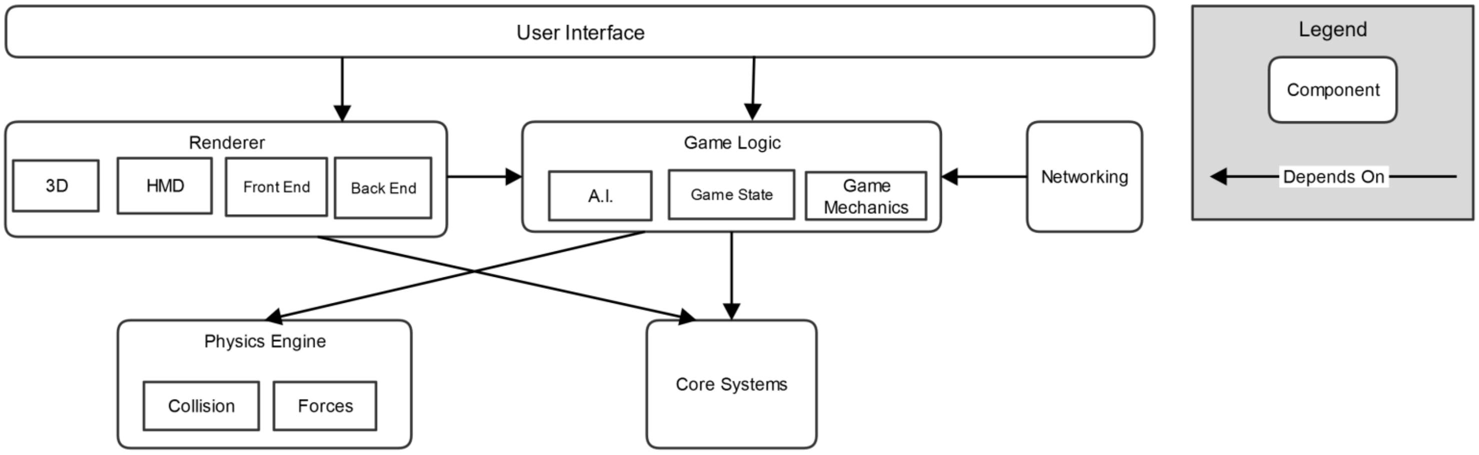


After analyzing the concrete architecture we found that the Doom 3 BFG source code didn’t have an Assets layer. Instead or the Assets code were included in the Core Systems layer. One possible explanation is that the Developers Team didn’t release the Assets when they open sourced the code since they might have been proprietary Music or Sounds used.

Also, we didn’t find any Platform Independence Layer code in the source code either because the team used proprietary code or they didn’t use any platform specific functions in C++ and just relied on the

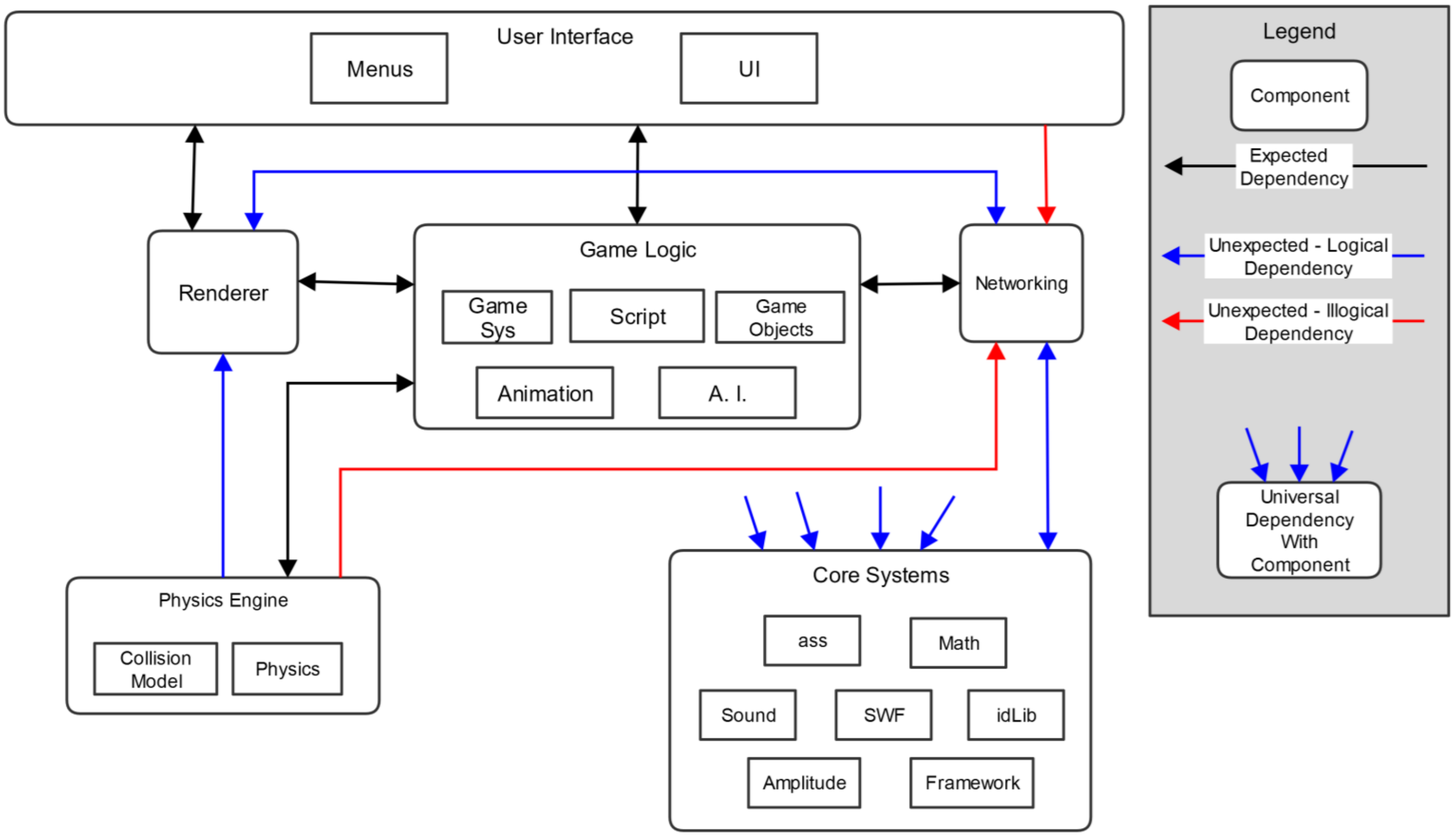
Compiler to build their systems to different platforms. One fact to support this hypothesis is that the team re-implemented most of the C++ STL functions used in the game.

Here’s the updated conceptual architecture after considering the previous two points:



Note that since the Assets component is now included in Core Systems, the Renderer now depends on the Core Systems.

Concrete Architecture



Unexpected Dependencies

Logical

The most significant unexpected, but logical dependency in the concrete architecture is a global bidirectional dependency between the **Core Systems** subsystem and **all other subsystems**. This is logical because the Core Systems subsystem contains files and functions that should logically be integral or common to the functionality of all other subsystems (and to the system as a whole). For instance, the entirety of the multi-threading system, a major performance-boosting architectural design change in DOOM 3 BFG, exists in Core Systems under idlib\sys, as evidenced by the existence of files such as idParallelJobList.cpp and Thread.cpp. Other universally useful files and functions included in Core Systems include math functions in the Math directory, and memory allocation functions throughout. Thus, it is logical that every subsystem would have some kind of communication with Core Systems.

Another unexpected, but logical dependency in the concrete architecture is a bidirectional dependency between the **Renderer** subsystem and **Networking**. This was unexpected initially because the possibility that other player characters and animations in online multiplayer had to be rendered client side was not considered. In actuality, the Networking subsystem makes calls to the Renderer subsystem for the purposes of rendering not only other player characters, but also user interface elements such as the scoreboard. Conversely, the Renderer subsystem makes calls back to the Networking subsystem to get information such as the locations of other player characters, their states, and scoring information. Thus, although not inherently obvious from inspection, the dependency between Renderer and Networking is unexpected, but logical.

The last unexpected, but logical dependency observed in the concrete architecture is a unidirectional dependency between the **Physics** subsystem and the **Renderer** subsystem. This was unexpected initially because it was assumed that a mathematical subsystem like Physics would not have any meaningful exchange between a visual subsystem like Renderer. In actuality, the Physics subsystem makes one-way calls to Renderer in order to extract information such as the character models, player location and state, etc. in order to make the appropriate Physics calculations. It is also logical that this dependency is unidirectional, because there is no useful feedback a visual subsystem could return to a mathematical subsystem. Thus, the dependency between Physics and Renderer is unexpected, but logical.

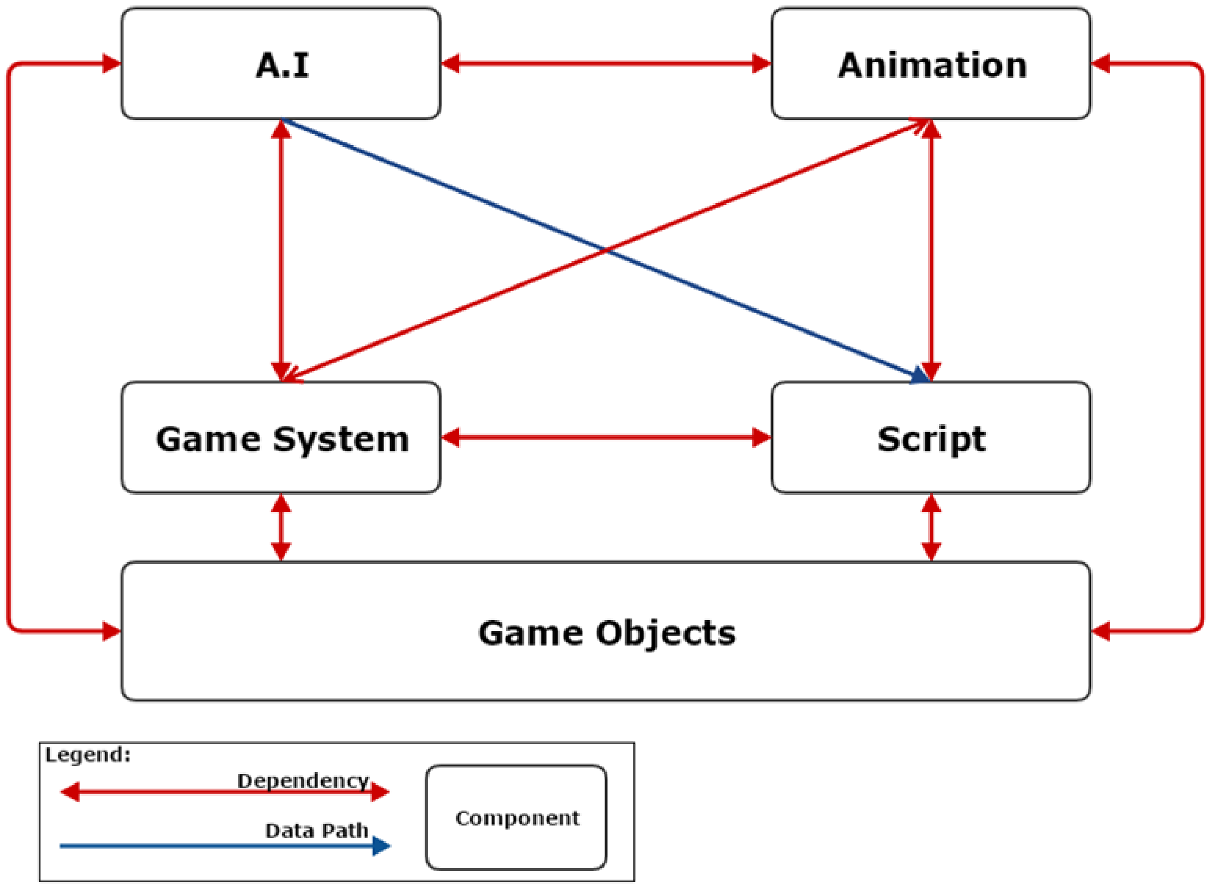
Illogical

The first illogical unexpected dependency observed in the concrete architecture is a unidirectional dependency between the **UI** subsystem and the **Networking** subsystem. Upon initial observation, one could make the assertion that UI communicates with Networking in order to gather information about score so it could render a user interface element such as a scoreboard. However, as observed earlier, scoreboard rendering is handled by communications between the Renderer and Networking subsystems, so having a separate dependency that performs the same task would be redundant. Furthermore, logically UI such as menus, pause screens, etc. are generally client side, as they are not shared amongst players during multiplayer. As such, the unidirectional dependency between UI and Networking is both unexpected and illogical.

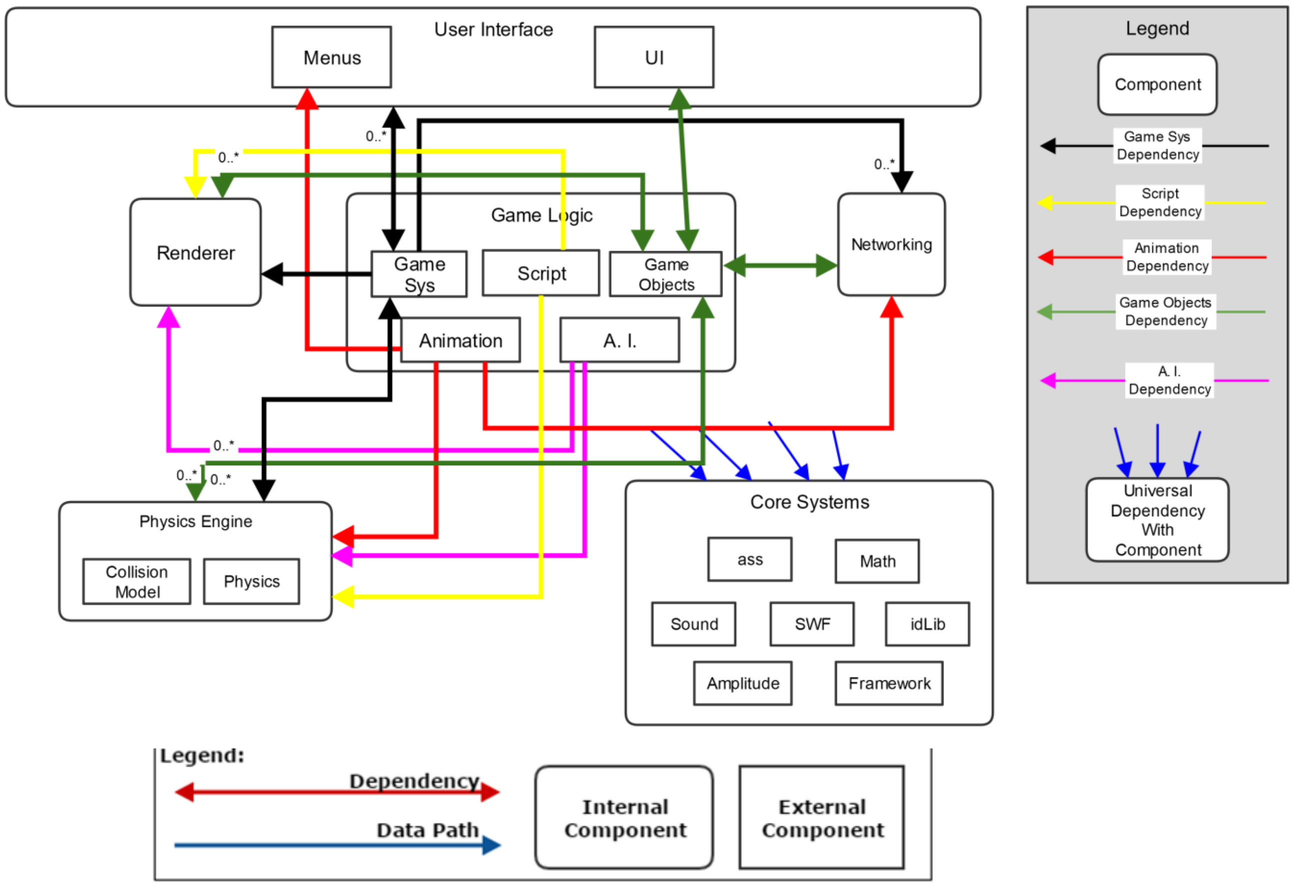
The second illogical unexpected dependency observed in the concrete architecture is a unidirectional dependency between the **Physics** subsystem and the **Networking** subsystem. While one could make the assertion that the Physics subsystem would require information such as other players' locations, states, etc. during multiplayer from Networking, this claim is unsupported by the fact that only 3 calls are made between Physics and Networking. The extremely low number of calls (in relative comparison to subsystem dependencies that can exceed 40,000 calls) makes it highly unlikely that these two subsystems would communicate in such a meaningful way. As such, the unidirectional dependency between Physics and Networking is both unexpected and illogical.

Game Logic Analysis

Internal Dependencies



External Dependencies



The game logic component of the concrete architecture is built primarily to handle the inner workings of game data storage and calculation of game entities and mechanics, effectively the creation of the Doom 3 experience. Five core components are involved in this process. It is based on five components.

ai (Artificial Intelligence) – This component contains object files pertaining to the management of the co-ordination of game entities with other components of code via their behaviour and handles positions/pathing in the game world. This is highly integrated with the physics, animation, and movement of said entities.

anim (Animation) – The component that contains object files pertaining to the animation of game entities, including factors such as frames, skeletal joints, blending, and model structure. This is naturally highly codependent on the Renderer.

gamesys (Game System) – This component has three main functions: containing virtual constructors for game classes and callbacks, recording game data for game saving, and containing all the reference data and commands for the game's rules and mechanics like in-game modifiers for movement, damage values for damaging, among others.

script (Script Manager) – This component is a utility component written for the compilation, interpretation, threading, and programming of Doom 3 BFG's script files. This is for the sake of having a more efficient compiler that is optimized for the project.

Game Scripts – While filed under an individual component, this is the large collection of scripts that represent Doom 3 BFG's in-game objects, such as the player, cameras, and enemies.

Architecture Style

Doom 3 BFG's game logic component uses an oriented programming style, which is natural for the sheer number of individual game objects that need to be represented. The abstraction of the game objects in this way is extremely useful to the developers due to the potential for code reuse and the greater ease of maintenance, particularly when it comes to abstract game rules and mechanics.

The Unexpected and the Expected

The Game Logic component in our conceptual architecture was very basic, only containing Game Logic, Game State, and A.I. While our predictions aside from Game State were correct, we missed several components, including the Scripting Engine, game scripts, Same system, and Animation. While the first four we were just general about, we didn't specify animation anywhere and we assumed it to be contained in the Renderer.

Despite the new components, the dependencies make sense in them for the most part, such as the one-way path between Animation and the Renderer. Animation contains the abstract, mathematical data related to the animation of entities and is integrated into a cohesive construction of the game's mechanics. By constructing the abstract image here, it thus sends that to the Renderer to be processed, acting as bridge from the mechanics and game state to the Renderer.

An example of one that doesn't make sense is the interaction between AI and Renderer. It would be much more ideal to keep most of the rendering-related output of Game Logic to Animation for the sake of cohesion. On further inspection, the functions have to do with the flash of guns, the deaths of enemies, and weapon effects. While we can see why this is done, it's definitely not ideal; that information should be sent to Animation and dealt with from there.

Design Patterns

This component uses a notable Façade style of coding; Game Logic is designed as a unified game library that contains the game's core scripts and logic, but the less abstract uses of the library are focused on a small number of classes. The AI, Animation, Game System, and Game Scripts serve this purpose. They also interact with components outside of Game Logic for the sake of further back end calculation. The script engine would be our façade, since it's the aspect of Game Logic that compiles the entire library and works with the scripting language and engine. This component allows the rest of game logic to be contained in this subsystem of game scripts, but compiled in one place and be used by other components.

Notable Dependencies

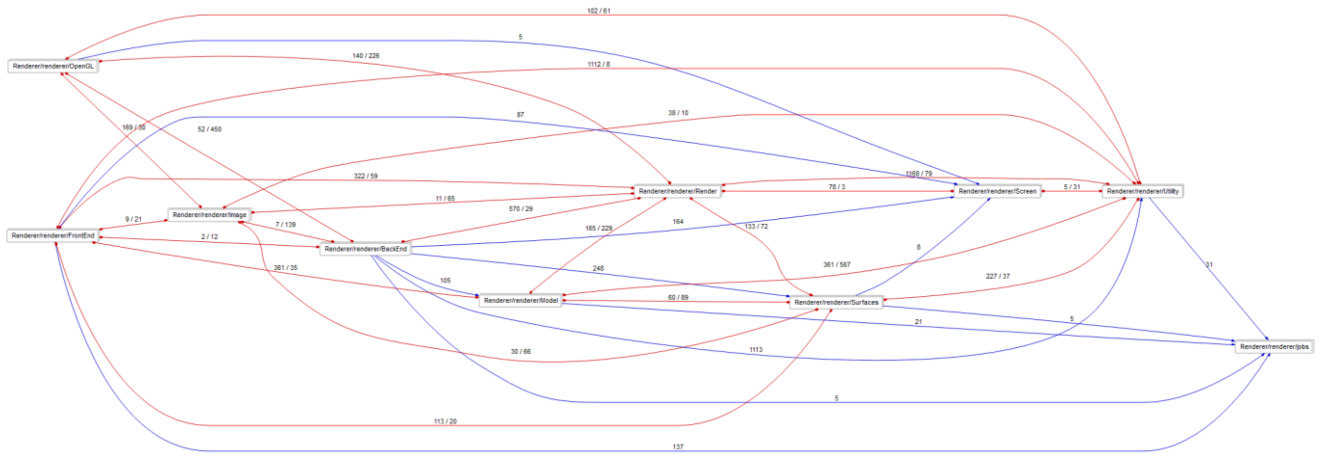
Artificial intelligence

This component handles all the artificial intelligence of non-player characters. So in here would be the path that a monster patrols in the world, whether or not the monster will attack you and how it will attack you.

Game System

Includes the rules of the game, what can and cannot be done in the game. Files corresponding to what actions a player can do will be in here, for example shooting a weapon, what happens when a weapon is shot and how much damage is dealt will be in here. This is also where the saved game data is processed.

Renderer Analysis



The Renderer component is built to handle the game's rendering processes for images, models, textures, G.U.I, and such. Each component of the renderer is highly cohesive to fit the multi-threaded rendering that Doom 3 BFG uses.

BackEnd – This component is a collection of classes for executing draw commands, and is run on one of the game's threads, making it key to the game's thread concurrency. It works by interpreting an abstract representation created by the FrontEnd component, which is done on a separate thread for maximum efficiency.

FrontEnd – This component is a collection of classes for creating an abstract, intermediate representation of the game state. This component is key to the game's thread concurrency, since after the abstract is sent to the BackEnd to be drawn, a new abstraction can be made.

Image – This component is a collection of image-related classes, including loaders, managers, processors, and a large variety of classes specifically for processing JPEG images. It is used to process asset images.

Jobs – This component contains jobs that are used by the FrontEnd and BackEnd to process shadow volumes on models and environments.

Model – This component is a collection of the classes pertaining to the modeling of the objects and sprites in the game. It handles the abstract representation of different game component models and internally manages them.

OpenGL – This is a free source, multi-platform API for the rendering of 2D and 3D vector graphics. In the context of Doom 3 BFG's architecture, it is primarily used in conjunction with the rendering operations and the BackEnd for multi-platform drawing. This component is what actually draws everything to the user's screen.

Render – This component is the collection of rendering operation classes that are used by all the other components in the renderer, as well as handling the automatic rendering process.

Screen – This component contains classes that are used as an abstract representation of the screen size and screen resolution, so that other rendering processes can be altered to fit it.

Surfaces – This component pertains to one of Doom 3 BFG's greatest feats, its interactive surfaces, allowing for the rendering of G.U.I models as well as cinematics into in-game surfaces. This component is also a key part of detecting the interactions of shadow and light on in-game surfaces, determining the light and shadow to render.

Utility – This component contains the rest of the rendering components that didn't fit in other components, such as font rendering, object buffering, a vertex cache, a Graphics API Wrapper for hardware abstraction, and others.

The Unexpected and the Expected

Overall, this component was far more comprehensive than we expected when we made our conceptual architecture. The FrontEnd and BackEnd components were really all that we predicted, as the concurrency of the Doom 3 system relies on their division of the intensive rendering process. The other components we generalized as 3D and Head-Mounted-Display, but these were far too generalized and vague. A notable example of this discrepancy is the usage of a general hardware abstraction layer for the entire architecture, instead of the specification of hardware abstraction to the Renderer in particular through the OpenGL and utility Graphics API Wrapper. We based it off the general 3-D game architecture, and despite the fact that only the Renderer really needs hardware abstraction, we would consider it a flaw in the architecture since if this Renderer was moved to another architecture, their components could require greater hardware abstraction; thus the Renderer would be insufficient.

However, despite the fact that we anticipated very little of the complexities of the Renderer, many of the dependencies make sense. An example of one that doesn't make sense is the connection from Image to Physics, since the component handles pre-determined image assets and shouldn't require any contribution from the mathematical calculations of Physics. Examined further, this is used to determine if any clipping is done off the image if a model is overlapping it. However, this should be handled in the Models component instead to keep the architecture cohesive.

Notable Design Patterns

This component also uses a notable Façade style of coding. This is largely due to the extreme cohesiveness of the majority of the components, except for OpenGL. This is the case since OpenGL is the component that actually handles the operations to the user's hardware to draw the game; everything else acts as part of a large subsystem and OpenGL does the least abstracted work. This effectively makes a whole game thread (the one based around FrontEnd), a helper thread to the BackEnd thread, which is where the dependency on OpenGL is most prominent.

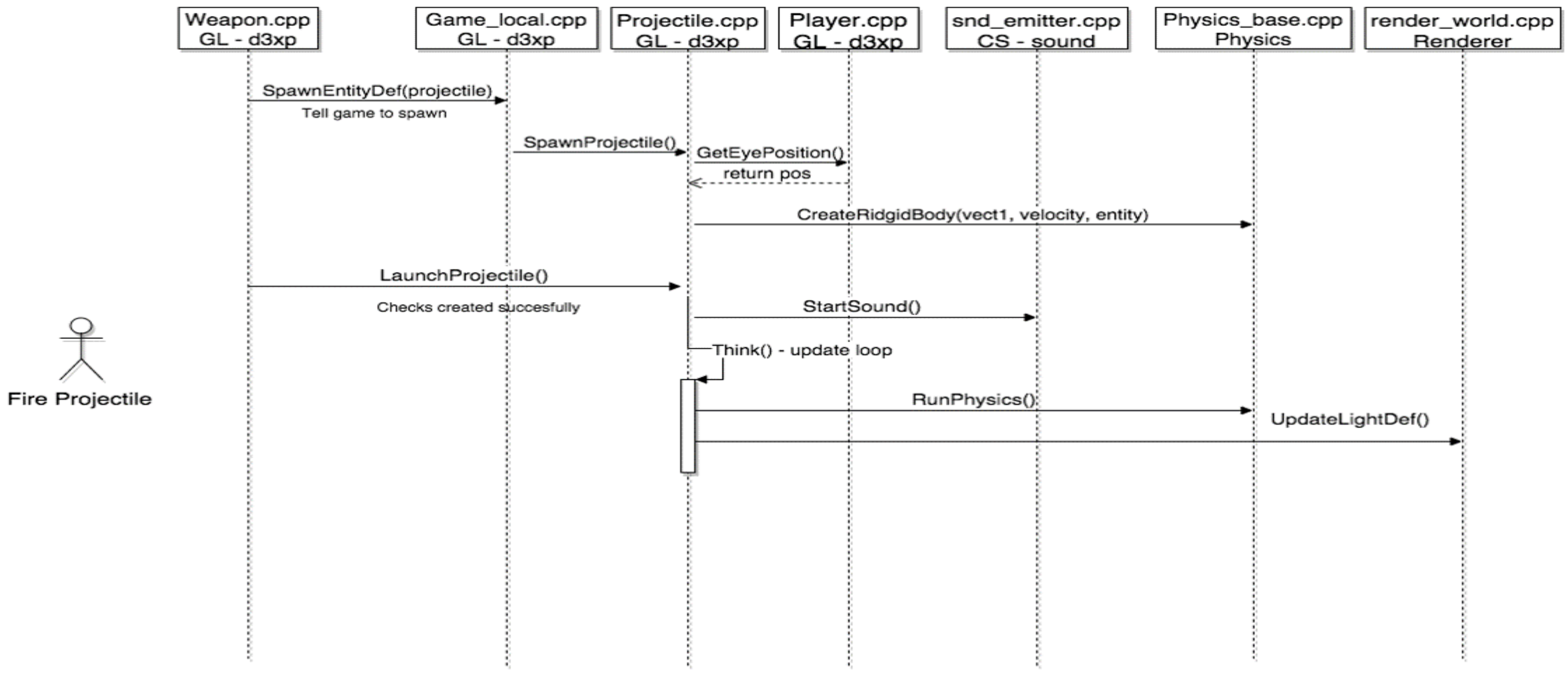
Alternative Architectures

The process of deciding on a final concrete architecture required a large amount of trial and error testing with the Understand software. Initially we tried to keep the platform independence layer that can be seen in our original conceptual architecture. However in order to do this we found ourselves having to remove single files that from various folders to add to the platform independence subsystem. However, as the files within each folder of the game source code are highly coupled with other files in the same folder, removing and reassigning single files complicates the architecture significantly. Having a platform independence layer that isn't concentrated in a single subsystem however would not be desirable as if the game wanted to expand to new platforms it would have to make modifications to several subsystems instead of just a single one. We are inclined to believe that most of the platform independence subsystem would not have been released with the core game source code, and so that is the main reason for the general lack of fit with it in our concrete architecture.

Another subsystem of our conceptual architecture that was not included in our concrete architecture was the game assets. Since the assets are not included in the core source code of the game, there is no need for that subsystem in our concrete architecture. This is intuitive as the software architecture of a game should be based off the functionality of the game, not the assets it uses.

In addition to the removal of components we tried to place the idLib folder inside the game logic subsystem as the idLib had quite a few dependencies on the game logic component. However since idLib contains most of the libraries core to the play of the entire game, it over complicated and blurred the relationships with the other components to game logic when idLib was inside it. A similar effect was found when we attempted to put the gamesys folder inside the Core systems component. While some aspects of the files in gamesys seemed more like core system files, they were highly coupled to other files in game logic that moving gamesys out complicated the architecture.

Sequence Diagram



The sequence diagram shows the control flow within the game logic subsystem analyzed above. When the event that the player has fired a project reaches the game logic subsystem, the weapon class (weapon.cpp) is notified. Weapon.cpp verifies that the projectile launch is possible by the player and then contacts the Game_local.cpp file in order to add an entity to the current game world object, which is controlled by Game_local.cpp. To do this Game_local.cpp creates an instance of a projectile via Projectile.cpp. The projectile classes fetches necessary information about where the projectile was launched by retrieving the info from the player object accessed through the player.cpp file. The Projectile.cpp then goes outside of the game logic subsystem to create a ridged physics body using the Physics_base.cpp class in physics. The Projectile object has it's state update to SPAWNED, at which point when weapon.cpp tries to get the entity from the game world object (GetEntity() to game_local.cpp) a pointer to the ready projectile is returned. From this point the weapon classes launches the projectile through the projectile class. This causes the projectile class to talk to the sound system to start playing the audio file associated with a projectile. Finally the projectile runs through an update loop handled by the projectile.cpp file which continually updates the renderer about it's position and state as well as running the physics calculations on the projectile.

There are noticeably few return statements in the diagram for the number of function calls. This is due to the fact that most of the functions in the game logic subsystem in general are void functions with no return values. Since the design pattern is object-oriented the files that are making function calls to instantiate new objects just keep pointers to the objects they create and keep track of them. For example once game_local.cpp instantiates a new projectile, it keeps track of the object instead of getting a return value from projectile.cpp

Concurrency

In terms of concurrency, the concrete architecture and source code of DOOM 3 BFG reflects the triple-threaded, worker-supported multi-threading model described in the derivation of the conceptual architecture. In summary, running the main executable spawns three master threads, the Renderer Backend, Renderer Frontend and Game Logic, and High Frequency Joystick Input threads, which all perform tasks in parallel with support from two Worker threads. These Worker threads sequentially take tasks called Jobs from a queue called an idParallelJobList.[1]

Evidence that the source code supports this multi-threading model can be seen in the files present under the Core Systems subsystem, under the idLib\sys directory. Here, several files that directly pertain to initiating and maintenance of multi-threading exist – for instance, Thread.cpp and idParallelJobList.cpp.

Furthermore, multi-threading is also supported by the concrete architecture, as evidenced by the fact that all subsystems have a bidirectional dependency with Core Systems (and with the files in idLib). As such, multi-threading is ubiquitous and incorporated globally throughout the system.

Use Cases

Limitations

The program Understand was used to help derive a concrete architecture. Understand has a steep learning curve making it difficult to use it effectively and efficiently. The frequent amount of crashes that occurred while deriving the concrete architecture had caused loss of progress having to restart the derivation process every time. Understand also does not support sharing of the same conceptual

[1] Sanglard, Fabien. 'Doom3 BFG Source Code Review: Multi-Threading'. Fabiensanglard.net. N.p., 2015. Web. 12 Nov. 2015.

architecture so each member had to reconstruct a conceptual architecture which was quite time consuming especially when changes to the architecture were made. The language Doom 3 BFG was written in also limited the analysis process as most members either did not know or had limited knowledge of the programming language (C++) used. Also there was little documentation on code making it hard to understand what was going, as well as us not having access to the development team to clarify any details. In addition our lack of experience with game architecture, particularly in a practical sense would limit our understanding of rationale for dependencies and component organization meaning that for example while we believe some dependencies are illogical, a higher knowledge of game design could change our opinion.

Lessons Learned

- It is easier to use a program to determine dependencies than it is to look through source code to determine dependencies
- Few or many unexpected dependencies may be discovered after analyzing the concrete architecture of a program
- Folder organization of files doesn't necessarily reflect how components are organized

Conclusion

In conclusion, after learning how to use Understand to derive the concrete architecture we decided the style was still object oriented layered. However we had to re-update our conceptual architecture based on our findings, moving assets to Core Systems and taking out the platform independence layer. There were a few unexpected illogical dependencies but most were logical. The concurrency of the game follower the same triple-threaded multi-threading model that was discussed in our conceptual report.