DOOM 3 BFG CONCEPTUAL ARCHITECTURE

Team: Basement Gurus

Isaac Chan
Daniel Elmer
Matthew Sherar
Yohanna Gadelrab
Dylan Liu
Kainoa Lloyd

Table of Contents

Abstract

A conceptual architecture for Doom 3 BFG Edition was developed using the architecture example found in the Game Engine Architecture textbook as reference. Doom 3 BFG Edition combines two styles of architectures: the object-oriented architecture style and the layered architecture style.

The Doom 3 BFG Edition is a remaster version of the original Doom 3 improving on many aspects of the overall system and components of the game. These components contain changes to the core system, renderer and assets to name a few. Significant upgrades are also seen with the support of online multiplayer and a platform independent layer.

There are many resources online describing the architecture of Doom 3 as well as multiple source code reviews. The source code for Doom 3 can also be found online if further analysis of the system is sought out.

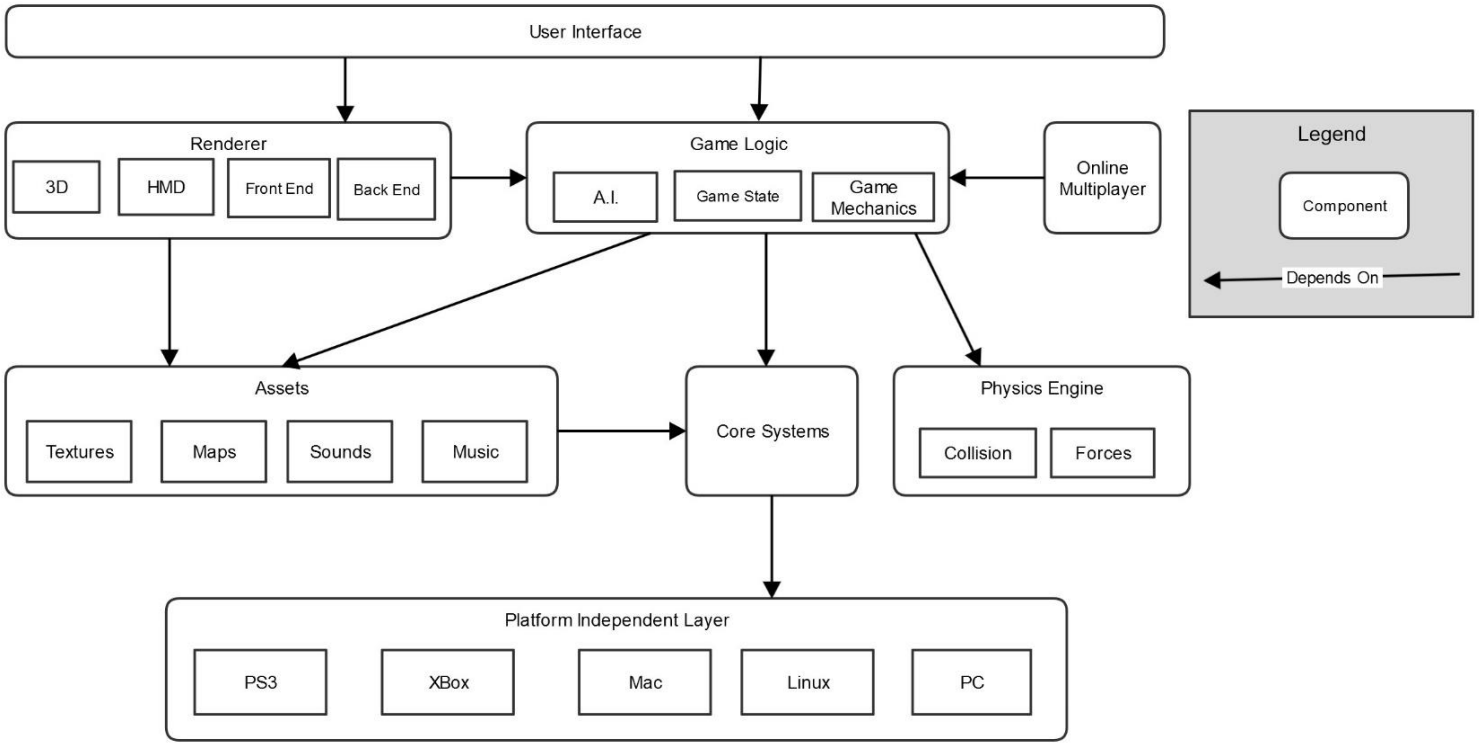
Introduction

Doom 3 BFG Edition is a science fiction survival horror first-person shooter video game developed by id Software and published by Bethesda Softworks. Doom 3 has come a long way since its original release back in 2004. Coming from a game developed solely for the windows PC to a game that can now run on multiple modern operating systems and consoles as well as the improved assets and multiplayer support seen in the latest iteration.

This report gives an overview of how the proposed conceptual architecture was developed, detailed descriptions of the enhanced components and systems within the BFG edition as well as the lessons learned from developing the architecture.

The architecture style used is an object oriented layered style with a client/server style used in the online multiplayer network architecture of Doom 3 BFG Edition. Further in the report there will be demonstrations using use case diagrams and sequence diagrams depicting how overall system and components interact with one another.

Conceptual Architecture



The overall architectural style that was chosen for this conceptual architecture was a **layered object-oriented style**. Which means that the components are layered in a hierarchical fashion where each higher layer is a higher level of abstraction. Components on the top are dependent on components in the layer below. The object oriented part of the architecture is having more than more than one component per layer.

This style is beneficial for multiple reasons, first of which is it allows for the division of responsibilities each component is separate and can be worked solely by understanding the dependencies and overall architecture rather than how each component works at its core. Separate components means that **testability** will be

smoother because each component can be for the most part tested independently of one another. This also leads to better **maintainability** (reduced maintenance) as higher level components are not dependent on lower level components, meaning that changing a higher level component will have no effect on lower level components. An example of this is if there was a part of the User Interface that needed to be fixed, changing it will have no effect on the Game Logic, however if a game mechanic such switching guns was taken out of the Game Logic the User Interface would have to change to take out the menu to switch guns.

This architectural style has increased **reusability** as each component can be taken out, replaced and used in other systems, without having to write a whole new set of code which would be the result if there was heavy coupling or if there was no separation of components. Good **reusability** and **maintainability** in the architecture also imply easy **evolution** if we wanted to completely change the look and feel of the games **User Interface,** we can completely take out that component and replace it with a complete new one without interfering with other components. In addition if we were updating to a new version of the game with added features and gameplay, mot components and remain unaltered like the physics engine which would naturally be the same regardless of changes made to the game logic.

User Interface

The user interface includes subsystems such as the GUI and heads up display and game menu. It is the most abstract layer of the system. This is the only layer that has a direct interaction with the player. It is what the Player sees while playing the game. Interacting with elements of the User Interface will draw on the Game Logic and Renderer.

Renderer

The renderer is responsible for drawing the graphics onto the screen-either regular or 3D or Head-Mounter Display (HMD) screens, according to the logic from the Game Logic system and using data from the Assets. An example of that would be when the player changes the weapon currently used, the Game Logic would detect, send the Renderer the new weapon that needs to be rendered, the renderer would then get the weapon model/texture from the Assets and starts drawing the animation where the player is changing the weapon to new one.

The Renderer Front End generates the draw commands which is then sent to the Back End

The Renderer Back End sends the draw commands to the GPU and handles any interaction between the game system and the GPU. Those are described in more details in the concurrency section.

Game Logic

This component is essentially the “Business Logic” logic of the architecture. In here is where the **game mechanics** (rules of what can and can’t happen in the game world, how players can more), **artificial intelligence** (how Non player characters act and react)**,** and **Game state (**Where players are in the world). An example of game mechanics is, whether a player is able to attack a creature and how much damage does the attack do to the creature.

Assets

The Asses layer would contain all “data” of the system. That includes Game Saves, Maps that will be used to draw the World by the Renderer, Music and Sounds to be played during the game and Textures and models for rendering the characters and any object in the world.

Physics Engine

The physics engine mainly resolves **collisions** and controls **physics of game** and sends that information back to the Game Logic. This is essential to making sure players can’t pass through each other and objects like walls. If a player does an action like jumping the physics engine must control how high he jumps and what is the rate the gravity will take him back to the ground, these are important characters in making the game realistic to play.

Core Systems

This layer has all the functions needed to run the game. This includes File System access for loading a Game Save, Math libraries used, Memory Allocation…etc. The Core Systems then communicates with the Platform Independence Layer to call the appropriate function according to the platform the game is running on.

Note that the Core Systems is not aware of the platform, that’s the Platform Independence Layer job.

Platform Independence Layer

For a AAA game such as DOOM III BFG to be successful in the current gaming industry it is important that it is available to a wide variety of platforms, which use a range of different hardware and operating systems. Since the scale of the DOOM III BFG game engine is so large it is unrealistic to attempt to modify the entire engine to be optimised for every target platform. Instead the game engine includes a *Platform Independence Layer* which is responsible for ensuring that the engines’ behavior is the same for each target platform. To do this the platform independence layer would wrap or replace several standard C/C++ library functions, it would have methods and wrappers to handle file input and output for different file systems and would include a graphics wrapper that is able to communicate from the engine to a variety of different GPU’s, among other tasks.

Online Multiplayer

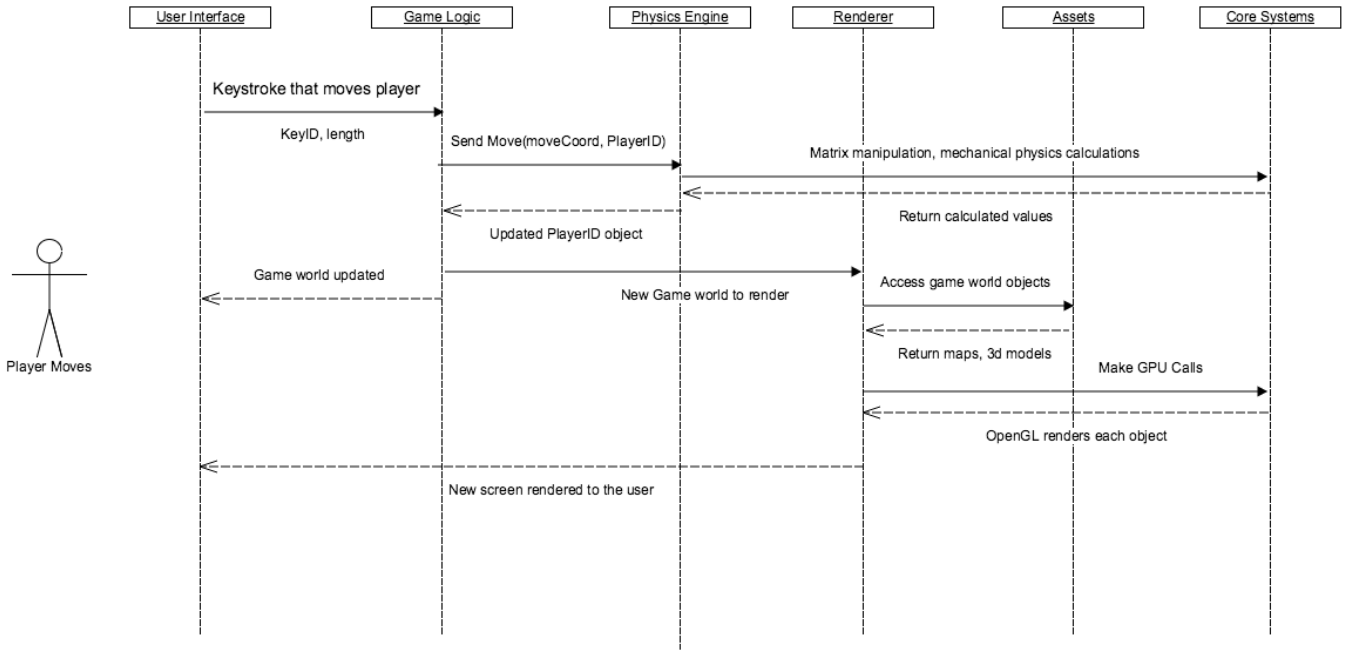
DOOM III BFG allows for the user to enter and online multiplayer mode. For this an *Online Multiplayer* component is needed. In our conceptual architecture the Online Multiplayer layer uses a Client/Server architecture model. One player in the multiplayer game acts as the server. The server interacts with the Game Logic component of the architecture to store the game state and to make all the game decisions. Each other player in the game is a client to the server. When the game logic component detects a keystroke or action from the clients’ game, it passes this to the server. The server updates the game world and returns it to the client to render. The client may only receive data from the server at a rate of around 15Hz, so in order to maintain a 60fps frame rate on the clients’ game, the clients game engine will extrapolate from the last two game world updates received from the server to render the game world until the next update is received. This helps to ensure that the performance of the game for the users is not affected by the speed of the network.

Sequence Diagram

The sequence diagram details the case of a user moving his player in the game. While the game is running there would be several threads dedicated to different tasks that are running concurrently. These threads handle tasks

that need to be checked and updated at different time intervals. For example, the renderer might need to render the screen at 60 fps while the artificial intelligence system might only need to check to see if it needs to make new decisions every tenth of a second. In this case the renderer would be checking for an updated world view from the game logic component 60 times per second, while the AI system would be checking and running less frequently. Describing these overlapping processes is beyond the scope of our conceptual architecture and better suited to the concrete architecture description. The sequence diagram shown does not show how the game engine handles commands concurrently; it is intended to show the major interactions and flow of control between key components in the system.

The user interface is always listening for any interactions that the user makes with the game. When it detects that the user presses a key that moves the player, it sends the keystroke information to the game logic component. The game logic component determines that the key that was pressed is supposed to move the player, and would determine how far to move the player based on the length of the key press. The game logic communicates with the physics engine passing the information of the player that was moved, and where it is supposed to be moved to. The physics engine would check for collisions and then update the matrices that store the information about the player and its position. To compute these transformations and physics calculations the physics engine would use the math library stored in the core systems component. Once the physics engine has finished all its simulations it returns the transformed objects to the game logic component (which would update the game world objects). The game logic component then passes the updated game world to the renderer for it to render. The renderer next needs to access a variety of game assets such as 3D models and level maps, before sending draw commands to OpenGL (which in our conceptual architecture would be encapsulated in the core systems component). Once the renderer is finished drawing the updated world, the user interface is updated with this new view.



Concurrency in DOOM 3 BFG

In computer science, **concurrency** is a property of systems in which multiple computations or functions are executed simultaneously in different streams or threads, with each of these streams having the possibility of interacting with the others. This is a stark contrast to single-threaded sequential programs, in which computations are executed sequentially one after the other in a fixed order. Such multi-threaded concurrency can be done virtually within a single computational core, but where its efficiency truly shines is when the threads (and thus the required computational load) is split across multiple cores.

When DOOM 3 was first released in 2004, many PCs still only had a single core, and as a result, the first iteration of the game was released as a single-threaded sequential program, simply because there were no other cores to take advantage of. Over the next few years however, rapid advancements in computer hardware led to the realization that a single computational core was insufficient – there was a limit to how much processing power a single core could have, and continuing to improve on a single core at that point would simply be inefficient. This naturally led to the advent of multi-core processors, precipitating a significant increase in potential processing power. The only catch now was that developers would have to actively change their code in order to harness this potential power in a multi-threaded concurrent fashion.

As a result, when DOOM 3 BFG was released in 2012, much of the source code and architecture was adapted to accommodate multi-threading. The evidence of these adaptations is immediately apparent from the point of running the executable. The DOOM3BFG.EXE process is broken down into three main threads:

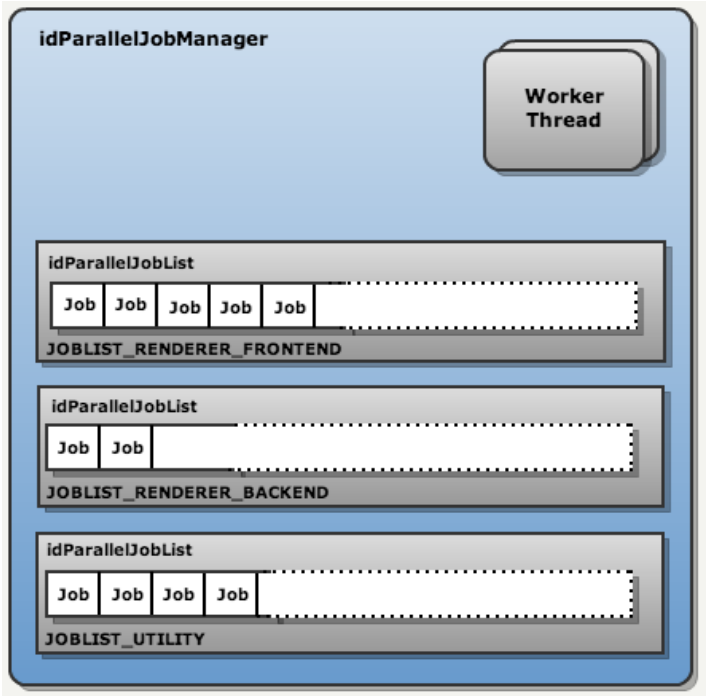
Thread dedicated to the Renderer Backend (which sends draw commands to the GPU to actually display the game environment).

Thread dedicated to Game Logic (i.e. rules of the game, AI, health/damage calculations, etc.) and the Renderer Frontend (which generates draw commands).

Thread dedicated to High Frequency Joystick input (which reads player input, e.g. movement).

Clearly, just from preliminary observations, these three threads are highly interlinked and most likely interact with each other a significant amount. To avoid desynchronization issues, interference, and other interaction problems that are common in multi-threaded programming, these three threads are also accompanied by two Worker threads (the Worked thread and the Worker thread) which perform processing tasks which are decomposed into units called "Jobs". These Worker threads remain idle until they receive a signal, after which they will attempt to fetch a Job from the master JobList. The elegancy of this system, as will be explained momentarily, is that synchronization is handled passively, simply as a result of how the system is designed, and thus the burden of maintaining synchronization is removed from active processing.

At the core of this multi-threaded architecture is the **idParallelJobManager**, which is responsible for spawning Worker threads and also for maintaining a collection of job lists called **idParallelJobLists**, one each for Renderer Frontend, Renderer Backend, and Utilities respectively. In each instance of idParallelJobList, Jobs are queued up and stored, waiting to be fetched and completed by a Worker thread. By dividing the job engine into these idParallelJobLists and limiting access to each list to a maximum of one thread, no synchronization between Jobs is required. Another significant change in the code architecture in order to accommodate multi-threading is relevant in the Renderer subsystem. Like in the original DOOM 3, the Renderer subsystem is still divided into two parts: the Renderer Frontend and Renderer Backend. However, in the BFG version, to take advantage of multi-threading, these two parts now occupy their own separate threads. Furthermore, the Renderer Frontend thread can split into further sub-threads by using the Worker system to perform tasks in parallel. Primarily, interactions

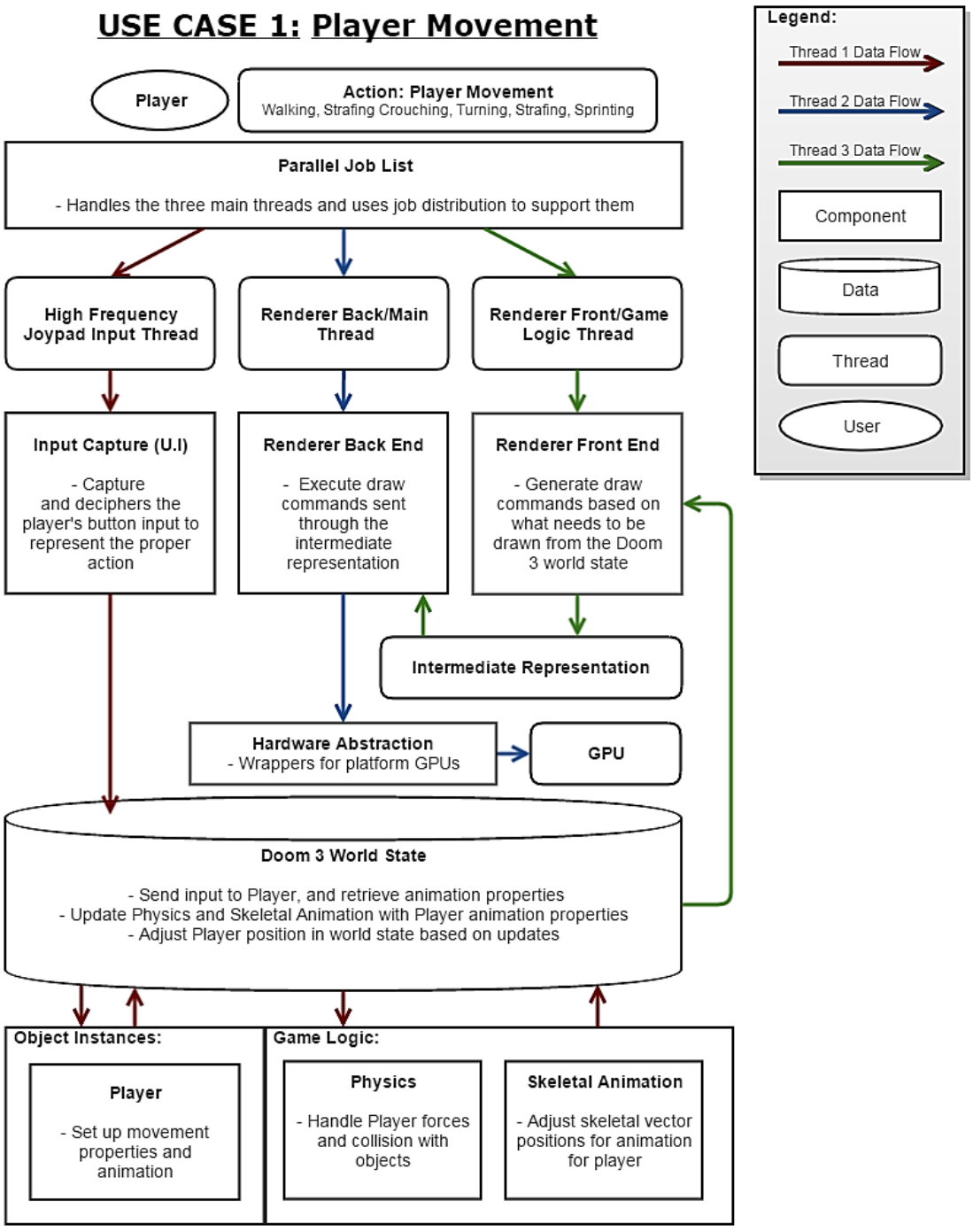


Source http://fabiensanglard.net/doom3_bfg/threading.php

detection (mainly for rendering of lighting) and shadow generation can now be performed in parallel, which partially accounts for the improvement in graphics between DOOM 3 and DOOM 3 BFG.

Use Cases

Use Case 1: Player Movement



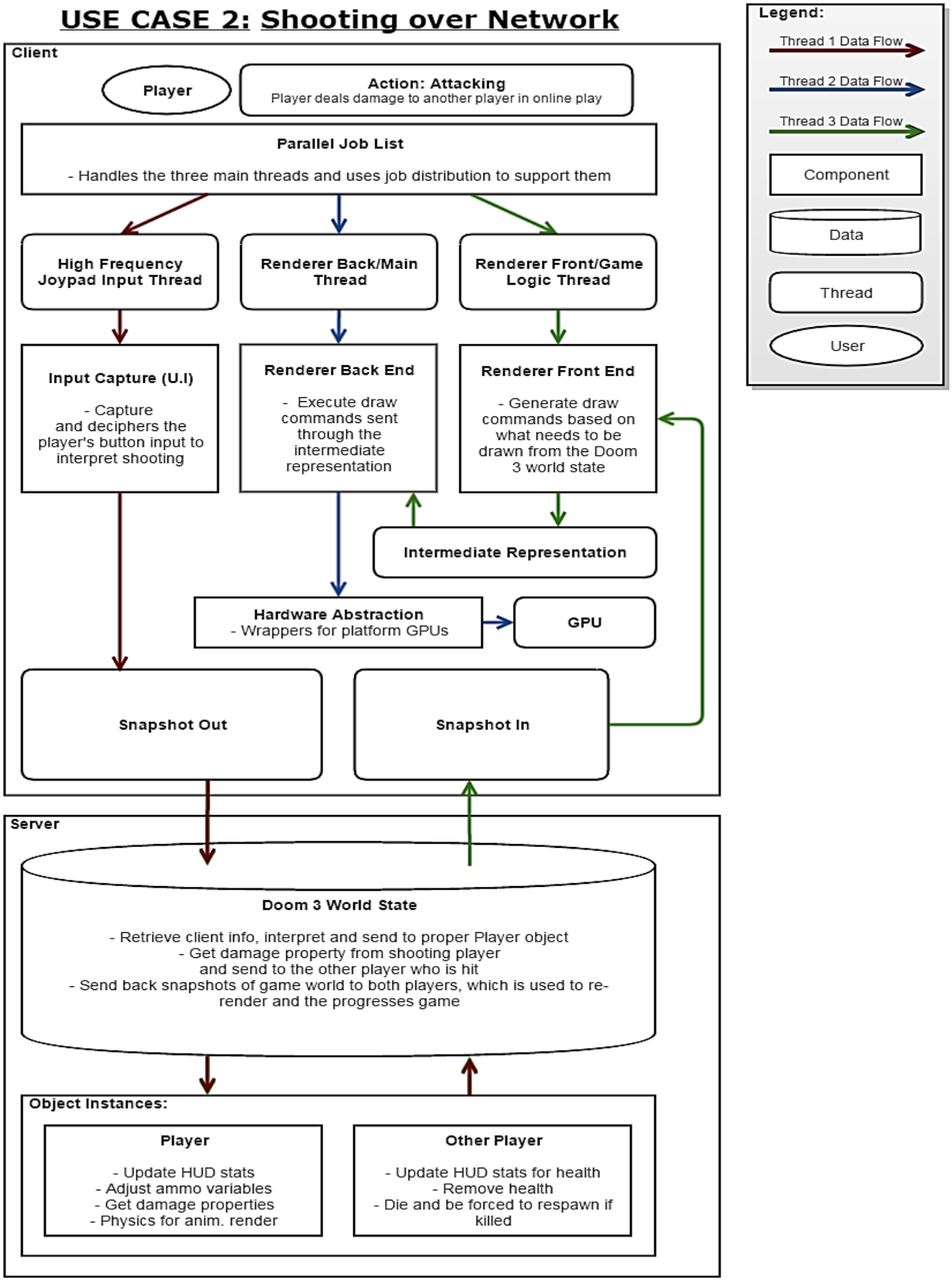
A good example of how the concurrency works in the architecture is the most basic of use cases for the game, moving around the player character. The system works through the process through the 3 threads that were specified; the two rendering threads and the high frequency input thread.

When the player presses any buttons tied to a movement, this is actively picked up by the joypad input thread where it can be interpreted and sent to the Doom 3 World state. This is then sent to the instance of Player held in the world state where, for whatever case of movement is needed, it sets up any properties for the player animation and movement required. This also influences other components that are interacting with each other through the world state, such as the physics engine and skeletal animation to properly depict the movement and animation, as well as adjust the location of the player in the world.

Now this is just one process. At the same time, rendering is done through the other threads. The renderer front end interacts with the Doom 3 world state at the same time, determining what should be drawing, accounting for any changes that have been made to it. From here, the intermediate representation created by the front end is sent to wake up the back end which is handled on the third parallel thread. The back end actually sends the commands to the GPU and draws the game state, with a hardware abstraction layer as a bridge to account for the specificities for GPUs of the platforms that Doom 3 is available for. Since a third thread can actually handle the drawing, itself the front end can continue observing the World state without getting bogged down.

Overall, the parallel work over the three threads makes the system highly cohesive for the sake of maximizing performance and efficiency.

Use Case 2: Shooting Over the Network



Another case to bring up is the play of the game over the online network; in this case we'll look at what happens when a player fires their weapon and inflicts damage on another player, and how it changes the architecture to match a larger system.

The basis is the same as movement; the player enters the command to shoot, the input is picked up by the joypad input thread where it can be interpreted and sent to the Doom 3 World state. Except during online mode, the role of the built-in game state is highly downplayed. Instead, a client-server architecture is used, and that information is bundled in a snapshot that is sent to the game server where the majority of the game processing occurs. Here, the input on the player can be interpreted and sent to the proper Player object, where ammunition is used, damage properties and trajectory is received, and animation properties are set. The trajectory is calculated and is determined to hit another player, where the Player object representing them loses health according to the damage properties from the other player's weapon and dies if they can't survive it. If this happens, the world state removes them and they need to wait for respawning. Either way, the game state creates a new snapshot to send back to the two players who were affected for rendering. While not on the diagram, the skeletal animation and texture mapping is still on the client, foregoing the need to send that complex information over the network.

Suffice to say, this approach is required to lighten on the load on each client as much as possible regarding the state of the game, but also minimizes the amount of information that is needed to be transferred between the two, taking care of the intensive process of rendering on the clients.

Conclusions

To sum up, we decided to go with a layered architecture with an Object Oriented design since that will allow easier modification to the system later on with the added abstraction of the OO design.

The layers and the grouping were determined according to their functions resulting in an architecture with minimal coupling. This also allows future upgrades to the games (newer versions, patches…etc.) and any change in the implementation to be seamless without requiring any changes in the architecture.

The Object Oriented design added an extra layer of security to help in the implementation of Anti-Cheating system for the multiplayer game.

Also, Multithreading is one of major aspects of the BFG version compared to the 2004 version that drastically changed the architecture and implementation of the BFG version.

Lessons Learned

- Grouping similar components together like we did in the Assets layer make the architecture more cohesive.
- Reference architectures are very helpful and can save a great deal of time in deriving an architecture for a new system.

References

1. Game Engine Architecture, By Jason Gregory
2. DOOM3 BFG SOURCE CODE REVIEW