Presentation Notes

Intro

While these games have quite different aesthetics, they both share a significant amount of code. They are both built on id Software's idTech3 engine. id's earlier engines were widely licensed and formed the basis for other companies' games, as well as other engines, most notably Valve's GoldSrc (the engine that powered Half Life 1). idTech3 had more competitors. Other engines, like Unreal and Source, ultimately had more games built on them, but idTech3 was behind several influential or critically acclaimed games, which shaped the next decade of First Person Shooter's.

John Carmack, the CTO of id Software and the primary developer on all of their games, is an important figure in the history of the shooter. His technical preferences and engineering decisions have shaped the entire genre. While the slower, more realistic style of Valve and others won over id's fast twitch, arcade shooter, a complete picture of the genre must take id's games, and their engines, into account. idTech3 is particularly interesting for me, as it is a turning point where id Software began to retreat from mainstream popularity. As other companies began developing competing visions of what a shooter could be, id's *Quake III Arena* continued pursuing extremely fast paced, cartoon-like gameplay, without a single player story mode.

idTech3 is also an interesting example of what I will call "immaterial platforms", software platforms that shape and constrain further development in the same way that game consoles do. The game engine is the key immaterial platform for computer games. Today, I'll closely examine idTech3's networking component on several levels. While there are many interesting parts of the engine, this tight focus allows me to trace its impact on several key communities, and is a lense to view the broader place of id Software in the genre.

Immaterial Platforms

From the very beginning, Bogost and Montfort include software platforms in the scope of inquiry. This is their definition from 2007.

However, the first books to come out of the MIT book series have all focused on physical, commercial game systems. While platform studies is not constrained exclusively to this book series, it is the most significant outlet for platform studies work. Dale Leorke has criticized the series for becoming formulaic, and prompted platform studies scholars to expand their methodology and scope (2012).

Following Friedrich Kittler's observations on software obscuring or abstracting away hardware concerns, immaterial software platforms are those which hide the operations of a physical, material platform which does the real computational work. Many games are not tied to a single hardware platform, but are instead targeted to a general software platform, which can then be executed on a variety of physical platforms, with as little modification as possible. The daily engagement of game developers with these software platforms brings them closer to the surface of player experience, and their generality makes them a worthwhile platform to study.

Immaterial platforms occupy an inbetween space in this five layer stack. They are a platform for further development, but are made up of code. It is important to make the distinction between game code, which encodes the form of the game, and the infrastructural platform code that underlies it. This is a messy, liminal space that is not clearly defined. If we place immaterial platforms as a halfway point between "code" and "platform" they become a useful lense with which to investigate the intersection of those layers.

Game Engine as Platform

The most important immaterial platform for any game is its engine. They emerged as a way to reuse code between games which shared many of the same core requirements. 2D or 3D graphics, networking, sound, and realistic physics simulations are typically easier to reuse than reimplement, as is interacting with hardware.

A game engine can help mitigate the challenges of developing for multiple physical computing platforms. Using an engine allows the developer to program to that engine, treating it as the primary platform, rather than the computing system which runs both the engine and game. To the developers, the engine can be a more important substrate than the computer hardware.

Andrew Hutchinson has called working inside of these technological limits the “pragmatic expression” of games. In a paper contrasting the aesthetics of two 1993 games, *Doom* and *Myst,* Hutchinson notes that both games had similar goals, but, due to the technological limits of their time, both were forced to make compromises. The engines of these games were specifically tuned for two radically different aesthetics. It is impossible to make a game as visually immersive as *Myst* on the *Doom* engine, just as it is impossible to make a fast paced game on the *Myst* engine. The engine is not just important for developer experience, it determines what they can possibly create for the player.

History of idTech3

The early history of id Software is documented in the nonacademic history book *Masters of Doom* (David Kushner, 2003). In addition, John Carmack was very open during this time with many of the technical details of day to day development. He posted regularly to a “.plan” file, which other programmers could follow to receive his updates. Using these two resources, combined with interviews and the code itself we can learn a great deal about the development history of the idTech3 engine.

Kushner frames his story in terms of “the two Johns”, Carmack and Romero. In his book, Carmack was the hyper focused technical wizard, while Romero was the free-wheeling design genius. Together they invented the First Person Shooter, but were left behind as other companies took the genre to new heights creatively and commercially. The development of idTech3 was intrinsically tied into the development of *Quake III Arena* (id Software, 1999). This took place after Romero had left id to start his own company, Ion Storm. The game abandoned any pretense of narrative, and was heavily focused on competitive multiplayer. The engine’s architecture was oriented towards this goal, particularly the networking.

Protocol in idTech3

In a popular technical code review of the idTech3 engine, coder Fabien Sanglard declared that “the network model of Quake3 is with no doubt the most elegant part of the engine.” (2012). This pursuit of engineering elegance and excellence, of doing “The Right Thing”, is documented in *Masters of Doom* as one of Carmack’s driving goals.

Network communication happens over multiple layers of mutually agreed upon protocols, which exist simultaneously. In his book *Protocol: How Control Exists after Decentralization* (2004) Alexander Galloway thoroughly develops the concept of “protocol” as key to understanding control in a networked society. With his concept of protocological control in mind, the centralized model of idTech3, which I am about to describe, takes on new connotations. Galloway politicized protocol.

Quake III Arena has a client-server model, which allows all of the game state to be handled by a central game server, while thinner clients primarily handle graphics presentation and relay player input to the server. This provides anti-cheat regulation to server owners, allows them to determine which mods can be run, and generally subordinates the players' clients.

The Quake application level protocol is called NetChannel, and it is tailored for deathmatch. The server sends unreliable packets (Hook, 2006), meaning that it continues to send the data without waiting to confirm that the client has received it. The contents of each packet are the compressed state values that the client will need to render its screen, and a sequence value to keep the packets in order. The server sends the differences between the current state and the last state which the client acknowledged receipt of. In the event of data loss, both the server and the client can make predictions as to what the next frame of the game will be based on the current state.

This prediction allows the client to continue rendering the frame, but can result in jerkiness when the network connection is re-established and the client updates itself to match up with the server. The differences between the clients earlier prediction and the server's "canonical" state is the the classic visual lagging that anyone who has played networked shooters has experienced.

The client-server model of idTech3 was designed specifically for the fast paced deathmatch genre, which id pioneered and *Quake III Arena* perfected. This architecture works very well for this type of game, but proves challenging for creating single player experiences. I expand on the implications this had for modders and licensees in the section "Protocological Impact".

Networking Implementation

Galloway observes that the regulation provided by protocol must “always operate at the level of coding”. In the case of idTech3, the code governing this protocol has been opensourced, and is available for study. Using analytical tools from Mark Marino’s Critical Code Studies (Marino, 2006) we can unpack the implementation of the protocol.

The project is large, with approximately 1,110 unique files, 241,000 lines of C, 1,100 lines of assembly and small amounts of a few other languages. Diving into a large software project can be overwhelming, particularly if you are unfamiliar with the domain. I had no prior experience with game engines, and am indebted to Fabien Sanglard’s “Quake 3 Source Code Review” series (Sanglard, 2012). It outlines the code’s high level architecture, with some details of each section.

In general the code did not yield to code studies well. It turns out that infrastructural code is fairly dry. Game code seems much more fruitful. This code is terse, but confusing parts are commented, and everything seems to follow a similar style. In short, it is fairly boring. However, I’ll share one brief selection.

In code/qcommon/msg.c: Here we can see the player_state variables that are passed between the server and the client. This is potentially useful information for a developer, but from a code studies point of view, what interests me is the comment and small helper function at line 1098. Its an example of code written for humans rather than machines. It is perhaps less performant, but much more readable. The famous introductory computer science textbook, Structure and Interpretation of Computer Programs states in its introduction that “programs must be written for people to read, and only incidentally for machines to execute." (Abelson, et al). This is the essence of code studies. This code is terse, infrastructural library code, which is typically abstracted even from other developers. Still, even here we find traces of the human authors'.

Materiality of the Network

Networking code provides an interesting lense to view the fickle nature of physical hardware, as well as the challenges of protocological implementation details which may not fully abstract away the network's materiality. Transporting bits around the world forces programmers to recognize the physical limits of computing, and the gaps between protocol and reality.

An incident from the post-release support of *Quake II* (id Software, 1997) is illustrative of this phenomenon. In a .plan entry that Carmack posted at 4:40 AM, December 31, 1997, he explains a complex and obscure bug caused by a players' router mishandling the less common UDP datastream. The router was making assumptions that the data would be TCP, or acknowledged UDP.

Carmack solved the problem with a few extra bits per packet, but it irked him.

This incident is fascinating for several reasons. The first is the unhealthy and fanatical work ethic he had at this time. Carmack was working extremely late on New Years, solving a bug that only affected a few users.

The other issue it illustrates is the difference between the abstract, interoperable protocol and the messy reality of physical routers. Carmack's application protocol should function on all routers if they properly implement the UDP standard. However, certain routers' optimizations or engineering tradeoffs caused his unusual approach to break. This disconnect between the clean, logical, abstract protocol and the stubborn materiality of computing is important to understanding computing in general.

Protocological Impact

This client/server split allowed clean code, and was an appealingly simple architecture. However, it caused problems for licensees and modders. Carmack prioritized his needs and desire to do the engineering “Right Thing” over the needs of other developers.

In a .plan entry from November 3, 1998 Carmack explains some difficulties of his client/server split:

The security challenge is that game mods would now execute code on the client, rather than just the server. This meant that everyone, not just server owners would need to assess the safety of a mod. Similarly, with client side mod code, “less popular systems would find that they could not connect to new servers because the mod code hadn’t been ported.” His solution was the interpreted C virtual machine. This is an important technical decision, and a significant amount of engineering effort, all to support game mods.

When I started this investigation I was following up on an idea I had seen repeated many times in fan forums for id games. id’s games are just tech demos for their engines, and the games themselves are secondary. This is used as an excuse for their lack of design innovation. However, after a deep dive into this engine I found that to be simply not true. All of the engineering decisions made in the networking module are to support the optimal Quake III Arena experience. It is not a flexible architecture, and makes single player experiences harder to craft.

This quote strikes at the core of id’s fall from prominence, and the changing direction of the genre in later years. They stopped focussing on licensing their technology, and allowed other players like Valve and Epic to take over the market.