Implications of Fog Computing on Multiplayer Games

Background

Interest in VR

When the Oculus Rift prototype was first released in 2012[5], it opened up an entirely new market: affordable consumer VR. Four years later, the introduction of the HTC Vive made “room scale” experiences, where the player can walk, crawl, and jump in the three dimensions of reality to move themselves in the three dimensions of their virtual space, possible[6]. The technology that drives the Vive’s room scale innovation are their “lighthouse” infrared cameras which track the headset and two controllers in three dimensions.

Unique Properties of VR

Given the recent introduction of VR and room scale VR, developers face new and greater performance challenges compared to traditional, two dimensional platforms. Two such challenges are the higher frame rate requirements of VR applications and the orders of magnitude increase in position updates due to the high fidelity tracking of the headset and controllers made possible by HTC’s lighthouse technology.

Traditional platform games require between 30 to 60 frames per second (fps) to be considered performant. Below that, the game may seem laggy and unplayable. In VR applications, the requirement is much higher: 90 fps at a minimum. The costs of dipping below 90 fps are also higher in VR. Because the display covers your entire field of vision, dropped frames may result in serious nausea and a ruined immersive experience. Beyond just being frustrated, the consumer will get sick.

Because of the high fidelity tracking of the headset and controllers in room scale VR, position updates happen every frame, for three objects, in three dimensions. Compare that with traditional two dimensional games where the player has two joysticks, each of which can move in only two dimensions, and do not send any updates at all when the player is not touching the controller. More positional data is generated in VR applications for one player than would be generated for several players in traditional games.

Why VR and not AR?

In this study, we will consider only VR applications for a few reasons. First, VR technology is currently more mature than AR technology. Second, many of the design and performance improvements that are made in the VR space are also directly applicable in the AR space without modification. For example, treating menus as physical objects like tablets instead of floating 2D menus like they would be in traditional platform games. Similarly with the performance implications of VR that are being studied here. Third, VR is relatively less expensive than AR. To see this, you can compare the cost of the HTC Vive ($800) to that of the HoloLens (~$3000).

Constraints

As briefly mentioned above, there are a number of performance constraints required by VR applications. In this study, we will consider the following constraints:

- We are studying a networked application, so it must be able to support multiple players in a world with shared state
- The application must be able to maintain over 90 fps over the life of the game
- The application must be able to guarantee 90 fps for a to-be-determined number of players

Problem

Using the constraints above, the following problem statement can be formulated:

What real time and frames per second guarantees can be made for a multiplayer VR application using a novel peer to peer model based on state sharding?

Test Application

In order to test and measure a library that will be created to solve the above problem, a test application must be created that has qualities of a real game, but is simple enough to be built quickly. Our test application will be an environment created in Unity which has a cube, a sphere, and potentially other objects which can be picked up and thrown around by VR players. This environment can be viewed with either a VR headset or a computer, although only the VR headset will be able to fully interact with the environment. In this application, the VR player can pick up and throw items like the square and the cube using his or her controllers, and the computer player can simulate “catching” the item by pressing a certain key. If there are multiple VR players, they should be able to throw and catch the objects with each other over the network.

Proposal

In order to solve the stated problem, a library will be created that will support multiple players within a single three dimensional, interactive environment. Both objects within this environment and actions which players execute on the environment must be replicated on every player's world instance.

Existing Solutions

There are three primary pre-existing solutions to this problem:

Client-server model

In the client server model, clients send their "actions" to the server, which reconciles those actions with the actions of other players and distributes the updated world state back to the clients. In this approach, each client can interpolate their actions so as to reduce perceived lag. The client-server model is less susceptible to cheating than the peer-to-peer model.

Peer-to-peer model

In the peer-to-peer model, clients send their "actions" to each other rather than to a central server. Each client then executes the action that they received on their own machine. At varying intervals, the state on all the clients must be reconciled so that drift does not occur. Existing literature[2] uses the peer-to-peer model to dynamically partition game state and only send players updates that are relevant to their area of interest (AOI). State is maintained among peers with a distributed hash table a Voronai diagram for game world decomposition. Objects within the world aren't necessarily relocated to different peers when they move in the game world because the distributed hash table provides a reliable way of finding objects on any peer. The benefits of such an approach are realized most in MMORPGs where game worlds are very large, players are numerous, and it would be impossible to retain all of that state in a single server or client.

Two primary challenges of the peer-to-peer model are how to catch cheaters and how to reduce network load. To catch cheating, the New Event Ordering (NEO) method can be used. This approach slows down the speed at which peers can talk to each other, but is guaranteed to address the five most common protocol level cheats[1]. See the "Cheating" section for more details.

Client-server with distributed servers

Instead of using one central server and scaling it with hardware as demand increases, this approach uses several central servers which communicate both with each other and with the clients. World state is partitioned among all servers and pieces of world state are uniquely identifiable by GUIDs. Each piece of world state is considered either a primary or a secondary copy. Primary copies—of which there is exactly one per piece of world state—are the “source of truth”, and all actions are executed directly on them. To improve performance, secondary copies are made on servers that are located closer to the clients they serve. These secondary copies can be used for reads only—any actions that they receive are forwarded to the primary copy that they inherited their state from. After the primary copy executes the action, it sends a delta encoded update to all of its replicas. If no updates arrive over a given period of time for a secondary copy, it can safely be deleted. Execution partitioning is used to execute think functions for primary copies of objects owned by a node[3].

Our Solution: Peer-to-peer with state sharding

We propose an approach which combines the merits of the peer-to-peer and distributed server approaches enumerated above. This approach is strictly peer-to-peer—it does not require any central servers to be administered for the game—but uses state sharding concepts to improve peer performance and decrease network traffic.

In this approach, primary copies of the state are distributed among the peers such that each peer owns a portion of the world state. Then, secondary copies of primary objects are created as necessary to decrease load. These secondary copies are to be created spatially such that peers which access a particular state object frequently can access a version that is closest to them.

Similarly to the distributed servers approach, secondary copies forward actions back to the primary copy for execution, and the primary copy sends delta-encoded updates to all of its replicas. Here we require that every primary copy have at least one associated secondary copy for redundancy. If a peer disconnects, one secondary copy for each of its primary copies will be chosen to become the new primary copy.

When sharding data, research into the “TrueTime” API for Google Spanner[9] can be used to consider how updates to shards can maintain consistent timestamps across many peers. While syncing to atomic or GPS clocks is out of the scope of this project, it is not unreasonable to expect that such authoritative sources could be used to limit the scope of time based errors during sharding.

Separate Proposal: Performance Placement

As a separate part of this project, Shashank will research secondary copy prediction and location optimization.

Cheating

For our approach to be viable, it must not regress on cheating guarantees made by existing peer-to-peer solutions. While we will not implement any cheating protections, we will consider different kinds of cheats and ensure that our solution can address them using the NEO protocol described in prior work.

In “Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games”, five protocol level cheats are enumerated:

1. Fixed-Delay Cheat
2. Timestamp Cheat
3. Suppressed Update Cheat
4. Inconsistency Cheat
5. Collusion Cheat

The NEO protocol divides time into rounds which have strict bounds on the maximum latency that can occur from one player to the majority of other players. As with my application, the protocol works for a small subset of players that exist within one AOI. Each round, all of the players attempt to send their update packets to all other players. An update packet contains an encrypted, timestamped update, and a key to their previous round’s update. Then, the players vote as a group on how many packets they received from the other players using bit vectors. This vote provides proof that players sent their updates on time; however, like with Bitcoin, the cheat can still occur if a majority of players collude to cheat. Using the NEO protocol, all five of these protocol-level cheats are addressed with only a *2 * round length* performance impact when a pipelined approach is used.

Another way to detect cheating is by using the approach used by bitcoin and Ethereum—a blockchain[7]. Each game would begin with a genesis block, and game state transformations would be represented as a blockchain. Each time a player performs an action, they would create a new transaction with a SHA256 hash and broadcast it to the other players. The other players can work to verify a set of transactions by collecting several of them in a block and varying a nonce until they find a solution to a problem. After the block has been mined, it is broadcast to all other players and added to their respective versions of the blockchain. If forks occur, they are dealt with in the same way as in Bitcoin – the longest one wins. Therefore a malicious player would need over 50% of the computing power in the game to send out two conflicting actions.

Smart contracts as Ethereum[8] would enable even more flexibility. Since ethereum provides a Turing complete language alongside its blockchain, transactions that contain logic can be considered “trusted” by two parties and the “agreement” that those parties make can be enforced autonomously. While it would be practically difficult given the unique “Solidity” language baked into Ethereum, player actions could potentially be written directly as code as part of a smart contract which would be added to the blockchain and mined by other players. Over time, the blockchain would contain a record of every action that was performed throughout the game. Since forks would be detected using the same blockchain concepts as Bitcoin, a player could not send two differing actions to two of its peers.

Simplifications

In the interest of time, the following simplifications will be made: First, the MMORPG case will not be studied and therefore only one Area of Interest (AOI) will be used. Additionally, given the lack of hardware and time, this study will focus on incremental scalability from one to eight players. In doing so, we hope to answer questions such as, “How is performance impacted as each new player joins.”

Strategy

Implementation

To measure our hypothesis, we will create a library for multiplayer games that can be used in any game created with the Unity engine without modification. This library will follow the architectural guidelines set out in the “Matrix: Adaptive Middleware for Distributed Multiplayer Games” paper[4] to keep a clean separation between networking and application logic.

Hardware and Software Components

The physical components used in this project are the HTC Vive, Windows or Macintosh computers, ZeroMQ for messaging, the Unity engine for creating the environment, and the C# language for creating the shared networking library. Because of the requirement to use the Unity engine, we must use C# with .Net 3.5.

When developing, a single computer can be used which spawns multiple instances of the application which speak to each other over the local network. However, the final implementation should demonstrate several computers on different networks communicating with each other over the internet so that realistic lag is taken into account. Since we may only have one or two HTC Vive’s to work with, the other “players” in the game can be computers which are viewing the game with a normal desktop display. They will receive the same updates as the HTC Vive, but will have more limited ways to interact with the world.

Networking Library Architecture

For reusability purposes, it is important that the networking library be isolated from any application code. Here it is helpful to follow the guide set out by the Colyseus architecture[3]. In this architecture, there are 5 components:

1. Game application code
2. Local object store
3. Object locator
4. Replica manager

Each peer will have *game application code* which defines the rules of the game. When actions occur, they can be interpolated using this game application code before the real state of the object is retrieved from an authoritative source.

As mentioned previously, each node will contain a set of primary and secondary replicas of local objects. These will be held in the *local object store*. Each primary replica will have a unique identifier, while each secondary replica will have a pointer to the primary replica that it was created from.

An *object locator* is maintained on each node. When given a key for an object, the object locator can speak to other nodes and determine where that object lives. To do so quickly, a distributed hash table can be queried which will indicate which node owns the replica that is being searched for. In the interest of time, Redis may need to be used in place of a distributed hash table. The locator can also speak to the replica manager in the case that a found object must be replicated. Finally, the object locator can use range based queries to find objects that its primary copies depend upon (this last item may be out of the scope of this project).

A *replica manager* updates the secondary copies owned by this node with their authoritative primary copy sources. It keeps track of which nodes own the secondary copies that it maintains and speaks directly with those nodes to update its replicas.

Measurements

The peer-to-peer sharding approach described previously will be measured for success on two verticals: First, the system usage of each Unity client, and second, the amount of network traffic between nodes in the system. These two verticals will be measured across a second axis: incremental scalability as each new player is added to the system.

To measure the performance of each Unity client, the built-in Unity profiler will be used. It can profile CPU/GPU usage, memory, and frame rate while the application is running with minimal cost to performance. For measuring the amount of network traffic between nodes, Wireshark will

be used for packet sniffing. In all of the following performance bounds, we consider a game which has eight or less players as an arbitrary bound.

When measuring frame rate performance, we will require a strict lower bound of 90fps on a computer with an Nvidia 1070 or 1080 GPU. As mentioned previously, going below this number can induce nausea.

CPU and GPU usage can be evaluated by comparing it against running the same application in “single player mode” – i.e. with no networking component enabled. We hypothesize that CPU and GPU usage in a networked game can be kept at less than 30% above that of a single player game using our methods.

Finally, in measuring network traffic, we expect the amount of traffic per player to increase sub-linearly. That is, the amount of traffic that one player experiences when playing with seven other players should be less than the amount between two players times seven.

(amount of traffic between one player and the other seven) < (amount of traffic between two players) * 7

Our goal is that each incremental player should only increase traffic for the first player by 30%.

References

1. GauthierDickey, Chris, et al. "Low latency and cheat-proof event ordering for peer-to-peer games." Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video. ACM, 2004.
2. Buyukkaya, Eliya, Maha Abdallah, and Romain Cavagna. "VoroGame: a hybrid P2P architecture for massively multiplayer games." Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE. Ieee, 2009.
3. Bharambe, Ashwin R., Jeff Pang, and Srinivasan Seshan. A distributed architecture for interactive multiplayer games. School of Computer Science, Carnegie Mellon University, 2005.
4. Balan R.K., Ebling M., Castro P., Misra A. (2005) Matrix: Adaptive Middleware for Distributed Multiplayer Games. In: Alonso G. (eds) Middleware 2005. Middleware 2005. Lecture Notes in Computer Science, vol 3790. Springer, Berlin, Heidelberg.
5. "Oculus Rift: Step Into the Game". Kickstarter. Retrieved June 17, 2015.
6. "Valve's VR headset is called the Vive and it's made by HTC". The Verge. Retrieved 1 March 2015.
7. Tschorsch, Florian, and Björn Scheuermann. "Bitcoin and beyond: A technical survey on decentralized digital currencies." IEEE Communications Surveys & Tutorials 18.3 (2015): 2084-2123.
8. Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." Ethereum Project Yellow Paper 151 (2014).
9. Corbett, James C., et al. "Spanner: Google's globally distributed database." ACM Transactions on Computer Systems (TOCS) 31.3 (2013): 8.