An Event-Driven Distributed Network for Real-Time Remote-Rendering

Michael Wong and Taylor Beebe
Williams College

Abstract

Outsourcing graphics computation to the cloud for use in real-time applications like video games is becoming a viable alternative to purchasing the expensive hardware required to perform high-quality renders. Many modern implementations of real-time computation outsourcing uses a client-server model which is highly susceptible to network and server failure.

This paper outlines the design and implementation of an event-driven distributed network alternative to the aforementioned client-server model. We seek to amortize the rendering of a client's screen by utilizing multiple networked systems which render predetermined sections of the screen and stream the result back to the client. In addition, we propose several methods to mitigate network and server failure. Our results were promising and suggested that increasing the number of rendering machines, and thereby decreasing the amount of work each must do, can greatly increase framerate.

1 Introduction

3D graphics is a rich area of research, driven largely by consumer demand for constant improvement in visual fidelity in computer applications, animated films, and video games. Rendering in *real-time* means to produce images at least as fast as the human eye's refresh rate giving the illusion of motion.

Real-time rendering is computationally expensive and requires specialized hardware to produce visuals at industry standard quality. *Remote-rendering* is a promising solution (for the consumer) which offloads expensive graphics computations to servers in *the cloud*. For the purposes of this paper, we are primarily interested in remotely-rendered interactive video games.

In real-time remote-rendered gaming, keyboard/controller input is collected by a client device and streamed to a server. The server then renders an image and streams it to the client. In the industry, this is referred to as *cloud gaming*. With low enough network latency, such a system gives the illusion that the application is running solely on the client's hardware by making input lag imperceptible.

There is enormous potential for these services to remove the hardware barrier in gaming, but network reliability has always been a major hurddle to commercial viability. Our project aims to expand the cloud gaming model. We seek to increase the fault tolerance of cloud gaming systems and take advantage of the client CPU. Our solution outsources computation from a client device to multiple nodes on a distributed network in order to make parallel the rendering of each frame that will be streamed back to the client. The time to render a single frame is proportional to the size of the client's screen. In theory, our approach will be feasible if the time to receive and combine multiple sections of a screen into a single frame scales slower than the time gained from shrinking the amount of work done by each node through adding more nodes.

The success of such a system would imply that outsourcing high-fidelity rendering to an array of machines with mid-tier graphics hardware can approximate or even surpass the capabilities of a single machine with top-tier hardware. In essence, a divide-and-conquer approach would allow expensive applications to run on re-purposed machines with very modest specifications.

In addition, the event-driven design of our architecture is fault tolerant. We take advantage of the client to do low-quality renders in the event of network trouble. Having multiple network nodes also allows flexible network configurability. In the event of any node failure, the network can do ad hoc reconfigurations to redistribute work across nodes.

We implement a proof-of-concept using a custom distributed networking layer built for the G3D innovation engine[1]. When running a simple game on our network, we observed promising performance improvements as the number of nodes was increased. We are confident that with optimizations and additional testing, our architecture will offer an interesting alternative to current cloud gaming systems.

2 Related Work

Cloud gaming has existed for the past decade, but early attempts were commercially unsuccessful. A notable attempt was OnLive[2] which offered cloud gaming through a specialized piece of hardware which connected to the OnLive servers. Unfortunately, OnLive was plagued with inconsistent video quality, high input lag, and an unenthusiastic market which was not ready to adopt a new way of playing games. The poor reception caused the company to discontinue their service after just five years. Since then, advancements in computer networking have caused a resurgence of interest in cloud gaming with new services being intruduced such as Google's Stadia[3], Nvidia's GeForce NOW[4], and Microsoft's xCloud[5].

A few interesting papers outsource graphics computations in different ways[6][7]. One such study is *An Architecture for Java-Based Real-Time Distributed Visualization*[8]. Researchers in the study used a method very similar to ours where images were rendered using an array of machines called *servers*, each of which renders a predetermined portion of the screen. Their network was star-shaped meaning every server directly interfaces with the *client*, the machine which actually displays the image and accepts user input. The client holds the master copy of entity data and sends out updates to servers at fixed intervals. Despite Java not being designed for rendering, testing of the system yielded promising results. However, researchers conceded that networking and rendering techniques at the time of writing were inadequate for distributed rendering of high-resolution, full-screen applications.

CloudLight[9] by Crassin et al. implements three different lighting algorithms that are run separately on a client-server model. Their three methods *Voxels*, *Irradiance Maps*, and *Photons* all place the network differently in the pipeline, *Voxels* is most similar to our approach. In *Voxels*, a server can calculate view-independent illumination data and a separate GPU will render a frame from a given perspective for a client and stream it to that client with H.264. Note that their system is built and optimized for ray-tracing and multiple clients. Their approach was able to serve a consistent 30 frames per second to the client.

3 Method

For some interactive 3D program X running on a client device, our architecture uses a distributed network of *n remote* nodes to render and stream single frames to a user-controlled device – the *client* – for display. For example, if $n = 1$, then only a single machine is rendering the entire image which is a scenario similar to cloud gaming techniques used by GeForce NOW, Stadia, and xCloud.

In addition to the client and remote nodes, the network has a *router* node which facilitates communication between the client and remote nodes, offering a convenient layer of abstraction. Our method differs from previous approaches by making extensive use of the client CPU and only using off-device resources to perform renders. However we seek to improve two aspects with our method: 1) lack of use of the client computing power, and 2) network fault tolerance.

The event-driven approach does not refer to user input events, but refers to the fact that idle is the default state of the network. This design choice is to best leverage the computing power of our client device. We collect user keyboard/controller event data and simulate X's 3D environment on the client's CPU. Whenever the client wants to display a frame, it broadcasts an *application state* update to the router which decides how to distribute the work among the available remote nodes. Every remote node runs a modified instance of X while sitting idle and listening for instructions on a separate thread. When it receives an application state update, it renders a predetermined section of the screen and sends it back to the router. We chose to split the screen in rows from top to bottom to take advantage of locality when copying to a buffer. When the router receives all sections (*fragments*), they are stitched together and sent to the client which displays the image to the user. In an alternative design, remote nodes can bypass the router and send their screen sections directly to the client. This design remains future work.

The number of remote nodes is at least one, and theoretically bounded above only by the height of the resolution, more specifically the number of rows of pixels. However, there would be a significant amount of overhead in having hundreds of remote nodes, thus there must exist an optimal n.

While the network renders a frame, the client machine renders a low-quality image in case it does not receive a frame in a reasonable amount of time, allowing an added degree of fault tolerance. If a failure occurs in an off-device render, we can simply display the low-quality render of the scene. The most likely scenario is one of the remote nodes crashes, in which case the router must reallocate resources, redivide the screen and reconfigure the remaining remote nodes. Our implementation does not currently support this feature.

There are obvious tradeoffs and inefficiencies worth addressing. For example, between clients and remote nodes, there is some repeated computation. The network must apply the application state update multiple times across the network, first on the client (where it is computed) and then on the remote nodes. In practice and depending on the application, these application state updates are relatively small and are not a major performance bottleneck. It would largely depend on the game - in games with many physics based objects we might expect to see the transit time of the state update packet impact the framerate. However, the largest bottleneck is the latency of sending image data. To mitigate this, our architecture enforces a set framerate for the hosted application, if the network cannot meet the next deadline, the

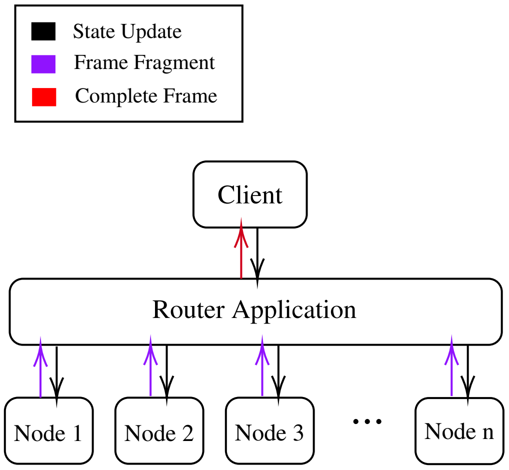

Figure 1: *Our architecture for a distributed remote rendering network. The client node emits a state update which propagates across the network. Each node renders a section of the screen and sends the frame back to the client via the router.*

client can display its low-quality render.

3.1 Implementation

We implemented this network in the G3D Innovation Engine, an open source research 3D graphics engine written in `C++`. Our library is a simple extension of the engine's native application framework and is easy to integrate into existing games and projects. We refer to these existing applications running with our library as *hosted* applications. Once integrated, an application will be able to run as an instance of a client node or a remote node. An instance of the router application can be run agnostic of the hosted application's design.

In the G3D Engine, the `GApp` class constructs and maintains a 3D scene that can be rendered at a given framerate. Developers can create their own applications by extending `GApp` and overriding app hooks like `onInit`, `onUserInput` and `onGraphics` to define custom app behavior. Our library adds background tasks to the `GApp` pipeline that are invisible to the developer while preserving flexible abstraction.

The network can be setup by initializing the router standalone driver application. Each remote node is then allowed to start up. After a remote node initializes the game, it will ping the router to be registered in the network. Once all remote nodes are registered, the client can be initialized and will similarly ping the router. Once the client is registered, the router will configure each remote node then broadcast a ready message signaling the network is now active. The client will receive the ready message and begins its game loop.

Integrated with our library, a hosted application running as a client will execute normally, but each time it completes its simulation, keyboard event listening, and network listening, our library will send a state update across the network and busy-wait for a rendered frame to be sent back. If the frame does not return in time, the client app times out and defaults to rendering its own low-quality frame (this feature was disabled in experiments to best measure network latency). A hosted application running as a remote node is a much more limited version of its client counterpart. The hosted application will busy-wait for state updates. In our implementation, the application state update is simply a batch of entity transform data, though this can easily be extended to many types of application state events. On reception, the remote node reads the update and syncs its entities with the received transform data. The remote app then triggers `onGraphics` and writes the resulting display framebuffer to the network.

When $n > 1$ remote nodes are used, each remote node is assigned a specific section of the screen to render in the configuration phase of setup. For n remote nodes and a screen of dimensions *height* and *width*, the ith node will get a screen section that is width *width* and height $height/i$ starting at $y = height * i$. This splits the screen in horizontal strips and because images in memory are stored left-to-right top-to-bottom, reassembling an image from these sections can be a simple `memcpy`. If *height* is not perfectly divisible by n, we give the remaining pixel rows to the last remote node.

Note that our implementation has many inefficiencies and much room for optimization. Additionally, a successful build of our system is contingent upon a few bugs being patched in `G3D`.

n	1	2
Client latency	80	60
Router latency	78	55
Remote latency	45	40
FPS	10	13

Table 1: Network latencies and FPS with 1 and 2 remote nodes. Note that all measurements are averages over a fixed time period. As the number of remote nodes increases, there is a clear decrease in client and router latency and an increase in framerate, showing that parallizing screen rendering can help absolve the network overhead.

4 Experiment

We hypothesize that sending images over a LAN (Local Area Network) exhibits latency decrease when image size decreases resulting in higher framerates. Figure 2 illustrates the relationship between image packet size and transit latencies on a LAN. The transit time of an image on a LAN is proportional to its size, so a linear decreasing relationship is expected. Note Figure 2 only shows latencies for single $1/n$th portions of the screen.

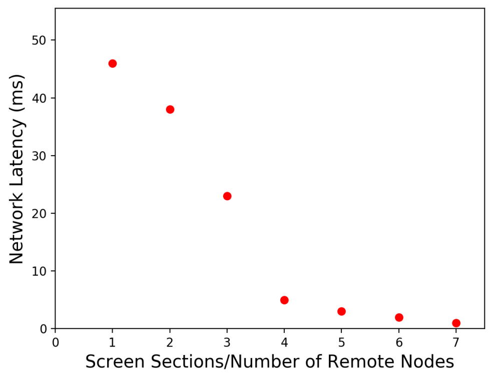

Figure 2: Average LAN latency in milliseconds of constructing and sending a single screen section. The screen was sectioned off vertically, thus for each number of remote nodes n, a single fragment was of size $width \times (height/n)$ for the *width* and *height* of the screen. By our network architecture, each screen section is (ideally) sent in parallel and in theory we would expect to see a noticeable decrease in framerate as the number of remote nodes is increased because of the exponential decrease in the time to send a single screen section. Images were encoded in JPEG which resulted in lower quality images but faster LAN transit times than PNG.

We used four machines: one client, one router, and two remote nodes (each with NVIDIA GeForce GTX 950 GPUs). We ran a game called `Simple Game` with our network setup. `Simple Game` ships with `G3D` and is a first-person style game where the player can move and interact with a basic 3D environment. Out of the box, the game runs with 115 fps on average on the client machine we used. We also measured the average fps on our network with $n = 1$ and $n = 2$ remote nodes. We tracked three additional figures:

Client latency refers to the time it took for the client to receive a rendered frame after it issued a state update, i.e. the time between frames.

Router latency refers to the time it took to receive all screen sections after it broadcast an update to all remote nodes. Note that router latency and client latency differ only in the time necessary to send the complete frame from the router to the client.

Remote latency refers to the time it took to sync with the network and render a single screen section after receiving a state update.

Note all figures are averages over multiple trials. Due to hardware constraints, our experiment was limited to a maximum of two remote nodes, but our library supports arbitrary numbers of remote nodes.

We observed a significant decrease (20 milliseconds) in time between frames on the client machine as more remote nodes were used. This difference is proportional to router latency, as expected. Remote latency does not decrease at the same rate because remote latency primarily measures render time. While we do see a decrease in the amount of time to render, the biggest value added was in decreasing the image transit latency. We expect a similar decreasing trend in client latency and increasing trend in framerate as more remote nodes are used. The optimal number of remote nodes can be determined with further experimentation as the optimal n is where client latency as a function of n has a global minimum.

5 Conclusion

We described, implemented and evaluated an event-driven distributed network for real-time remote rendering. Our network outsources expensive graphics computation to an array of machines to amortize frame renders. Each machine renders a predetermined section of the screen. Each section recombined and streamed to the client.

Our library supports arbitrary levels of scalability, bounded only by the number of pixels on a client's screen. One issue in current cloud gaming technologies is gaming during network failure or server failure is not possible. The event-driven design of our network adds fault tolerance on the client side to these possible failures. Additionally, if a rendering machine fails, the flexible configuration of our network allows for ad hoc redistribution of work with no added complexity to the client.

We observed performance increases as the size of the network was increased. Our experiments were limited by the number of GPUs available and so the optimal number of remote nodes remains to be addressed by future work. Different variations of a distributed remote rendering architectures, like [5] and [6], have also yet to be implemented and compared with our architecture. Optimizing our own implementation will also help to accurately measure the benefits and pitfalls of this approach.

6 References

[1] McGuire, M., Mara, M., and Majercik, Z. ”The G3D Innovation Engine”, http://casual-effects.com/g3d, 2019
[2] Google Stadia, *Wikipedia*, 09-Jun-2019. [Online]. Available: https://en.wikipedia.org/wiki/GoogleStadia. [Accessed: 09-Jun-2019].
[3] Game anywhere on your Mac, PC, or SHIELD with NVIDIA's cloud gaming service - GeForce NOW, *NVIDIA*. [Online]. Available: https://www.nvidia.com/geforcenow. [Accessed: 09-Jun-2019].
[4] Project xCloud: Gaming with you at the center, *The Official Microsoft Blog*, 06-Dec-2018. [Online]. Available: https://blogs.microsoft.com/blog/2018/10/08/project-

xcloud-gaming-with-you-at-the-center/. [Accessed: 09-Jun-2019].

[5] *OnLive*. [Online]. Available: http://onlive.com/. [Accessed: 09-Jun-2019].

[6] J. Mahovsky and L. Benedicenti, An architecture for java-based real-time distributed visualization, *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 4, pp. 570579, 2003.

[7] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno, Protected interactive 3D graphics via remote rendering, *ACM Transactions on Graphics*, vol. 23, no. 3, p. 695, 2004.

[8] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno, Protected interactive 3D graphics via remote rendering, *ACM Transactions on Graphics*, vol. 23, no. 3, p. 695, 2004.

[9] Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, Peter-Pike Sloan, and Chris Wyman, CloudLight: A system for amortizing indirect lighting in real-time rendering, Journal of Computer Graphics Techniques (JCGT), vol. 4, no. 4, 127, 2015