Uniquitous: Implementation and Evaluation of a Cloud-based Game System in Unity

Meng Luo and Mark Claypool
Interactive Media & Game Development and Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609, USA
{mluo2|claypool}@wpi.edu

ABSTRACT

Cloud gaming is an emerging service based on cloud computing technology which allows games to be run on a server and streamed as video to players on a thin client. Commercial cloud gaming systems, such as Onlive, Gaikai and StreamMyGame remain proprietary, limiting access for game developers and researchers. In order to address this shortcoming, we present an open source cloud-based game system in Unity called *Uniquitous* that gives game developers and researchers control over both the cloud system and the game content. Detailed experiments evaluate performance of Uniquitous for three main parameters: game genre, game resolution and game image quality. The evaluation results are used in a data model that can predict in-game frame rates for systems with configurations beyond those tested. Validation experiments show the accuracy of our model and complement the release of Uniquitous as a research and development platform.

1. INTRODUCTION

Cloud gaming is estimated to grow from $1 billion in 2010 to $9 billion in 2017 [4], a rate much faster than either boxed games sold through traditional retail or games sold online. In 2012, Sony bought the Gaikai[1] cloud gaming service for $380 million and integrated this service into their PlayStation 4 [11, 12].

Cloud gaming provides benefits to users, developers and publishers over traditional gaming. Cloud gaming services mean players no longer need to upgrade their gaming hardware, such as desktops, laptops and game consoles, in order to play the latest games, as well as opens up game opportunities for users with low-end gaming hardware. Game developers only need to develop one version for the cloud server platform instead of developing a version for each client type, thus reducing game development time and cost. Publishers can more easily protect against piracy since cloud games are only installed in the cloud, thus limiting the ability of malicious users to copy them.

Despite some recent success, cloud gaming faces a number of challenges before it can be deployed widely for all types of games, game devices and network connections: 1) network latency, inherent in the physical distance between the server and the client, must be overcome; 2) high network capacities are needed in order to stream game content as video down to the client; and 3) processing delays at the cloud gaming server need to be minimized in order for the game to be maintained, rendered and streamed to the client effectively for playing. Research and development required to overcome these challenges needs cloud gaming testbeds that allow identification of performance bottlenecks and exploration of possible solutions.

Currently, several commercial cloud gaming systems, such as OnLive[2] and StreamMyGame[3] have been used for cloud gaming research. Although these commercial services can be readily accessed, their technologies are proprietary, providing no way for researchers to access their code. This makes it difficult for researchers to explore new technologies in cloud-based gaming, such as latency compensation techniques, and makes it difficult for game developers to test their games for suitability for cloud-based deployment. While GamingAnywhere [7] provides a complete open source cloud gaming system, it remains completely separated from the game itself, not supporting integration and simultaneous exploration of game code and the cloud system.

In order to provide a more flexible and easily accessed platform for cloud gaming researchers and game developers, we present *Uniquitous*,[4] a cloud gaming system implemented using Unity.[5] Unity is a cross-platform game creation system with a game engine and integrated development environment. Uniquitous blends seamlessly with Unity, making it especially convenient for Unity developers, one of the largest and most active developer communities in the world – the Unity community increased from 1 million registered developers in 2012 to 2.5 million in 2014, with approximately 600,000 active monthly [14].

Uniquitous is open source, allowing modification and configuration of internal cloud gaming structures, such as frame rate, image quality, image resolution and audio quality, in order to explore system bottlenecks and meet different client-

[1] `http://gaikai.com`

[2] `http://onlive.com/`

[3] `http://streammygame.com/`

[4] `http://uniquitous.wpi.edu/`

[5] `http://unity3d.com/`

Submitted to the 25th ACM SIGMM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), March 19th, 2015 Portland, Oregon, USA.

server requirements. In addition to system modifications, by being in Unity, Uniquitous enables game content adjustments for further exploring the relationship between game content and cloud gaming performance. For example, game objects can be adjusted to study the effect of scene complexity on network bitrates, or camera settings can be altered to study the effect of perspective on cloud gaming frame rates.

This paper presents the design and implementation of Uniquitous and performance evaluation on two levels: a) micro-experiments that detail performance for each component in Uniquitous, providing bottleneck analysis on components that limit game performance; and b) macro-experiments that measure the overall performance of Uniquitous under various configurations as perceived by the player. In addition, this paper presents a model for predicting Uniquitous with different configurations, enabling estimation of performance for games and hardware not yet tested. Validation of the model suggests it is effective for predicting frame rate over a range of configuration parameters.

The rest of the paper is organized as follows: Section 2 presents work related to the design, implementation and evaluation of cloud gaming systems and Uniquitous; Section 3 describes the design and implementation of Uniquitous; Section 4 details the micro experiments and their results; Section 5 does the same for the macro experiments, also presenting our model; and Section 6 summarizes our conclusions and mentions possible future work.

2. RELATED WORK

This section lists research work related to cloud gaming systems, in the areas of architecture, frameworks and measurement.

2.1 Cloud System Architecture

There is no single agreed-upon cloud system architecture. However, a four-layer architecture defined by Foster et al. [5] has frequently been used by researchers. Foster et al.'s model from bottom to top has a fabric layer, unified resource layer, platform layer and application layer. For our work, the Uniquitous server runs at the application layer. Foster et al. also list cloud services at three different levels: infrastructure as a service, platform as a service, and software as a service. Cloud gaming in general, and Uniquitous specifically, is an example of software as a service.

2.2 Cloud Gaming Frameworks

Cloud gaming frameworks can be classified into three types based on how they allocate the workload between cloud servers and clients [7]: 3d graphics streaming, video streaming and video streaming with post-rendering. All three approaches reduce computation on the client versus a traditional game architecture because the game world is managed on the server instead of the client. In 3d graphics streaming, as done by de Winter et al. [16], instead of sending video, the server sends graphics commands to the client and the client renders the game scene images. Shea et al. [13] describe a cloud gaming system using a video streaming framework where the server is responsible for rendering the game scene, compressing the images as video, and then transmitting to the client. The video streaming with post-rendering operations approach is in-between the other two approaches, performing part of the rendering process on the server and the rest on the client.

The video streaming approach is discussed the most in current research [8, 9, 13] and is currently used by most existing commercial cloud gaming systems since it reduces the workload on the client the most compared to the other two approaches. Uniquitous also uses the video streaming approach.

2.3 System Measurement

Huang et al. [7] measure system delays in GamingAnywhere, isolating the system measurements into individual components. Our micro-experiments for Uniquitous similarly decompose the system into smaller parts for bottleneck analysis.

Chang et al. [2] propose a methodology for quantifying the performance of gaming thin-clients. From results for LogMeIn, TeamViewer and UltraVNC, they demonstrate that frame rate and frame quality are both critical to gaming performance, with frame rate being the slightly more important of the two. Claypool et al. [3] use a custom game with levels that combine different actions and perspectives to measure user performance with different display settings. Their user study results show that frame rate has a much greater influence on user performance than does frame resolution. Based on these conclusions, recommendations for Uniquitous configurations preserve frame rate at the cost of frame quality and frame resolution, as appropriate.

3. UNIQUITOUS

Uniquitous, shown in Figure 1, is composed of three entities: Unity Project, Uniquitous Server and Uniquitous Thin Client. The Uniquitous Server and the Uniquitous Thin Client run on two separate computers connected by an Internet connection while the Unity Project runs on the same computer as the Uniquitous Server.

Figure 1 shows three types of data flows in Uniquitous, illustrated with different shades/colors: the red image data flow carries data for the game frames (Section 3.1); the green audio data flow carries data for the game audio (Section 3.2); and the blue input data flow carries data for user input (Section 3.3). Flows within components on the same machine are represented with dashed lines while flows across the network are shown with solid lines.

3.1 Image Data Flow

The Game Window component is part of the Unity Integrated Development Environment and works with the Unity camera to capture and display the game content to the player. The Screen Capture component is used to capture the game screen from the Game Window, reading the pixel data and storing it as a `Texture2D` object. The Image Encoding component is implemented using a JPEG encoder [1] that is configurable with a JPEG quality factor that changes the visual quality and compression ratio of the game image. The JPEG encoder compresses the `Texture2D` object into byte arrays. The Image Transmission component uses remote procedural calls (RPCs) provided by `ulink`[6] to send the image byte arrays to the client. Once the encoded byte arrays arrive on the client, the Image Reception component passes the compressed data to the Image Decoding component. The Image Decoding component loads the data back into a `Texture2D` object for display. The Image Dis-

[6]http://developer.muchdifferent.com/unitypark/uLink

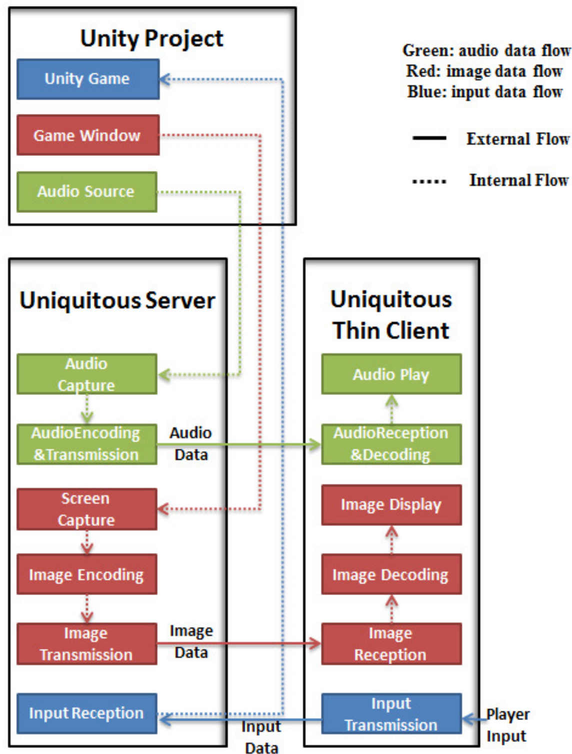

Figure 1: Uniquitous architecture

play component is implemented using the `DrawTexture()` method and called in the `OnGUI()` event function, drawing the game image on the client's screen. The image size and position on screen can be adjusted by users.

3.2 Audio Data Flow

The Audio Source component receives all game audio in the scene from the Unity audio listener. The Audio Capture component records audio data from the Audio Source using `OnAudioFilterRead()`, getting a chunk of audio data about every 20 ms. Each chunk is converted it into an array of 8192 bytes and written into a localhost TCP network stream to the Audio Encoding & Transmission component. The Audio Encoding & Transmission component uses FFMPEG,[7] a cross-platform system to record, convert and stream audio and video, running in a separate process. The audio data is compressed via MP3 and streamed over UDP to the Audio Reception & Decoding component on the client. The Audio Reception & Decoding component uses FFPLAY, a portable media player implemented using FFMPEG and SDL[8] running as its own process to receive the compressed audio stream from the server. The Audio Encoding & Transmission component decodes the audio stream and plays the sound on the client's speakers.

3.3 Input Data Flow

The Image Transmission component receives input data from the player via Unity. Unity provides access for three types of player input data: mouse positions, mouse clicks and keyboard strokes. Mouse movements are translated into the `Vector3` coordinates of the mouse cursor on the client's screen. Mouse clicks are translated into an integer value, 0 or 1. Arrow keys are translated into values between -1 and 1 along the horizontal and vertical axes. Other keyboard strokes are translated into the ASCII values of the keys pressed. The Image Transmission component uses UDP RPC for network transmissions from the client to the server. The interactions with the Unity built-in GUI system such as `GUI.Button` and `GUI.TextField` cannot be transferred since these interactions are event based. Instead, the Input Reception component passes player input to the Unity Game component using a custom GUI system for all game scripts affected by user input, such as character movement and GUI interactions.

4. MICRO EXPERIMENTS

This section presents experiments evaluating the Uniquitous server components based on the architecture from Figure 1, focusing on processing time. Due to space constraints, only the most important components are presented, with a full evaluation available in the thesis [10].

4.1 Setup

All experiments were run on PCs with Intel 3.4 GHz i7-3770 processors, 12 GB of RAM and AMD Radeon HD 7700 series graphic cards, each running 64-bit Windows 7 Enterprise. The PCs were connected by a 100 Mbps network LAN. The games tested, the Car Tutorial[9] and Angry Bots (version 4.0)[10] are provided by Unity Technologies.

4.2 Measuring Time

Three different methods provided the processing times of different components: 1) The Unity Pro Profiler allowed observation of the CPU time for each component on the server for a fixed number of frames, averaged to provide the per frame CPU time. This method was used for the Unity Project and the Game Window. 2) Time stamps in the source code before and after each component measured time per frame for a fixed number of frames, averaged to provide per frame times. This method was used for the Screen Capture time, Image Encoding time and Image Transmission time. 3) The Unix (Cygwin) command `time` was used to get CPU time for running a component. Ten runs were repeated and averaged from the results. This method was used for the Audio Encoding & Transmission component.

4.3 Screen Capture

Figure 2 shows the per frame screen capture time for both games at nine different resolutions. The x axis is the number of pixels (width × height) and the y axis is the screen capture time, in milliseconds. Each point is the average, shown with standard error bars. The blue square trend line is for the Car Tutorial game and the red dot trend line is for the Angry Bots game.

Generally, the capture time increases linearly with pixels, with the exception of the mid-range where the trendlines are relatively flat. The capture time is mostly independent of the game. Capture times are potentially low enough to

[7] http://ffmpeg.org
[8] http://libsdl.org
[9] https://www.assetstore.unity3d.com/en/#!/content/10
[10] https://www.assetstore.unity3d.com/en/#!/content/12175

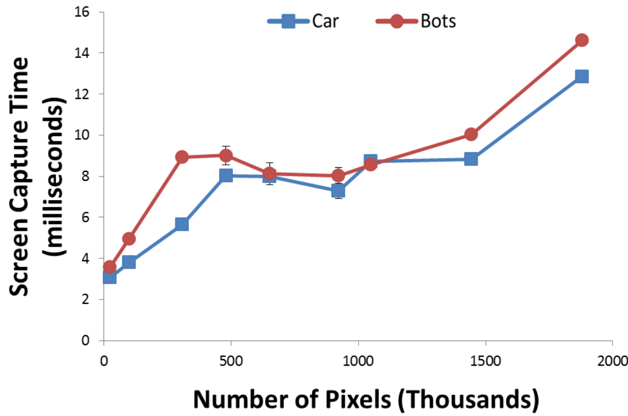

Figure 2: Screen capture time versus resolution for both games

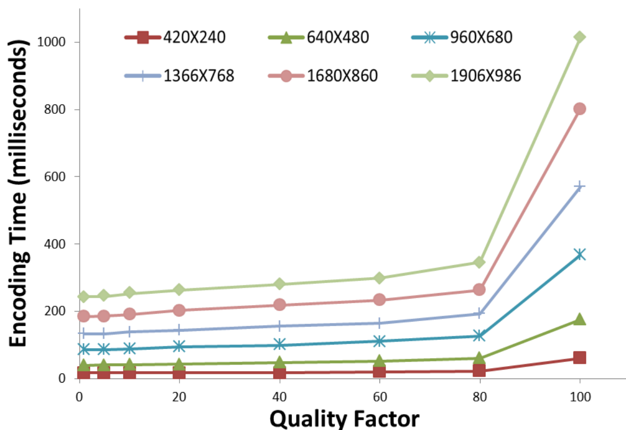

Figure 3: Image encoding time versus JPEG encoding quality factor for Angry Bots

achieve a target frame rate time of 30 fps (i.e., under 33 milliseconds).

4.4 Image Encoding

Figure 3 shows the image encoding time versus JPEG encoding quality factor at different resolutions when running Angry Bots. The x axis is the JPEG quality factor, where a higher factor represents a better game image. The y axis is the encoding time, in milliseconds. Each source image resolution is depicted by a separate trend line.

There is a slight increase in per-image encoding time as the quality factor goes from 0 to 80, with a sharp increase at 100. Higher resolution images take more processing time than lower resolution images and encoding times low enough to achieve a target frame rate time of 30 fps are only possible for images size 640×480 or smaller.

4.5 Network Bitrate

Uniquitous was configured with a resolution of 640×480 and a JPEG encoding quality factor of 20, and then Angry-Bots was run for five minutes. Wireshark[11] captured the network packets sent between the Uniquitous server and the client during this time. Data collected during a one-minute period in the middle of the game was analyzed. The downlink bitrate fluctuates around 3.5 Mbps, with a standard error of 0.025., and the uplink bitrate fluctuates around 32 Kbps with a standard error of 0.159.

[11]`http://wireshark.org`

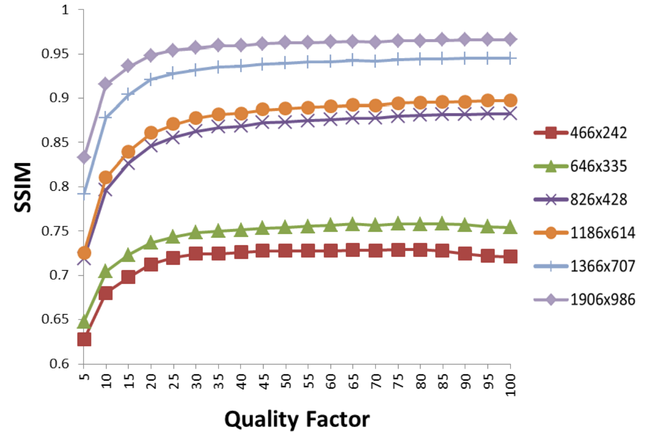

Figure 4: SSIM versus JPEG encoding quality factor for Car Tutorial

5. MACRO EXPERIMENTS

This section presents experiments evaluating the Uniquitous system as perceived by the player, focusing on game image quality (Section 5.1) and frame rate (Section 5.2). The setup is as for the micro experiments. As for Section 4, a full evaluation is available in the thesis [10].

5.1 Game Image Quality

In order to evaluate the game image quality provided by Uniquitous, we used 200 images with 20 different compression ratios and 10 different resolutions. For reproducibility, all results derive from an image captured from Car Tutorial. Structural Similarity Index (SSIM) [15] is used to evaluate the visual quality of the compressed images.[12] SSIM models the image distortion as a number from 0 (low) to 1 (high) due to compression as a combination of loss of correlation, luminance distortion and contrast distortion.

The SSIM results are shown in Figure 4. The x axis is the JPEG quality factor, and the y axis is the SSIM. Each point is the average SSIM per frame for an experimental run, with trendlines grouping the different screen resolutions.

From Figure 4, higher resolution images have better SSIM. All lines show a marked increase in SSIM from quality factor 0-15 and modest increase from 15-35. After 35, the increase is slight or, in the case of the lowest resolutions, none. This suggests there is little visual benefit to JPEG quality factors above 35.

5.2 Frame Rate

To evaluate frame rates achievable in Uniquitous, 44 configurations for Car Tutorial and 37 configurations for Angry Bots were tested. Each configuration varied the JPEG encoding quality factor and resolution. To compute the frame rate, the time difference methodology (Section 4.2) provided the average frame intervals, and the inverse provided the

[12]PSNR results available in in [10].

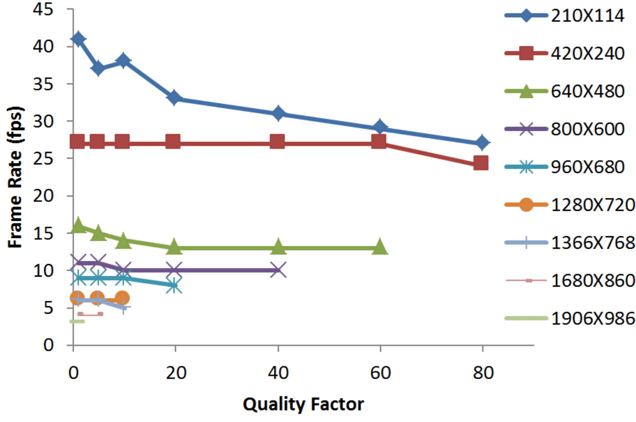

Figure 5: Frame rate versus JPEG quality factor for Angry Bots

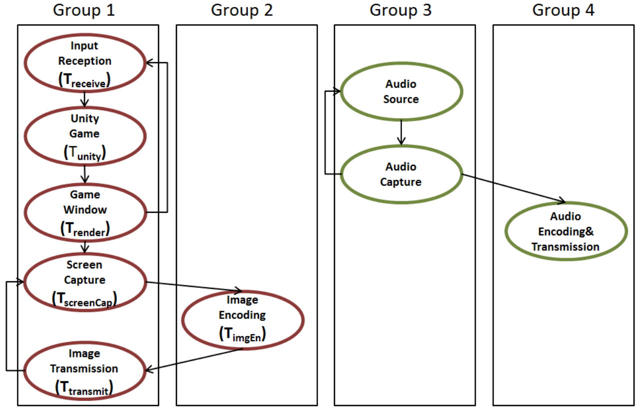

Figure 6: Parallel working structure of Uniquitous Server

frame rate. Due to space constraints, only the Angry Bots results are discussed, but Car Tutorial results are similar.

Figure 5 shows the frame rate results for Angry Bots. The x axis is the JPEG quality factor, and the y axis is the frame rate. Each point is the average of an experimental run, with trendlines grouping the different screen resolutions. Note, the higher resolution images are not tested at higher JPEG quality factors since RPC limits prevent images larger than 64 Kbytes from being transmitted.

From Figure 5, Angry Bots can achieve a maximum frame rate of 41 fps at a 210×114 resolution and 1 JPEG encoding. With the exception of this smallest image resolution, decreasing the JPEG quality factor does little to change the frame rate. However, increasing the frame resolution has a pronounced effect on decreasing the frame rate for both games. Based on previous results [3], frame rate is more important than resolution and a game system needs to provide a minimum of 15 fps for reasonable player performance. Both games tested can achieve 15 fps at a resolution of 640×480, the recommended resolution setting for Uniquitous on this setup.

5.3 Predicting Frame Rate

Since the frame rate depends on the processing and delivery time for each frame, predicting the frame rate for a Uniquitous configuration proceeds by modeling the bottleneck components on the server.

By default, the Uniquitous Server operates over 4 separate groups, illustrated in Figure 6, each group running in a separate thread: group 1 runs in the main Unity thread; group 2 runs the Image Encoding component (the JPEG encoder); group 3 runs the Unity audio engine; and group 4 runs an independent FFMPEG process. Since analysis shows audio is not the bottleneck [10], only group 1 and group 2 are considered when modeling performance. In group 1, the Input Reception, Unity Game and Game Window components run in parallel with Image Encoding from group 2, while Screen Capture and Image Transmission do not. Each frame, Screen Capture captures and delivers the screen then waits until both the Image Encoding and Image Transmission components are done before capturing the next screen. Similarly, Image Transmission waits until Image Encoding is done. Thus, although running in separate threads, Screen Capture and Image Transmission can be blocked by Image Encoding.

Whichever processing time is larger – the first three components of group 1 or the last two components of group 1 and group 2 – determines how fast each frame can be provided and the overall frame rate:

$$F = \frac{1}{T} \quad (1)$$

$$T = max(T_{1'}, T_2) + T_{screenCap} + T_{transmit} \quad (2)$$

$$T_{1'} = T_{unity} + T_{render} \quad (3)$$

$$T_2 = T_{imgEn} \quad (4)$$

$T_{1'}$ is the processing time of the first three components of group 1. T_2 is the processing time of group 2 (Image Encoding). T is the frame interval, the sum of $T_{screenCap}$, $T_{transmit}$ and the maximum of T_1 and T_2. F is the predicted frame rate. $T_{receive}$ is effectively ignored because the processing time for receiving input is negligibly small compared to the processing time of other components.

In addition, since Screen Capture, Image Encoding and Image Transmission run in a coroutine that is called periodically every 20 milliseconds, the transmission of the compressed image (and thus the processing time for all three components) has an additional delay, $T_{coroutine} \in [0, 20]$:

$$T = max(T_1, T_2) + T_{screenCap} + T_{transmit} + T_{coroutine} \quad (5)$$

In order to build a model predicting frame rates for Uniquitous configurations not measured, we used a Weka classifier [6] with a 10-fold cross validation to make a linear regression model for both games:

$$F_{CarTutorial} = 1 \ / \ (0.1348 \times R + 0.118 \times Q + 21.0) \quad (6)$$

$$F_{AngryBots} = 1 \ / \ (0.1361 \times R + 0.1224 \times Q + 22.5) \quad (7)$$

F is the predicted frame rate, R is the total pixel resolution divided by 1000, and Q is the JPEG quality factor.

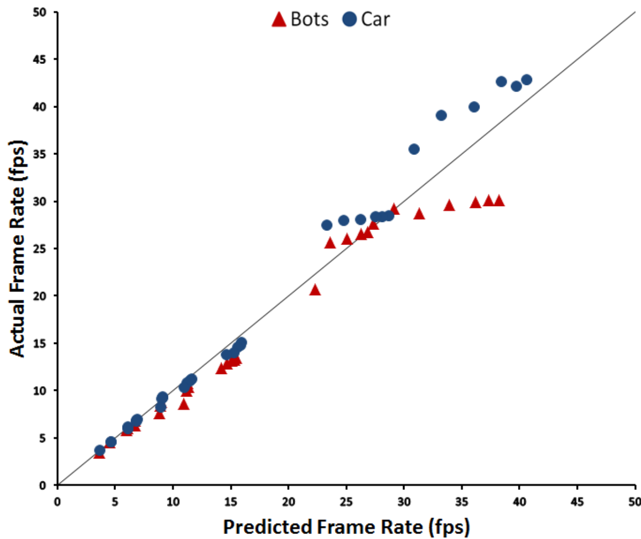

Figure 7: Actual versus predicted frame rate

In order to validate our model, we chose new R and Q values that had not been tested before, 35 for the Car Tutorial and 30 for Angry Bots, and measured the actual frame rates recorded. The results are show in Figure 7.

The x axis is the predicted frame rate and the y axis is the actual frame rate as measured. Each point is the average frame rate over the experimental run. The diagonal line shows what would be perfect prediction. Generally, most of the data points are near this line, showing the accuracy of the model. The points are somewhat closer to the line for frame rates under 20 fps than for frame rates over 25 fps, probably due to unaccounted for processing that accumulates more with more frames per second. The actual and predicted frame rate have a correlation of 0.995 for Car Tutorial and 0.981 for Angry Bots.

6. CONCLUSION

Realizing the potential for cloud gaming requires testbed systems for researchers and developers. This paper presents Uniquitous,[13] an open source cloud gaming system in Unity. Uniquitous seamlessly blends with Unity game development, providing control not only over the game system but also over the game content in a cloud-based environment. Micro experiments provide performance evaluation of the Uniquitous components, macro experiments evaluate game quality of the Uniquitous system, and models allow for prediction of Uniquitous frame rates for configurations not yet tested.

The micro experiments show that game resolution does not correlate with processing time for the Unity Project and Game Window components, but does with the Screen Capture, Image Transmission and Image Encoding components. Moreover, the processing time of Image Transmission and Image Encoding correlates with game image quality. The Unity Project running the game is the most time consuming component on the server when the game image quality and resolution are both low, but Image Encoding becomes the bottleneck for higher resolutions.

The macro experiments show game image quality increases markedly as the JPEG quality factor increases between 1 and 15, shows a modest increase between 15 and 35, and little improvement after 35. For our system testbed, JPEG quality factors below 35 and resolutions below 640×480 provide frame rates near 15 fps, suitable for game play [3]. Validations show our models based on resolution and image quality can be used to accurately predict frame rates, and thus pre-configure Uniquitous for acceptable performance.

Future work can seek to increase Uniquitous frame rates and/or support higher resolutions and image qualities. In addition, more game genres can be tested, exploring the relationship between the game genre and cloud gaming performance. Uniquitous can be deployed on mobile devices, allowing for evaluation of cloud gaming on wide-area networks and resource limited devices.

[13] `http://uniquitious.wpi.edu/`

7. REFERENCES

[1] A. Broager. JPEG Encoder Source for Unity in C#. `goo.gl/FwOfOW`. [online; accessed 06-May-2014].

[2] Y.-C. Chang, P.-H. Tseng, K.-T. Chen, and C.-L. Lei. Understanding the Performance of Thin-Client Gaming. In *Proceedings of Communications Quality and Reliability (CQR)*, Naples, FL, USA, May 2011.

[3] M. Claypool and K. Claypool. Perspectives, Frame Rates and Resolutions: Its all in the Game. In *Proceedings of Foundations of Digital Games (FDG)*, FL, USA, Apr. 2009.

[4] Distribution and Monetization Strategies to Increase Revenues from Cloud Gaming. `goo.gl/YnlE9V`, 2012. [online; accessed 01-May-2014].

[5] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Proceedings of Grid Computing Environments Workshop (GCE)*, pages 1–10, Austin, TX, USA, Nov. 2008.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1), 2009.

[7] C. Huang, Y. C. C. Hsu, and K. Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proceedings of ACM MMSys*, Oslo, Norway, Feb. 2013.

[8] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoβfeld. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. In *IEEE Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 330–335, Seoul, Korea, 2011.

[9] Y.-T. Lee, K.-T. Chen, H.-I. Su, and C.-L. Lei. Are All Games Equally Cloud-gaming-friendly? An Electromyographic Approach. In *Proceedings of IEEE/ACM NetGames*, Venice, Italy, Oct. 2012.

[10] M. Luo. Uniquitous: Implementation and Evaluation of a Cloud-Based Game System in Unity 3D. Master's thesis, Worcester Polytechnic Institute, 2014. Adv: M. Claypool.

[11] J. Plafke. CES 2014: Gaikai Becomes PlayStation Now, Streaming Games To Just About Everything. `goo.gl/AOMK5V`, Jan. 2014. [online; accessed 01-May-2014].

[12] S. Sakr. Sony Buys Gaikai Cloud Gaming Service for $380 Million. `goo.gl/S9P3KJ`, July 2012. [online; accessed 01-May-2014].

[13] R. Shea, J. Liu, E. Ngai, and Y. Cui. Cloud Gaming: Architecture and Performance. *IEEE Network*, 27:16–21, Aug. 2013.

[14] Unity Company Facts. `http://unity3d.com/public-relations`, acc 01-May-2014.

[15] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, Apr. 2004.

[16] D. Winter, P. Simoens, L. Deboosere, F. Turck, J. Moreau, B. Dhoedt, and P. Demeester. A Hybrid Thin-client Protocol for Multimedia Streaming and Interactive Gaming Applications. In *NOSSDAV*, Newport, RI, USA, May 2006.