2014-12-18

Uniquitous: Implementation and Evaluation of a Cloud-based Game System in Unity3d

Meng Luo
Worcester Polytechnic Institute

UNIQUITOUS: IMPLEMENTATION AND EVALUATION OF A CLOUD-BASED GAME SYSTEM IN UNITY 3D

by

Meng Luo

A Thesis

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Interactive Media & Game Development

December 2014

Approved:

_____

Thesis Advisor: Professor Mark Claypool

Approved:

_____

Committee: Professor Robert W. Lindeman

Approved:

_____

Committee: Professor David Finkel

ABSTRACT

Cloud gaming is a new service based on cloud computation 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 these shortcomings, we developed an open source Unity3d cloud-based game system called *Uniquitous* that gives the game developers and researchers control of system and content. Detailed experiments evaluate performance of 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 that have not been tested. Validation experiments show the accuracy of our model and allow us to use the model to explore cloud-based games in a variety of system conditions.

Table of Contents

1. Introduction

Cloud gaming is a new service that is based on cloud computing technology. It allows games to be run on the server and be played remotely by players through a thin client. Cloud games in video games are estimated to grow from 1 billion US dollars in 2010 to 9 billion US dollars in 2017 **[1]**. The growth rate of cloud game sales during this period is forecasted to be much faster than either boxed-games or online-sold games. In 2012, Sony bought Gaikai **[2]** service for 380 million US dollars and integrated this service into their PlayStation in January 2014 **[3, 4]**.

Cloud gaming can bring more benefits to users, game developers and publishers over traditional gaming. With the widespread use of cloud gaming service, users no longer need to upgrade their hardware devices, like desktops, laptops and game consoles, in order to install and run new released games that are not compatible with the old systems. Moreover, cloud gaming makes it possible for users to play the same game on different platforms and provides more game choices to users who own low-end hardware devices. For game developers, they only need to develop one game build for the target platform on a cloud gaming server instead of considering developing multiple versions of the game adapted to various platforms, resulting in reduction in the game development time and cost. For publishers, they will not need to concern about piracy issues since cloud games will not be distributed and will only be available on demand with small monthly subscription fees.

Since cloud game based technology is still in its early stage, there are a number of major issues and challenges cloud gaming providers are facing. Firstly, there is network latency, brought by the physical distance between the server and the client. Secondly, higher bandwidths are needed in order to transmit large amount of video content between the server and the client

over the Internet. For example, the recommended minimum bandwidth for OnLive is 2 Mbps **[5]**, which can be difficult to achieve for a single user. Last but not least, the processing latency caused by the cloud gaming server needs to be reduced effectively by improving the performance of the software and hardware. For research purposes, it is useful to have a cloud gaming testbed to seek solutions to these issues and explore possibilities of the new technology.

At present, there are a number of commercially used cloud gaming systems, such as Onlive **[6]**, Gaikai and StreamMyGame **[7]** that have been successfully used in research testbeds. Even though their services can be easily accessed by anyone, the technologies they apply and the detailed architecture of their cloud gaming systems remain proprietary. Most cloud gaming companies like Onlive run and maintain servers in data centers and researchers are not able to run server on their own. Moreover, there is no way for researchers to access the code on either the client or the server to explore technology such as latency compensation. It is difficult for game developers who want to create their own cloud games to test playability of their games. Even though StreamMyGame allows users to set up their own server and client on a private LAN testbed, flexibility is limited in the game content and system is hidden to users.

In 2013, the first complete open source cloud gaming system, GamingAnywhere, was released. As an open system, the video streaming pipeline can be replaced by another component implementing a different standard, algorithm or protocol **[8].** With its extensibility and reconfigurability, GamingAnywhere is a good option for researchers and game developers to observe and measure different aspects of cloud games with customized parameter settings of the cloud gaming system.

However, both GamingAnywhere and StreamMyGame can only run games that are already built, so the game on the server runs independently without control over the game content itself.

Researchers/developers have no access to the source code of the game as it interacts with the game application through operating system specific API calls. For instance, users are not able to study the effect of some specific game content such as a game character, a game environment object or a game audio on the performance of a game running on cloud, since they cannot modify the game itself unless they have source code of the game.

In order to provide a more flexible and easily accessed platform for cloud gaming researchers and game developers, we developed a cloud gaming system called Uniquitous, which is implemented using Unity 3d **[9]**. Unity 3d is a cross-platform game creation system including a game engine and integrated development environment. It has one of the largest and most active developer communities in the world - the Unity community has increased from 1 million registered developers in 2012 to 2.5 million registered developers in 2014 and there are 600,000 Unity developers active monthly **[10].** Since Unity 3d allows creation of game projects and Uniquitous can be integrated into any Unity game development project seamlessly, Uniquitous provides platform control over the game content and game system in a cloud based game environment. This is especially convenient for Unity developers as Uniquitous blends seamlessly with their game development. Uniquitous allows modifications to its internal structures, configurations on its system parameters such as image quality, image resolution and audio quality, in order to meet different client-server requirements. Most importantly, game content adjustments can be done in Uniquitous for various purposes. For example, different game objects can be removed or added so that the effect of scene complexity in the game on the performance of cloud games can be adjusted. Or, in order to adjust performance of different camera views on frame rate for cloud gamers, camera settings can be adjusted or multiple cameras can be added to the game scene at the same time.

Uniquitous is composed of two entities, the Uniquitous server and the Uniquitous thin client. The architecture allows Unity game developers to run their Unity projects on the server with some minor modifications to input control and interact with their projects on the client remotely. In Uniquitous, players provide input at the client is then sent to the server. Upon receiving the input from the client, Unity projects running on server update their game states and the Unity engine renders new content based on the updated game states. The Uniquitous captures rendered game screens and game audio, compresses the images and audio data and transmits both to the client. The client continually receives the image data and audio data and decompresses them for displaying and playing.

After implementation of the system, we did a micro evaluation of the Uniquitous server. In order to understand bottlenecks to perform in cloud game systems and predict Uniquitous performance under alternate configurations, we mainly measured the performance of its subcomponents under different settings of three parameters: game genre, game resolution and game image quality. We did a macro evaluation to evaluate and predict the performance of Uniquitous. In the macro evaluation, we first did objective measurements of the game quality under different system settings. Then we measured the frame rate on both the server and the client under different system settings. We also derived a model to predict the frame rate on the server based on our analysis of the working structure of the server and validated its correctness. Finally, we built and validated a model for Uniquitous to predict the frame rate on the client based on the given resolution and quality factor settings.

In this thesis, we explain in detail how we implemented and evaluated Uniquitous. The rest of the paper is organized as follows. Section 2 presents related work which provides guidance and reference for implementation and evaluation of Uniquitous. Section 3 describes the

implementation of Uniquitous in detail. Section 4 provides the results of performance evaluation on each subcomponent comprising the Uniquitous server and the network estimate. Section 5 gives the results of performance evaluation on the entire Uniquitous system set up over LAN. Section 6 gives the conclusion and future work.

2. Related Work

This section lists several experiments, user studies, and research work relevant to cloud gaming systems. This provides an overview of the system architecture and insights into focus areas for evaluating system performance of Uniquitous, including the processing time, game screen resolution, game image quality, game frame rate and factors critical to players' performance.

2.1 Cloud Systems

There is no single agreed-upon cloud system architecture. However, a four-layer model for cloud system architecture, defined by Foster et al. **[11]** has been used very frequently by researchers. The model is shown in Figure 1. The fabric layer contains the hardware level resources such as storage resources and network resources. The Unified resource layer contains resources that have been encapsulated. The Platform layer contains specialized tools, middleware and services to provide a deployment platform such as a Web hosting environment and scheduling service. The Application layer contains the application that runs in the cloud. For our work, the Uniquitous server is running at the Application layer.

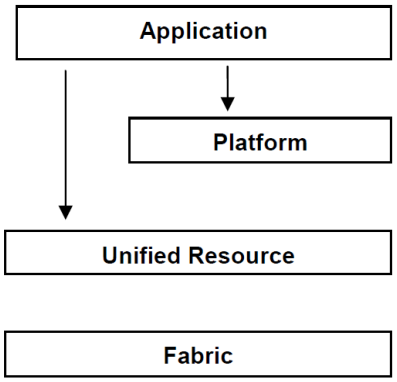

Figure 1. Four-layer model for cloud system architecture **[11]**

Foster et al. **[11]** also listed cloud services at three different levels: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS provides the user with virtual server instances, storage and APIs used for dealing with the instance and storage. In PaaS, software development tools hosted on the provider's infrastructure can allow users to create application on provider's platforms over the Internet. SaaS provides the hardware infrastructure, the software product and interacts with the user through a front-end portal **[12]**. Cloud gaming, and Uniquitous, is an example of SaaS, where users can send inputs and get streamed game content through a client program on a remote device.

2.2 Cloud Gaming Frameworks

Different types of frameworks are classified into three approaches based on how they allocate the workload between cloud servers and clients. They are 3D graphics streaming approach, video streaming approach and video streaming with post-rendering operations approach **[8]**. All three approaches can help reduce the workload of the game client because all game logic is running on the server instead of the client. Shea et al. **[13]** outlined the framework for a cloud gaming system with the video streaming approach, which is shown in Figure 2, where the server is responsible for the video rendering, compressing and transmission. In the 3D graphics streaming approach, the server moves GPU rendering task to the client side. Instead of sending compressed video frames to the client, the server sends compressed intercepted graphics commands to it and the client is responsible for rendering video frames. The video streaming with post-rendering operations approach is somewhere between the other two approaches, which performs a part of rendering process on the server and the rest on the client. The video streaming

approach is discussed the most in research **[13, 14, 15]** and it is currently used by most existing commercial cloud gaming systems such as Onlive, Gaikai and StreamMyGame as it reduces the workload on the client to the minimum degree compared to the other two approaches.

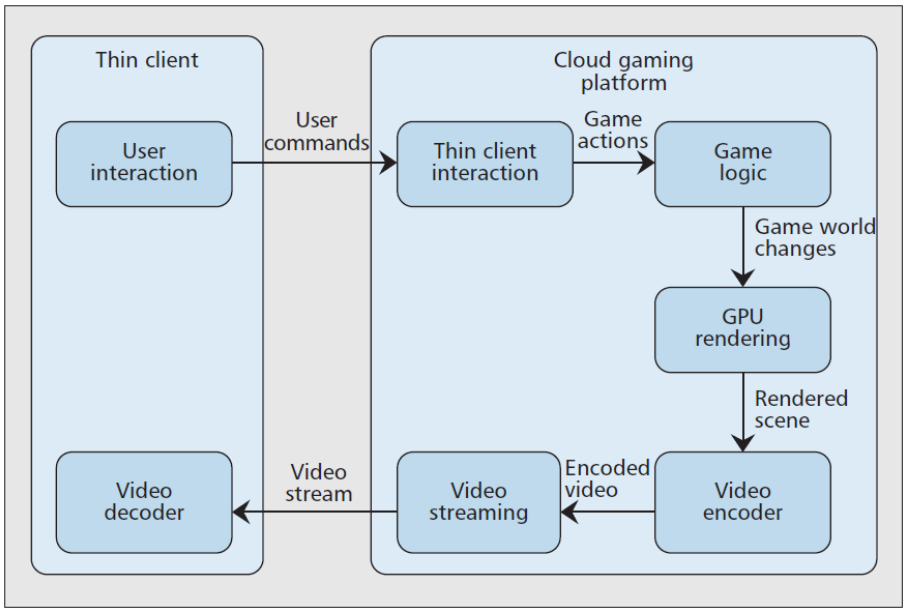

Figure 2. A popular framework for cloud gaming system **[13]**

For the purpose of reducing workload on the client, we used the video streaming approach to implement Uniquitous. In addition to GamingAnywhere **[8]**, there are many other systems developed using the video streaming approach. For example, a hybrid thin-client system using this approach was proposed in **[16]**, which is a real-time desktop streamer that uses a video codec to stream the graphics output of applications after GPU rendering to a thin client that is capable of decoding the video stream. And Holthe et al. **[17]** used the same approach to implement a system composed of a remote game server and a thin client. All these implementations can provide useful guidance to design the architecture of Uniquitous system.

2.3 System Measurement

Huang et al. **[8]** selected three games of different genres to run on cloud gaming systems and measured the system delay of GamingAnywhere. They divided the system delay into a number of smaller components to better analyze each delay subcomponent independently. Their experiment results show that GamingAnywhrere performs better than the two well-known commercial cloud gaming systems, Onlive and StreamMyGame, in terms of the processing delay and achieved video quality. In our micro-evaluation of Uniquitous, we decomposed the system delay into smaller parts in order to better understand and analyze sources of delays and optimized their performance respectively by minimizing the processing time as much as possible.

Kay et al. **[18]** measured send and receive processing times for various operations in kernel space on a DECstation 5000/200(25MHz processor) running Ultrix 4.2a, which includes computation of the Internet checksum in UDP and IP, various bulk data copying operations, allocation and freeing of memory buffers and all other operations. Their results under different length messages show that the processing time is increasing with the length of message, and the accumulated time of all operations for both send and receive processing is around 3000 microseconds at the maximum byte length (8192 bytes) tested. The DECstation 5000/200 uses processor with clock rate of 25MHz, which is about 140 times lower than the clock rate of the processor of hardware we used in evaluating Uniquitous, which is 3.4GHz. This suggests the processing time of sending and receiving UDP packets in kernel space on today's servers to be at microsecond level. Thus, in our micro evaluation, we did not measure the kernel space processing time of remote procedural calls in Unity.

Chang et al. **[19]** proposed a methodology for quantifying the performance of thin-clients on gaming. From their case study results on three thin clients, LogMeIn, TeamViewer and

UltraVNC, they demonstrated that display frame rate and frame quality degradation of the clients are both critical to gaming performance and that frame rate has a greater impact on gaming performance than does frame quality. Claypool et al. **[20]** used a custom game with levels that combines different actions and perspectives to measure user performance with different display settings. The analysis on user study experiments shows that frame rate has a much greater influence on user performance than frame resolution. Based on their conclusions, a cloud system like Uniquitous should preserve frame rate at the cost of frame quality degradation and frame resolution, if necessary.

2.4 Game Genres for Testing

Claypool et al. **[21]**, **[22]** did a categorization for different games based on "precision" and "deadline". Generally, they classified games into three genres: avatar-first-person, avatar-third-person and omnipresent. From the left to right, the sensitivity to latency of each genre is decreasing. A delay up to 100 milliseconds can be acceptable to the avatar-first-person genre, a delay up to 500 milliseconds can be acceptable to the avatar-third-person genre and a delay up to 1000 milliseconds can be acceptable to the omnipresent genre. The work shows that different game genres have different sensitivities to latency in traditional gaming. Lee et al. **[14]** used facial electromyography (fEMG) to measure players' experiences in cloud gaming and found that different games have different susceptibilities to latency in terms of the quality of experience perceived by the player (QoE). After testing nine games among the three genres, their results present a similar latency trends in cloud gaming as traditional gaming. To evaluate the performance of Uniquitous, we selected a 3D third-person car game and a 3D third-person shooting game provided by Unity. 500 milliseconds gives a reasonable threshold how much

processing delay is acceptable in each part of Uniquitous system when testing with this specific game.

Jarschel et al. **[15]** did a survey evaluating the impact of latency in an emulated cloud gaming service on QoE. They tested three game genres (fast, medium and slow) based on Claypool's classification **[21, 22**] under three latency scenarios: 80ms, 200ms, 300ms measuring the QoE by Mean Opinion Score (MOS). From their survey results, QoE is decreasing for all three types of games when latency is increasing. By comparing QoE values of each game genre under each latency scenario, they suspected faster games are more delay-sensitive than slower ones. Chen et al. **[23]** found that with an insignificant network delay, the streaming delay (processing delay on the server plus playout delay on the client) of cloud gaming between 135 and 240 milliseconds is acceptable. We use these delay values as the reference for improving the system performance of Uniquitous.

3. Implementation

This section introduces the architecture of the Uniquitous system. First, we give a high-level overview of the architecture based on data flows of the system. Then we describe implementation details of each component in system. The architecture of Uniquitous is shown in Figure 3.

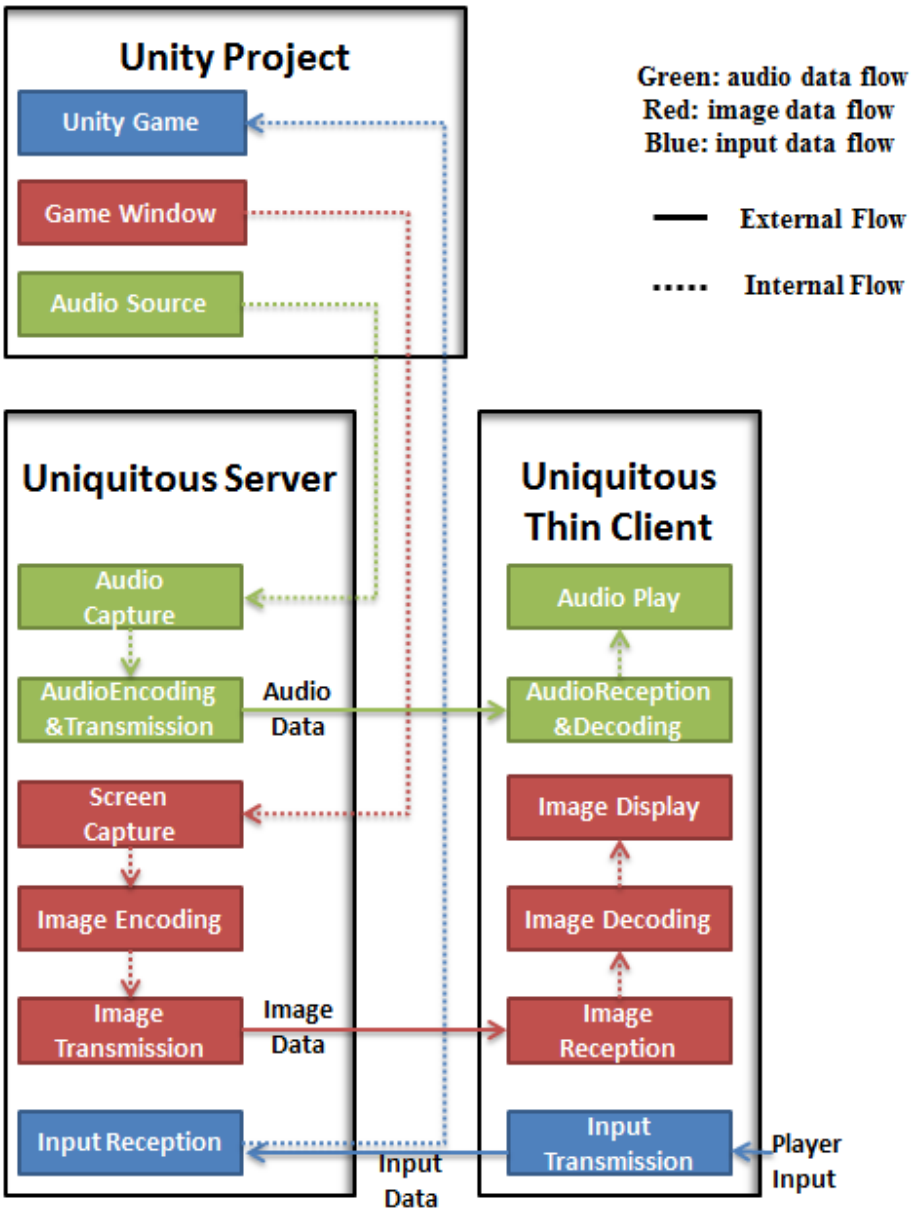

Figure 3. Architecture of Uniquitous

3.1 High-level Overview of the System

According to the architecture in Figure 3, there are three entities, which are Unity Project, Uniquitous Server and Uniquitous Thin Client. The Uniquitous Server and the Uniquitous Thin Client are set up on two different computers. Unity Project runs on the Uniquitous server and they both run within Unity. Three types of date flows are defined in the architecture:

- **Image data flow**

Game Window → Screen Capture → Image Encoding → Image Transmission → Image Reception → Image Decoding → Image Display

- **Audio data flow**

Audio Source → Audio Capture → Audio Encoding&Transmission → Audio Reception&Decoding → Audio Play

- **Input data flow**

Input Transmission → Input Reception → Unity Game

The image data flow carries data for the game frames. The audio data flow carries data for the game audio. The input data flow carries data for user input. The internal flow represents communications within Unity and the external flow represents communications between the server and the client over network.

3.2 Detailed Description of System Components

The implementation details of system components are described based on three data flows. Section 3.2.1 is for image data flow. Section 3.2.2 is for audio data flow. Section 3.2.3 is for input data flow.

3.2.1 Image Data Flow

- **Game Window**

The game window is a part of the Unity Integrated Development Environment (IDE) and works with the Unity camera to capture and display the game content to the player. The Unity camera is an essential component for rendering the game scene in Unity.

- **Screen Capture**

The Screen Capture component is used to capture the game screen from the Game Window. There are two modes of capturing the game screen: Capture with GUI and Capture without GUI. In Capture with GUI mode, *`Texture2D.ReadPixels`* is called in a coroutine to read screen pixels from the Game Window and pixel data is saved as 24-bit Texture2D format. The capture rate is automatically synchronized with the processing rate of image compression since the coroutine mechanism makes sure it will not capture a new frame until the image encoder finishes encoding the last frame. A timer is set before calling `Texture2D.ReadPixels` to make sure it captures all content at the end of each frame. In Capture without GUI mode, *`Texture2D.ReadPixel`* is called in *`OnPostRender`*. *`OnPostRender`* is an event function called after a camera finishes rendering the scene. Since the *`OnPostRender`* update rate is dependent on frame rate, this makes the screen capture rate not synchronized with the processing

rate of image compression, so the capture rate is set manually based on the processing rate of image compression.

- **Image Encoding**

We use JPEG encoder to do image encoding. The JPEG Encoder used by Uniquitous is a C# version script written by Andreas Broager from Unity community **[24]**. It is located in a namespace so it can be used by other scripts by importing the namespace. The *JPEGEncoder* constructor accepts Unity Texture2D object, quality factor and blocking option as inputs. The quality factor can be any values between 1 and 100 and higher quality factor indicates a lower compression ratio or higher quality after compression. When encoding images with blocking option set to be true, the Unity main thread will be blocked until the encoding is done. Since *JPEGEncoder* starts a new thread for image encoding and this thread takes a noticeable time to complete the encoding task for textures of large size, Uniquitous sets the blocking option to be false. This allows the JPEG encoding thread to run in a coroutine so that the Unity main thread can still run normally while the JPEG encoding thread is running.

Figure 4 illustrates how coroutine works, where one image encoding task is split into T1, T2 … Tn and each of them represents the time it takes to finish a portion of the entire task. The frequency of calling coroutine depends on the frame rate the game is running at.

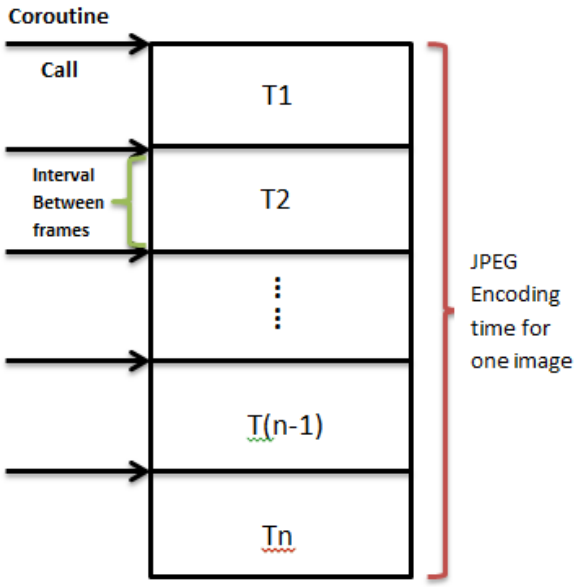

Figure 4. How coroutine works on completing JPEG encoding for one image

`JPEGEncoder` gets pixel data in Texture2D format from the Screen Capture component and compresses the date into byte arrays.

- **Image Transmission**

Remote procedural calls (RPC) provided by Unity's built-in networking system does not work well when sending big chunk of data such as an image since it causes increasing amount of delay when receiving the data on the client. So instead, Uniquitous uses a third-party networking package called uLink **[25]** to make connections between the server and the client. In image transmission, `uLink.NetworkView.UnreliableRPC` is invoked to get the byte arrays from the Image Encoding component and transmits the data via an RPC method `ReceiveImages` in the Image Reception component on the client. Both the game objects the RPC script is attached to on the client and server need to have the same Unity asset ID in order to communicate with

each other successfully. *ReceiveImages* on the client is called only when the data of a complete frame is available from the Image Encoding component.

- **Image Reception**

Image Reception is implemented by the RPC method `ReceiveImages` on the client. It gets byte arrays sent from the Image Transmission component on the server. The receiving rate is the same as the rate of invoking *uLink.NetworkView.UnreliableRPC* on the server.

- **Image Decoding**

All image data received by the Image Decoding component is processed on the frame base. We use *Texture2D.LoadImage* to load the byte array containing the data of each game frame. *Texture2D.LoadImage* can load a JPG or PNG image from a raw byte array. The byte arrays are decoded and converted back to 24-bit Texture2D format to use.

- **Image Display**

Image display component is implemented using the *GUI.DrawTexture* method and called in the *OnGUI* event function. It gets the texture passed from the Image Decoding component and draws it on the client's game window. The texture size and its position on the window screen can be adjusted by users. The image display rate is always faster than the image receiving rate because the *OnGUI* event function is called automatically by Unity several times on each frame.

3.2.2 Audio Data Flow

- **Audio Source**

Audio Source is delivered by the audio listener. The audio listener is an essential Unity component that receives all game audio in the scene and plays sounds through the computer speakers.

- **Audio Capture**

The audio Capture component is used to capture audio data from the Audio Source. We implemented Audio Capture using the *OnAudioFilterRead* function. It gets a chunk of audio data approximately every 20ms. Every chunk of audio data is an array of 2048 floats with values ranging from -1 to 1. 2048 is default buffer size of data passed into *OnAudioFilterRead*. While it is possible to change the buffer size, it is not recommended according to Unity documentation since performance will degrade when collecting audio data **[26]**. Every time *OnAudioFilterRead* passes a chunk of audio data, we convert it into an array of 8192 bytes and write the byte array into a network stream. The data is then sent to the Audio Encoding&Transmission component over a TCP socket using the localhost IP address.

- **Audio Encoding&Transmission**

The Audio Encoding&Transmission component is implemented using FFMPEG. FFMPEG is a cross-platform solution to record, convert and stream audio and video **[27]**. When audio streaming is enabled in Uniquitous, FFMPEG is started as a separate process by calling *process.Start*, setting the FFMPEG command line options which are passed to the process started. The options include settings for the input audio stream and output audio, such as the IP

address, audio sample rate, encoding bitrate, number of channels, and audio codec used. FFMPEG gets the audio stream from localhost IP address, which is sent from the Audio Capture component over a TCP socket. Then audio data is compressed with the MP3 encoder provided in FFMPEG and streamed over UDP to the Audio Reception&Decoding component on the client. The destination IP address for the output stream of FFMPEG should be the IP address of the machine the client is running on. The FFMPEG process runs in the background, so the terminal window does not show.

- **Audio Reception&Decoding**

The Audio Reception&Decoding component is implemented using FFPLAY. FFPLAY is a portable media player implemented using the FFMPEG libraries and the SDL library **[28]**. When the audio receive option is enabled on the Uniquitous client, FFPLAY is started as a separate process by calling *process.Start*, setting the command line options which are passed to the process started. FFPLAY is used to receive the audio stream sent from the Audio Encoding&Transmission component on the Uniquitous server. Then the audio stream is decoded to be played.

- **Audio Play**

The Audio Play component is implemented using FFPLAY. The decompressed audio data from the Audio Reception&Decoding component is played out in the SDL window. In our case, since FFPLAY plays audio only, the SDL window is disabled so it will not distract users from Unity game window.

3.2.3 Input data flow

- **Input Transmission**

Similar to the Image Transmission component, the Input Transmission component is implemented by unreliable RPC provided by uLink. *uLink.NetworkView.UnreliableRPC* is invoked to send input data to the RPC method *ReceiveInputs* in the Input Reception component of the server. There are three types of input data that can be sent: mouse positions, mouse clicks and keyboard strokes. Unity gives access to read all input data from the input devices. Mouse movements are translated into the coordinates (Vector3 type) of the mouse cursor on the client's screen. Mouse clicks are translated into integer value 1 or 0. Arrow keys are translated into values between -1 and 1 along the horizontal and vertical axes. Other keyboard strokes are translated into the exact values of the keys pressed.

- **Input Reception**

The Input Reception component on the server is responsible for receiving all the input data from the client and passing the data to the running Unity game on the server. Input reception is implemented using the RPC method *ReceiveInputs*. However, the interactions with the Unity build-in GUI system such as *GUI.Button* and *GUI.TextField* cannot be transferred by Uniquitous because these interactions are event based and only data on mouse positions, mouse click and keyboard strokes can be sent to the server.

Instead, we developed a customized GUI system for users to use. It has a simulated mouse cursor, a simulated button and a simulated text field. The simulate mouse cursor is implemented using a Texture2D on the server, which is drawn in real time based on the data on mouse positions sent from the client. The simulated button is implemented using *GUI.Button* on the

server. The *GUI.Button* is not used as a button but rather a component to give a response to players by changing its color. A rectangle area overlaying the button is made to detect whether the "fake" mouse cursor is within the area and whether the mouse has been clicked based on the data on mouse positions and mouse clicks sent from the client. Implementing the simulated text filed is similar to the simulated button except that when it is activated, it can enable a text input mode to start concatenating the values of the keys pressed on the client and update the text field in real time.

- **Unity Game**

The Unity Game refers to all the game scripts that are affected by user input such as game character movements and interactions with the GUI system. In order to make input sent from the client work for Unity game scripts on the server, modifications need to be made to scripts by users so that all game scripts that are affected by user input will reference only the input data passed from the Input Reception component. For example, the moving control script of the player's car in a racing game need to be modified to reference the remote arrow key values instead of the local arrow key values.

4. Micro Evaluation

This section covers micro evaluations of the Uniquitous server with two different Unity games. We divided the system into a number of subcomponents based on the architecture of the system in Figure 3 and evaluated each of them independently. The components include: Unity Project, Game Window, Screen Capture, Image Encoding, Image Transmission, Input Reception, Audio Capture and Audio Encoding&Transmission. The main parameters are the processing time of each component. In the last part, we give a estimate about the network bitrate for both the uplink and downlink of Uniquitous.

4.1 Setup

All the experiments in this part were carried out on machines in the Fossil Lab and Zoo Lab at WPI. The machines have Intel 3.4GHz i7-3770 processors and AMD Radeon HD 7700 series graphic cards, running a 64-bit Windows 7 Enterprise edition with 12 GB of RAM. The Unity game projects used for testing are provided by Unity Technologies, listed below:

a. The Car Tutorial project **[29]**. The Car Tutorial project was modified so that the car which originally is controlled by players is replaced by a car that races along a predefined series of way points.
b. The AngryBots project (Version 4.0) **[30]**. We used original version of AngryBots project for testing.

4.2 Unity Project

Unity Project is the entire block containing the Unity Game, Game window and Audio Source, as shown in Figure 3. Without setting up Uniquitous, we ran only the game project in Unity IDE for a fixed number of frames and observed the CPU performance using the unity Profiler **[31]** in the editor. The CPU time comes from the activities in the Unity IDE and Unity scripts. According to the statistics in the Profiler, the activities include rendering, scripts, physics, garbage collection, VSync and others. And we define the total CPU time of these activities as the non-Uniquitous CPU time. The Profiler can record how much time is spent on each of them on a per frame basis in terms of CPU usage. However, we did not use its deep profiling feature because its overhead is large enough to have some effect on increasing the CPU time of the scripts, which would make the evaluation result more imprecise.

Our goal in this part is to evaluate if different game screen sizes result in different overall CPU times when running the same Unity project. We changed the game screen size by manually resizing the game window in the Unity Editor. For each screen size, we collected CPU time statistics for 25 frames and computed the average value of them. Then, we got five data samples of five different screen sizes for the Car Tutorial and AngryBots respectively. The results are shown in Figure 5, where the horizontal axis indicates the number of pixels and the vertical axis represents the non-Uniquitous CPU time. The non-Uniquitous CPU time when running the Car Tutorial and AngryBots are both fluctuating around 15 milliseconds, which does not present much change when the screen size changes.

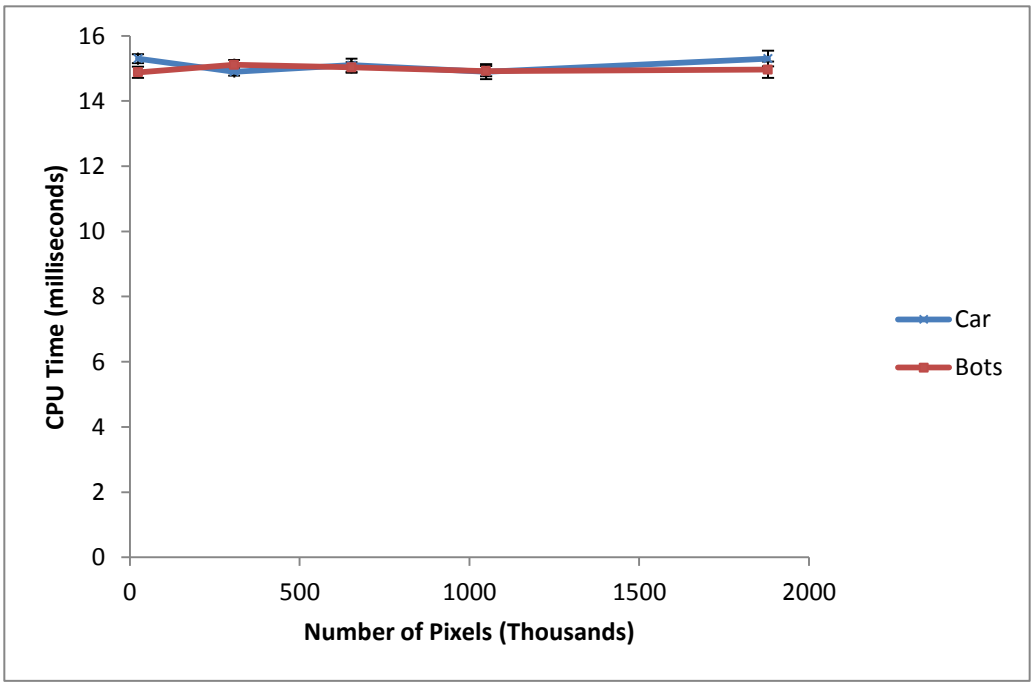

Figure 5. Non-Uniquitous CPU times for each project at five different screen resolutions

4.3 Game Window

The game window is an interface to present all content rendered by the Unity camera. The camera decides on which kind of rendering path the game scene uses. The CPU time of rendering process can also be captured by the Unity Profiler. Even though rendering time is included in the overall CPU time evaluated in section 4.2, our goal is to evaluate if different game screen sizes will result in different rendering times. For each screen size, we collected rendering time statistics for 25 frames and computed the average value of them. Then, we got five data samples of five different screen sizes for the Car Tutorial and AngryBots respectively. Figure 6 depicts the results. According to the results, for different game screen sizes, the time it takes to render the Car Tutorial game oscillates around 1.5 milliseconds while the rendering time

of the AngryBots game fluctuates between 1 and 1.2 milliseconds. Neither of the two games shows much difference in rendering time among different screen sizes.

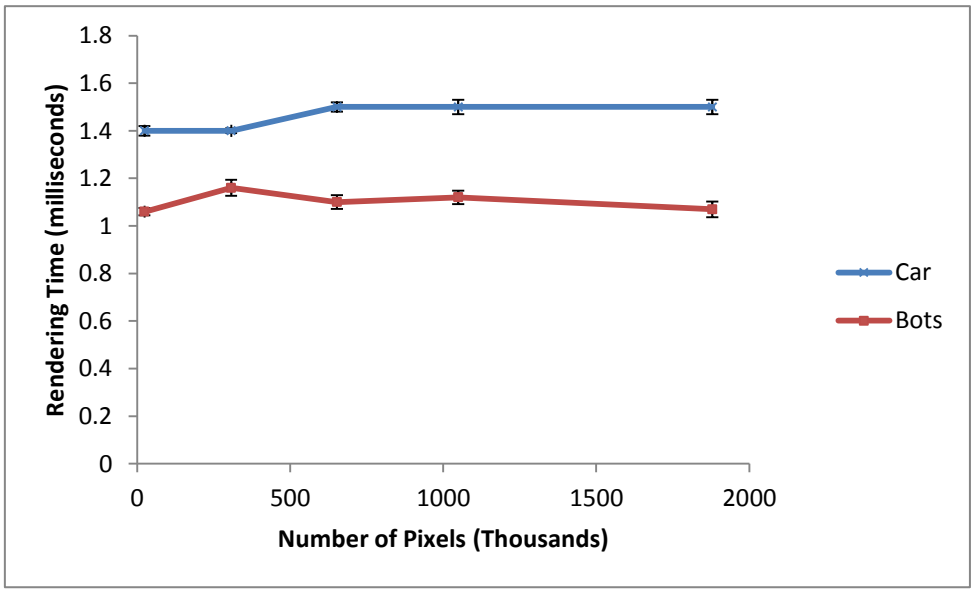

Figure 6. Rendering times for each project at five different screens resolutions

4.4 Screen Capture

In order to capture the screen to send to the client, the Screen Capture component uses `Texture2D.ReadPixels` to read screen pixels in the Unity editor game window and saves the pixel data as a 24-bit Texture2D format for further use in the compression stage. The screen capture time is defined from the time when `Texture2D.ReadPixels` starts executing to the time when pixel data in Texture2D format is available. We used `Time.realTimeSinceStartup` property to record timestamps. Timestamp T_b and timestamp T_e were put before and after screen capture process, respectively, with the time it takes to capture the screen T_e-T_b.

In order to produce consistent results across different settings, we conducted the experiment with a static image captured from the game (shown in Figure 7) instead of running the actual Car Tutorial game. The same static game image was displayed through the experiments to ensure the image content would not affect the result of our evaluation. The static image was imported into Unity as a GUI(Editor/Legacy) texture type in true color format with max size of 2048. This is to make sure the image is displayed in its original resolution. The image was displayed using *GUI.DrawTexture*.

Figure 7. Static image used for screen capture experiments

A key press event listener was initialized to capture the key event and this event was linked to the screen capture evaluation process, so we were able to trigger the screen capture evaluation process for once manually. During each round of the experiment, we evaluated the screen capture time for nine different screen resolutions. The game screen resolution was changed by resizing the Unity editor game window. The experiment for each game was repeated eight times and we calculated the average value (in milliseconds) and standard error (in Table 1 and Table 2) from the results of the eight rounds at nine different resolutions.

Resolution	Average (ms)	Standard Error
210×114	3.07	0.008
420×240	3.81	0.016
640×480	5.65	0.023
800×600	8.03	0.016
960×680	7.99	0.126
1280×720	7.28	0.376
1366×768	8.72	0.176
1680×860	8.81	0.096
1906×986	12.86	0.078

Table 1.
Average screen capture time and standard error at nine resolutions in Car Tutorial project

Resolution	Average (ms)	Standard Error
210×114	3.56	0.126
420×240	4.96	0.008
640×480	8.94	0.019
800×600	9.01	0.464
960×680	8.12	0.541
1280×720	8.02	0.389
1366×768	8.55	0.055
1680×860	10.02	0.022
1906×986	14.61	0.071

Table 2.
Average screen capture time and standard error at nine resolutions in AngryBots project

The nine screen resolutions tested are: 210×114, 420×240, 640×480, 800×600, 960×680, 1280×720, 1366×768, 1680×860 and 1906×986. They are represented by the data points in Figure 8 for the two projects respectively.

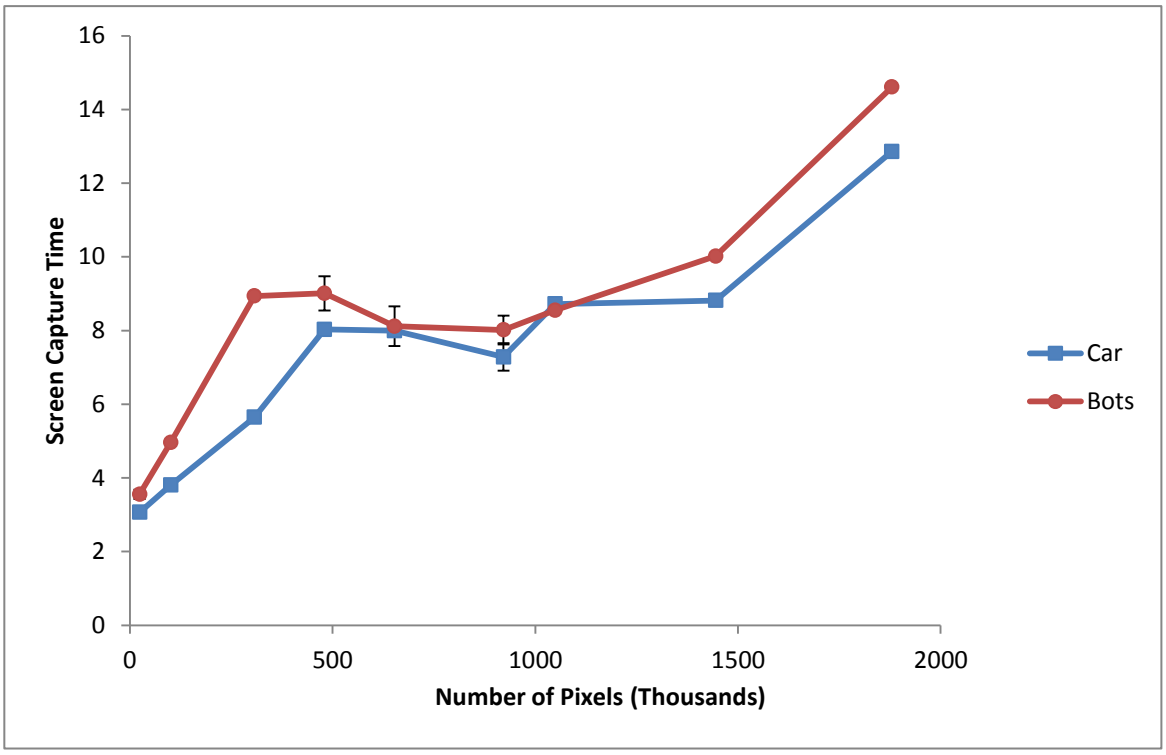

Figure 8. Screen capture times for both projects at nine different resolutions

In Figure 8, error bars were added to each data value to represent their standard errors. The data points for the Car Tutorial project generally give us a regular trend except for the resolution of 1280×720. The screen capture time in Car Tutorial project is ranging from 3 milliseconds to 13 milliseconds. The data points for the AngryBots project presents a general increasing trend except at 800×600, 960×680, 1280×720, where the range of error is noticeable. The screen capture time in AngryBots project ranges from 3 milliseconds to 15 milliseconds. Based on the screen capture results from the two projects, the processing time of screen capture in the Unity game window increases as the resolution increases. And given the same resolution, screen capture time can vary across different Unity projects.

Alternative Methods:

In order to minimize the processing time for screen capture, we also did a comparison evaluating the processing time of two other methods of doing screen capture and comparing the results of these two methods with the result of method 1. We did the evaluation using the Car Tutorial project.

In the second method, we used the *Application.CaptureScreenshot* method to do screen capture. *Application.CaptureScreenshot* captures the screen from the Unity game window and save it as a PNG file to local disk automatically. In order to access the screenshot image data, the *WWW.LoadImageIntoTexture* method was used to read image data from disk into pixel data in Texture2D format in real time. And the screen capture time is defined as the time difference between the time when *Application.CaptureScreenshot* begins executing and the time when the pixel data in texture2D format is ready to use. The processing time was measured using timestamps as in method 1.

In the third method, the only difference from the third method is that we called `File.ReadAllBytes` to read screenshot image data from disk and the screen capture time is defined as the time difference between the time when `Application.CaptureScreenshot` begins executing and the time when the byte array is ready to use. The processing time was measured using timestamps as in method 1.

In the comparison experiment, we used an image sample downloaded from GentsideDécouverte Website [32] and displayed it in full size on the game screen of the Unity editor. The experiment was repeated five times for three different resolutions. We used the average value of the results to do the comparison for each resolution. Figure 9 shows the differences between processing time for the three different methods to capture game screen.

We name the method used for the screen capture in the Uniquitous as Method 1, the second method as Method 2 and the third method as Method 3. Method 1 takes the minimum process time and is used for capturing game screen in Uniquitous.

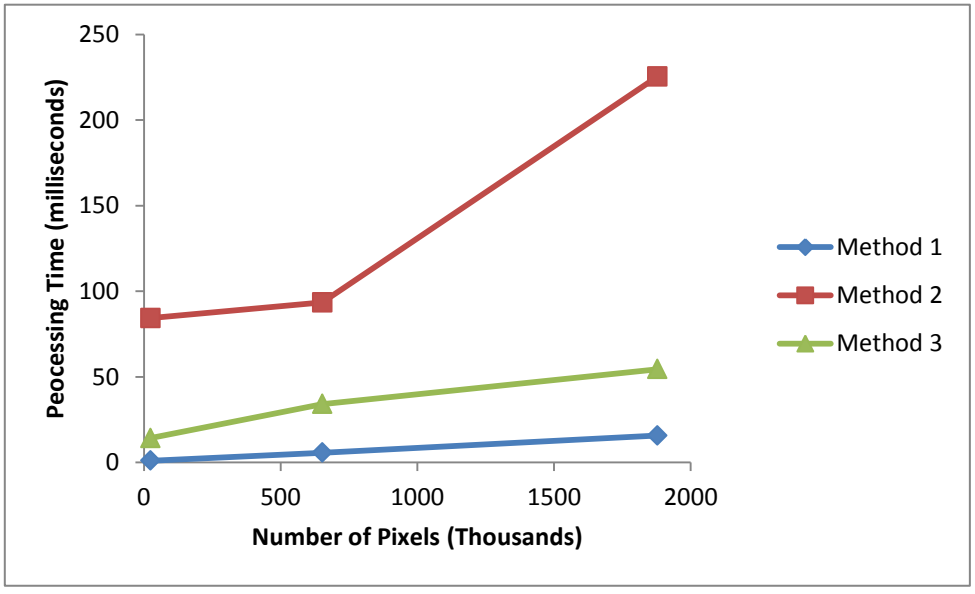

Figure 9.
Comparison of the processing time among three methods when the game screen is 210x114, 960x680 and 1906x986

4.5 Image Encoding

The processing time of the image encoder puts a limit on how fast the Uniquitous server processes each frame and thus controls the sending frame rate. The frame rate of the game for the players on the client is the same as the sending rate on the server.

We evaluated the JPEG encoding time by adding time stamps before and after the JPEG encoding. When the encoding starts, *Datetime.Now* method was called to record the beginning time Tb, and the end time, $T_{e,}$ was recorded as soon as the encoding thread signaled that it was done. We define T_e-T_b as the time it takes to do image encoding.

At the application level, the JPEG encoding time depends on the amount of pixel data stored in the image. There are two independent variables we can adjust to affect image size: quality factor (Q) and game window size (R). We conducted 72 experiments for each of the two projects by using different combinations of the quality factor and the game window size shown in tables 3 and 4.

Quality Factor(Q)	1	5	10	20	40	60	80	100

Table 3. Different JPEG quality factors for testing

Game Window Size(R)	210 by 114	420 by 240	640 by 480	800 by 600	960 by 680	1280 by 720	1366 by 768	1680 by 860	1906 by 986

Table 4. Different game resolutions for testing

4.5.1 Car Tutorial

For each round of experiment, the racing car moved from start point to end point along the same path and we recorded the number of frames processed by the JPEG encoder and recorded

the processing time for each frame. Then, we calculated average and standard deviation of the encoding time for each round and used these values to draw graphs in Figure 10 to Figure 17 for analysis.

We set up different scenarios based on different JPEG quality factors and observed what effect the game window size had on JPEG encoding time with quality factor fixed in each scenario. From Figure 10 to Figure 17, the encoding time for different game window sizes were plotted against the number of pixels. The results show that the encoding time is increasing linearly with the number of pixels.

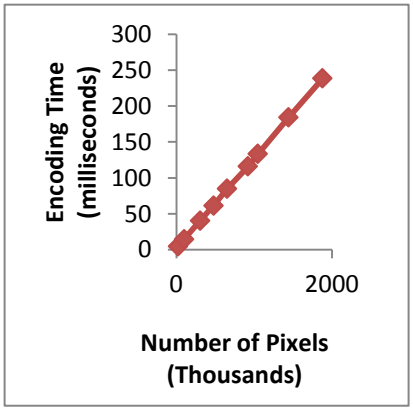

Figure 10. Q=1

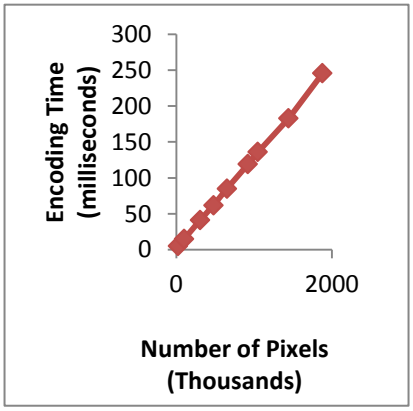

Figure 11. Q=5

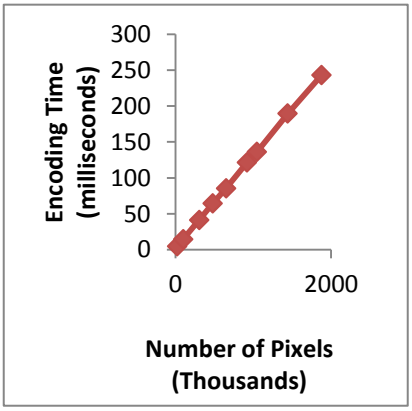

Figure 12. Q=10

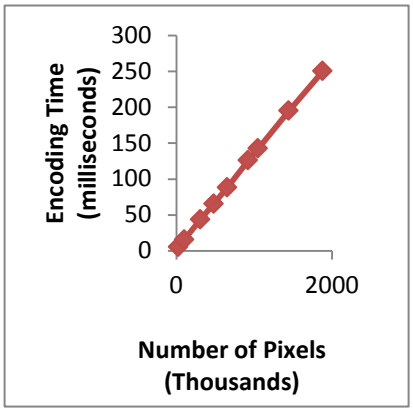

Figure 13. Q=20

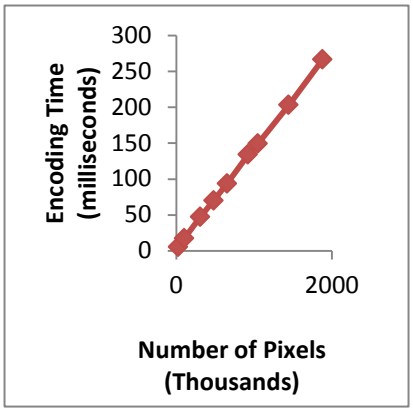

Figure 14. Q=40

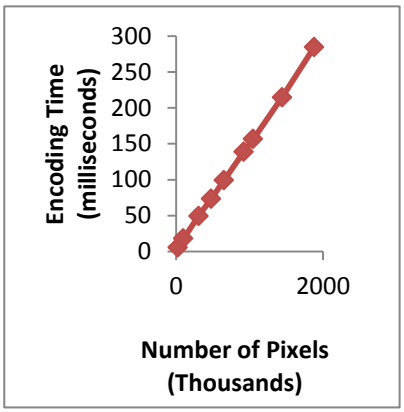

Figure 15. Q=60

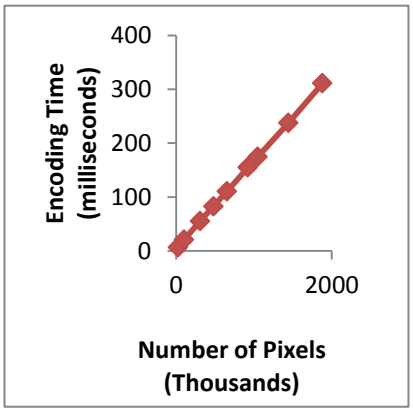

Figure 16. Q=80

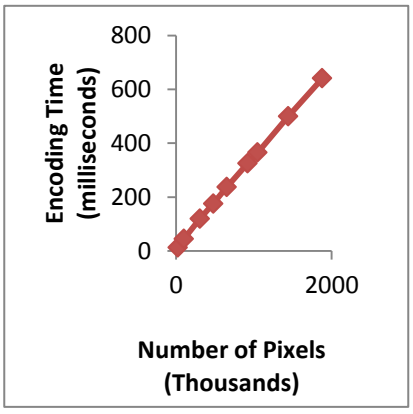

Figure 17. Q=100

Similarly, we set up different scenarios based on different game window resolutions and observed what effect the quality factor had on JPEG encoding time with the resolution fixed in each scenario. The results in Figure 18 to Figure 26 show that the encoding time overall increases with the quality factor. An increasing quality factor increases the encoding time gradually when quality factor is lower than 80. After a quality factor of 80, the encoding time increases rapidly for all scenarios.

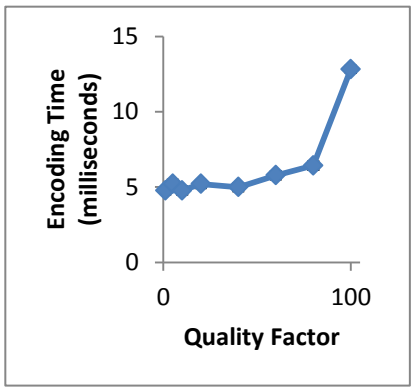

Figure 18. R is 210×114

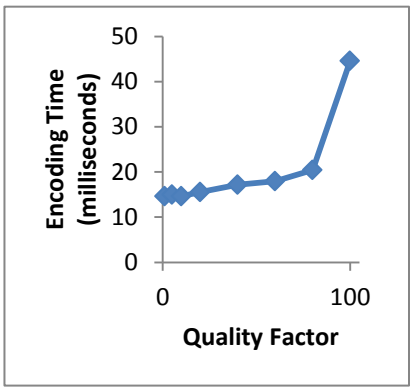

Figure 19. R is 420×240

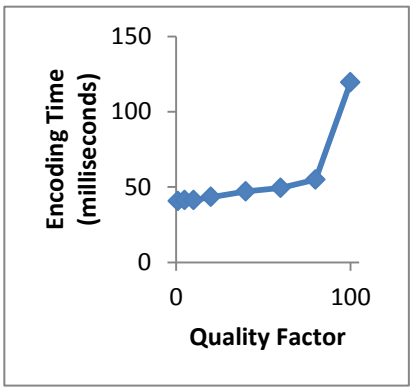

Figure 20. R is 640×480

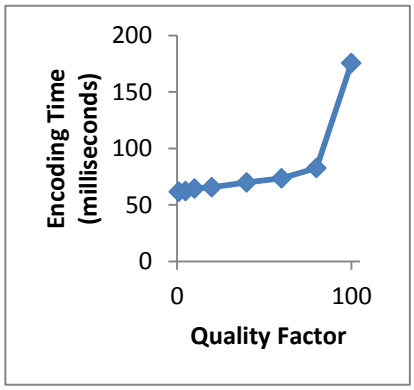

Figure 21. R is 800×600

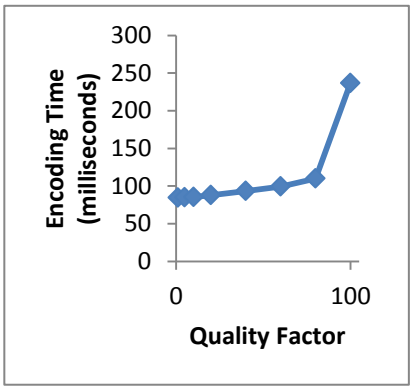

Figure 22. R is 960×680

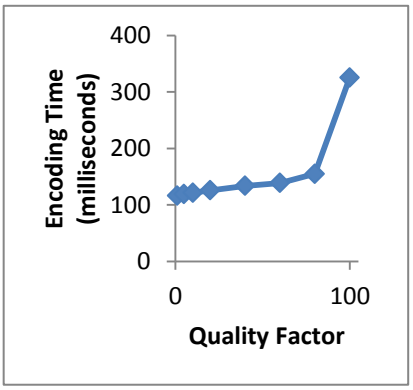

Figure 23. R is 1280×720

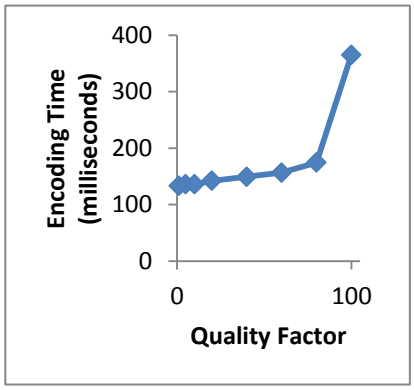

Figure 24. R is 1366×768

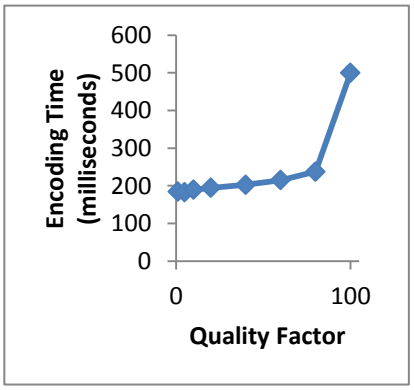

Figure 25. R is 1680×860

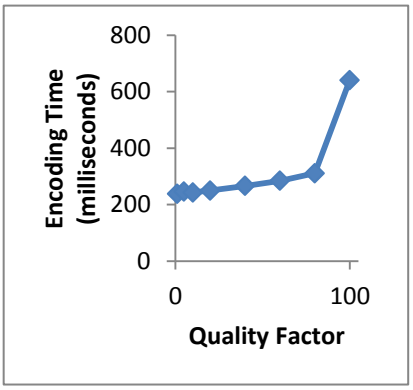

Figure 26. R is 1906×986

Putting the results together, either higher quality factor or higher game resolution makes the plot line shift further upward along the encoding time axis. In Figure 27, the range between encoding time value of the smallest resolution and encoding time value of the largest resolution is expanded. The time range is a lot higher when the quality factor is 100, which is about two times wider than the others. Comparing results from Figure 27 with Figure 28, changing resolutions has a larger impact on JPEG encoding time than changing the quality factor (between 1 and 80) since the slope of the lines in Figure 27 is larger than the slope of lines in Figure 28. This indicates changing game resolution in Uniquitous is a more effective way to decrease processing time when running the Car Tutorial game.

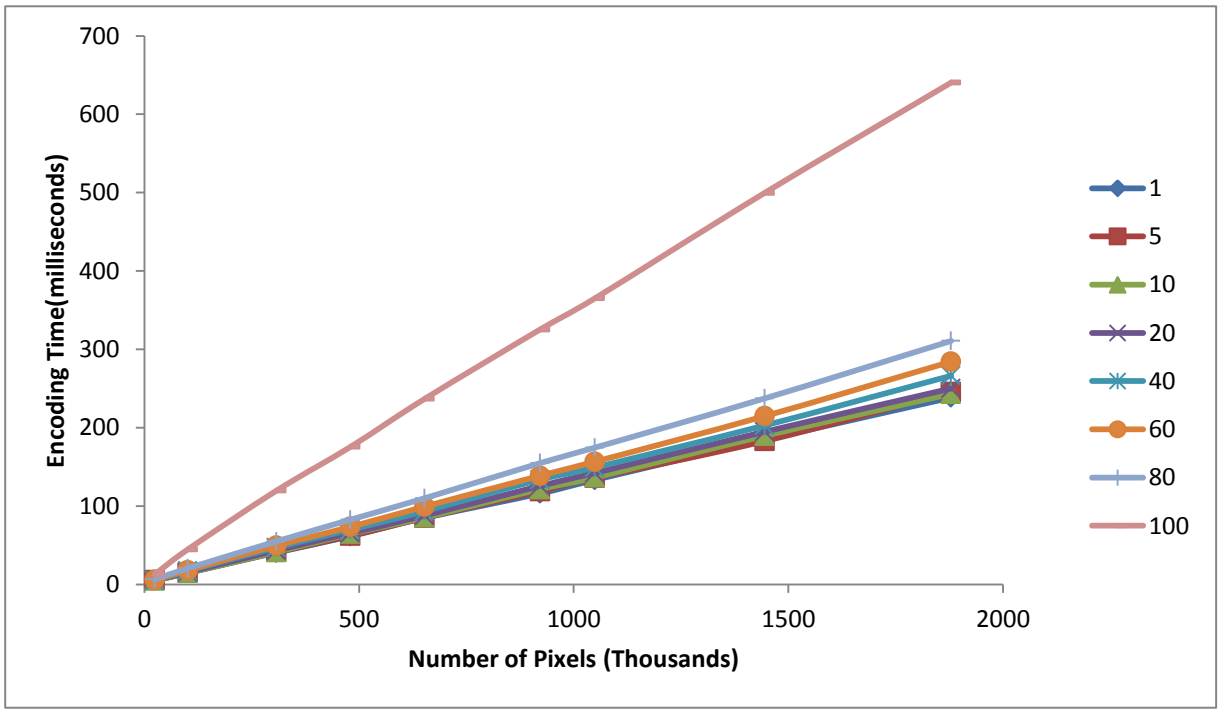

Figure 27. Comparisons of encoding time among different JPEG quality factors in Car Tutorial

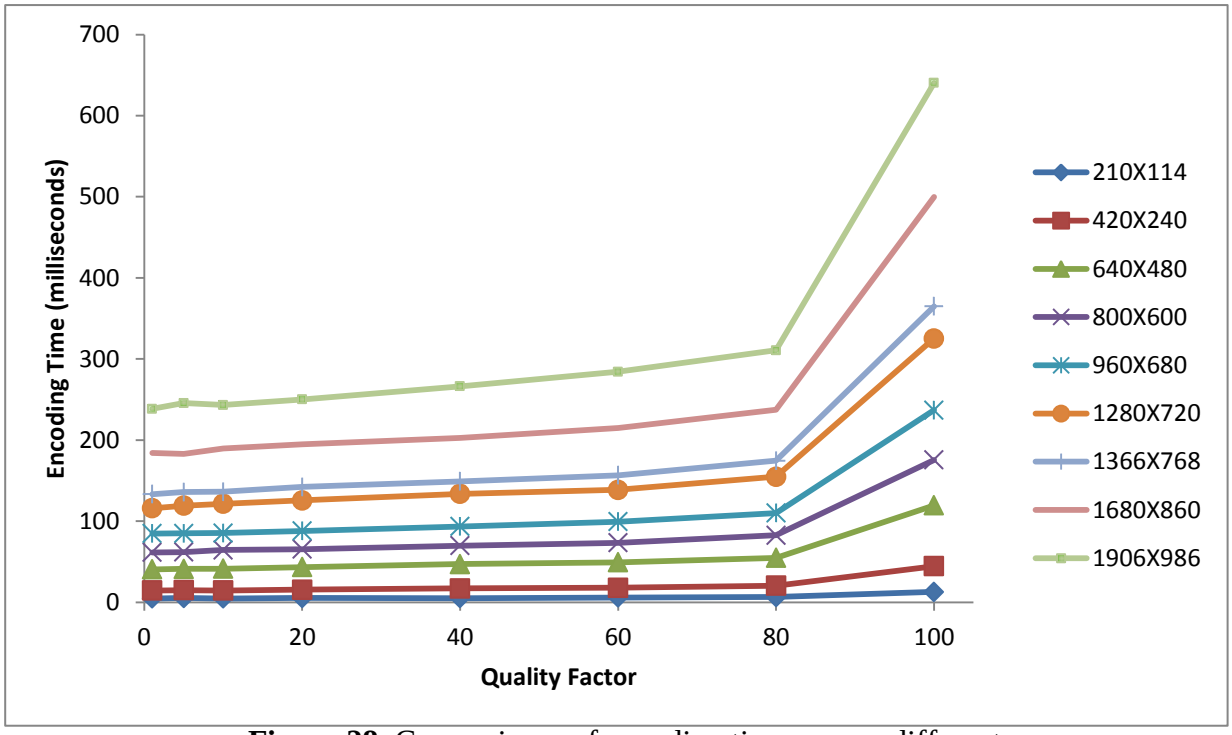

Figure 28. Comparisons of encoding time among different game window sizes in Car Tutorial

4.5.2 AngryBots

For each round of the experiment, the main character was moved from start point to end point along the same path and we recorded the number of frames processed by JPEG encoder and recorded the time of processing each frame. Then we calculated average and standard deviation of the encoding time for each round and used these values to draw graphs in Figure 29 to Figure 36 for analysis.

We set up different scenarios based on different JPEG quality factors and observed what effect the game window size has on JPEG encoding time with quality factor fixed in each scenario. From Figure 29 to Figure 36, the encoding time for different game window sizes were plotted against the number of pixels. The results show that the encoding time increases linearly with the number of pixels.

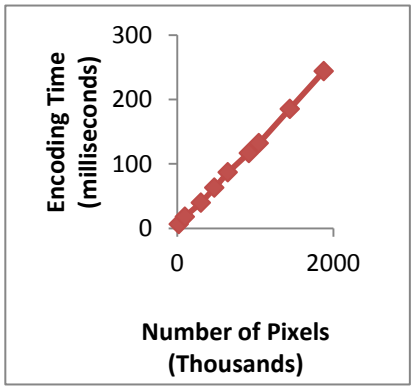

Figure 29. Q=1

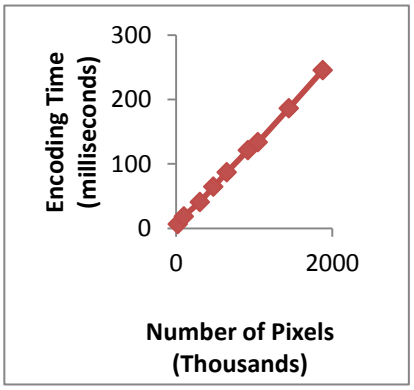

Figure 30. Q=5

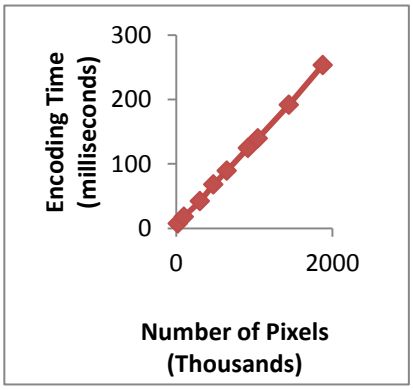

Figure 31. Q=10

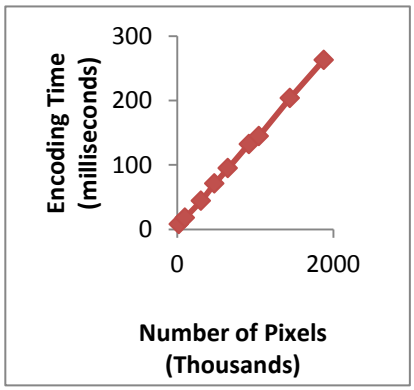

Figure 32. Q=20

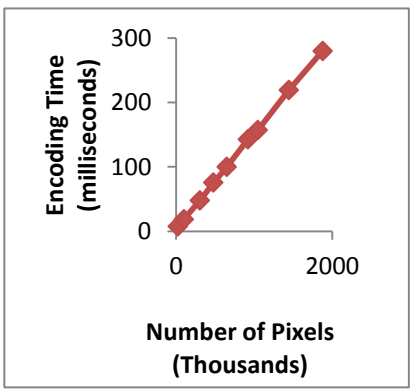

Figure 33. Q=40

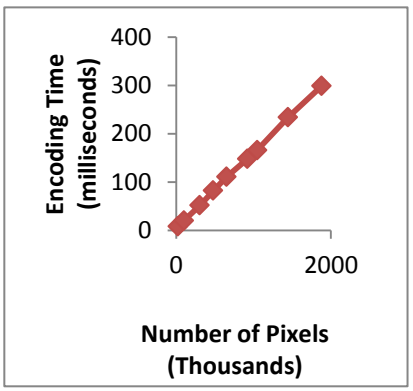

Figure 34. Q=60

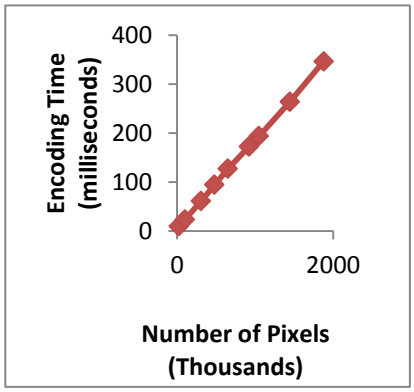

Figure 35. Q=80

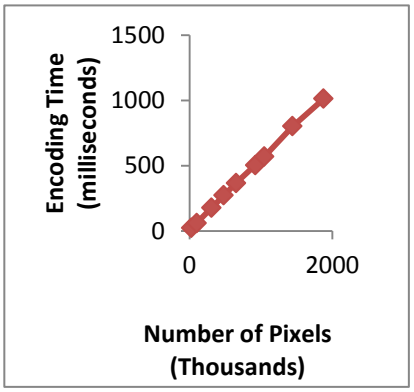

Figure 36. Q=100

Then, we set up different scenarios based on different game resolutions and observed what effect the quality factor had on JPEG encoding time with the resolution fixed in each scenario. The results in Figure 37 to Figure 45 show that the encoding time overall increases with the

quality factor, Increasing the quality factor increases the encoding time gradually when the quality factor is lower than 80. After quality factor of 80, it increases rapidly for all scenarios.

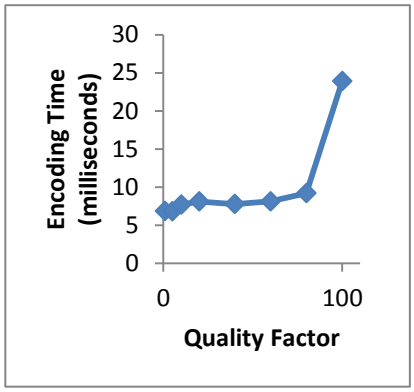

Figure 37. R is 210×114

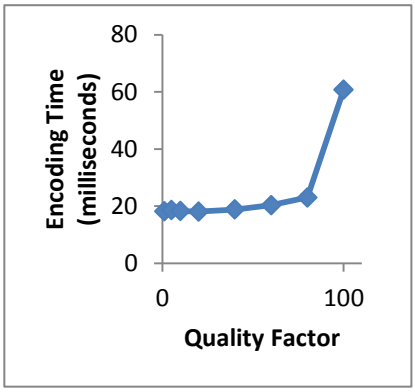

Figure 38. R is 420×240

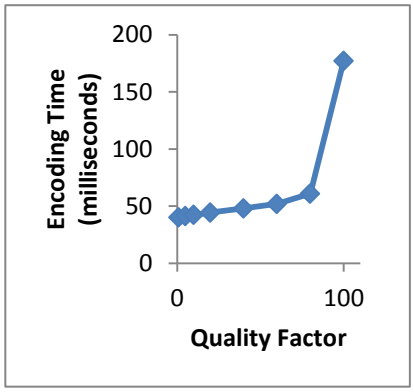

Figure 39. R is 640×480

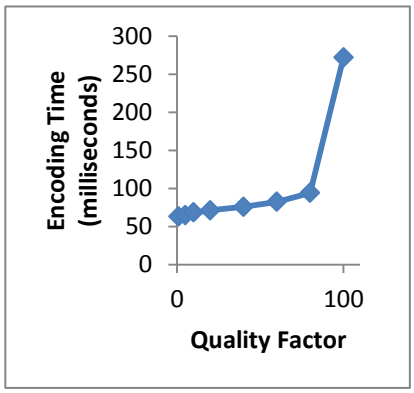

Figure 40. R is 800×600

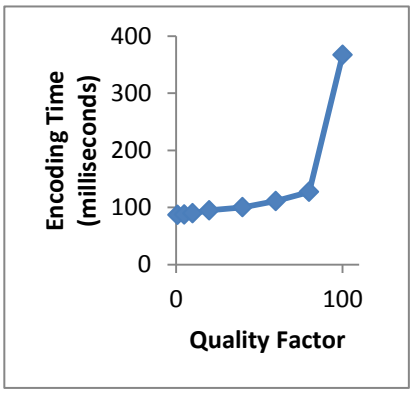

Figure 41. R is 960×680

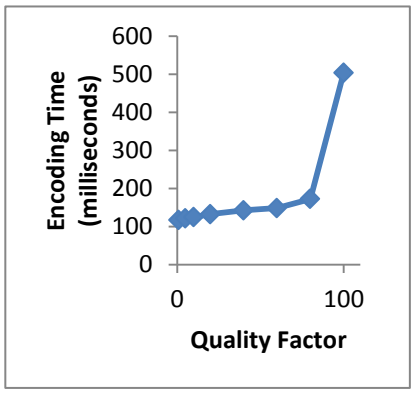

Figure 42. R is 1280×720

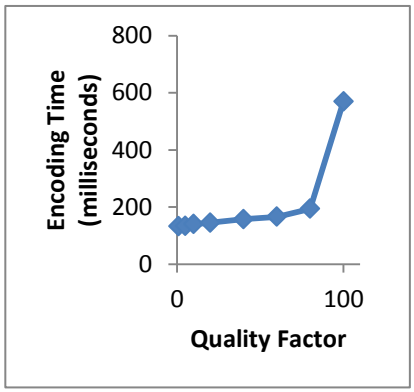

Figure 43. R is 1366×768

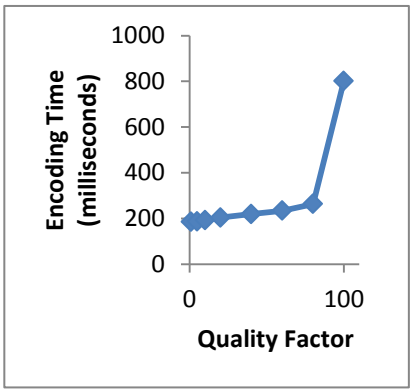

Figure 44. R is 1680×860

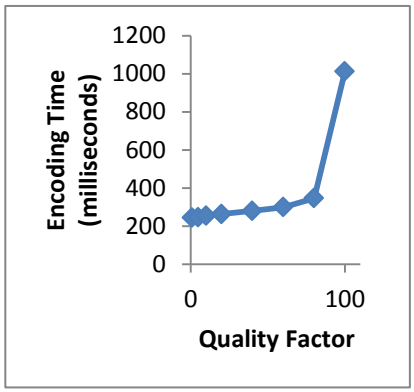

Figure 45. R is 1906×986

Each higher quality factor or higher game resolution makes the plot line shift further upward along the encoding time axis. In Figure 46, the range between encoding time value of the smallest resolution and encoding time value of the largest resolution is expanded, showing the processing time is a lot higher when the quality factor is 100, a gap about two times wider than the others. Comparing results from Figure 46 with Figure 47, changing resolutions has a larger impact on JPEG encoding time than changing the quality factor (between 1 and 80) since the slope of the lines in Figure 46 is larger than the slope of lines in Figure 47. This indicates changing the game resolution in Uniquitous is a more effective way to decrease processing time when running the AngryBots game.

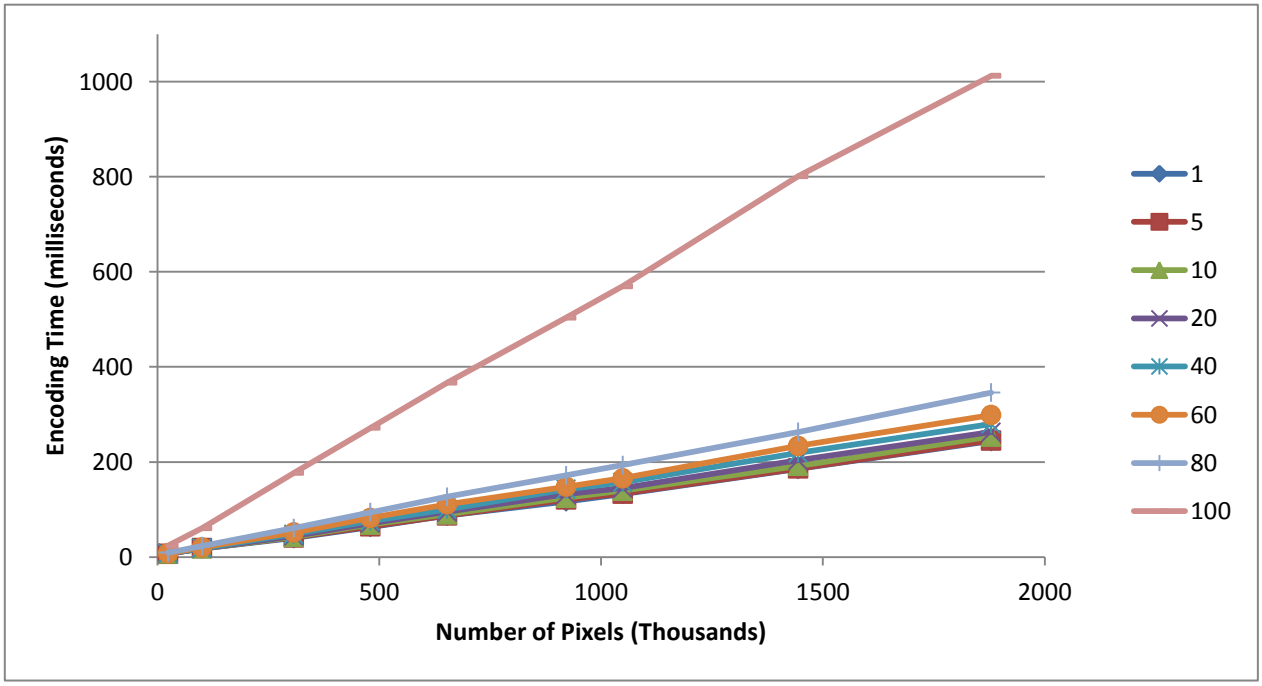

Figure 46. Comparisons of encoding time among different JPEG quality factors in AngryBots

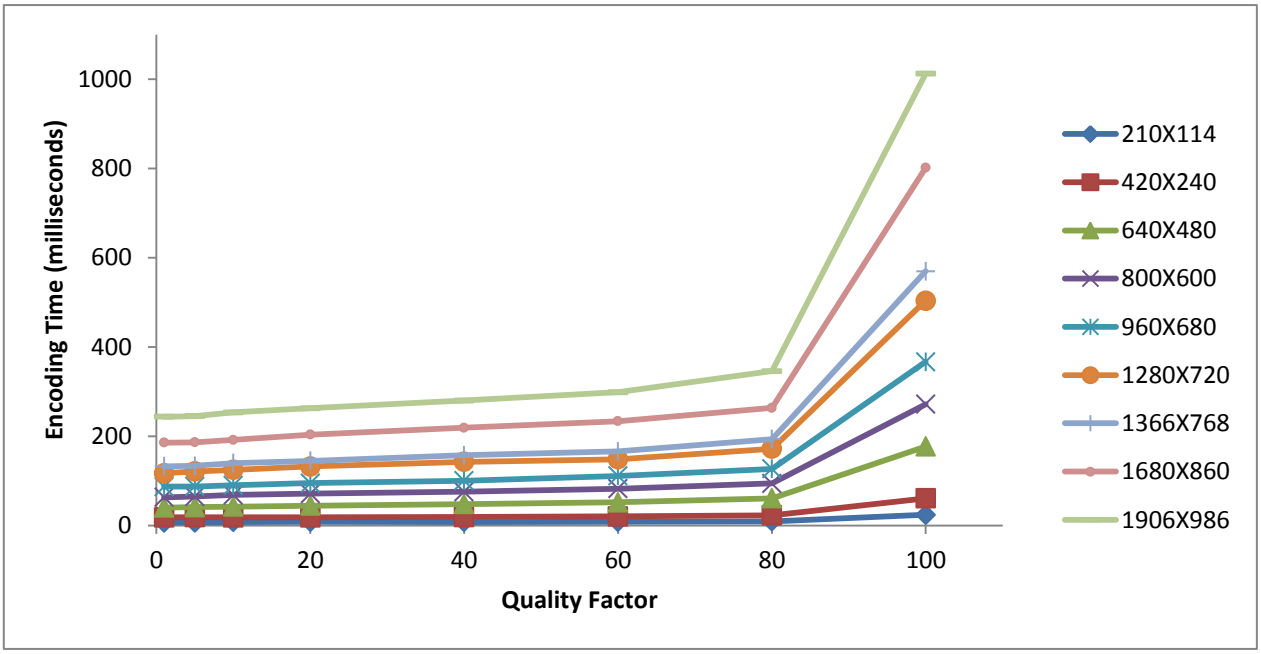

Figure 47. Comparisons of encoding time among different game window sizes in AngryBots

4.6 Image Transmission

The Image transmission time is the time it takes RPC function to process the compressed game image data and send out the data. Since the image transmission deals only with compressed image data of different sizes, the evaluation results can be applied to both game projects tested.

After the game image is encoded by the JPEG encoder, the pixel data stored in the byte array output from the encoder are sent through unreliable Remote Procedural Call (RPC) provided by uLink. We ran a script initializing a byte array of data and invoking RPC reading and transmitting the byte array on the Uniquitous server. The size of the byte array in each round was varied from 0 byte to 65480 bytes. Meanwhile, the Unity profiler was used to measure the CPU time of the RPC with RPC running in the `Update` function for several seconds in each round. RPC is called once on each frame and we collected 20 RPC processing times from the 20

frames in each round. We computed the average of the processing time for each round and produced the graph in Figure 56. The processing time of RPC increases linearly from 0.01 milliseconds to 6.02 milliseconds with an increasing in the size of the data transmitted. Note the maximum data size transmitted cannot exceed 65480 bytes because the length of UDP datagram is a 16-bit integer and has to include the length of UDP header.

Due to limited interpretation in Unity documentation in what exactly the CPU time in Profiler represents, it is unclear that whether the time includes only the CPU time when executing RPC in user space or the CPU time when executing RPC in both user space and kernel space. However, based on all experiment results of measuring send processing times for various operations in kernel space **[18]**, we can infer the processing time of sending and receiving UDP packets in kernel space less than 1 millisecond and probably does not significantly affect the accuracy of the transmission times measured.

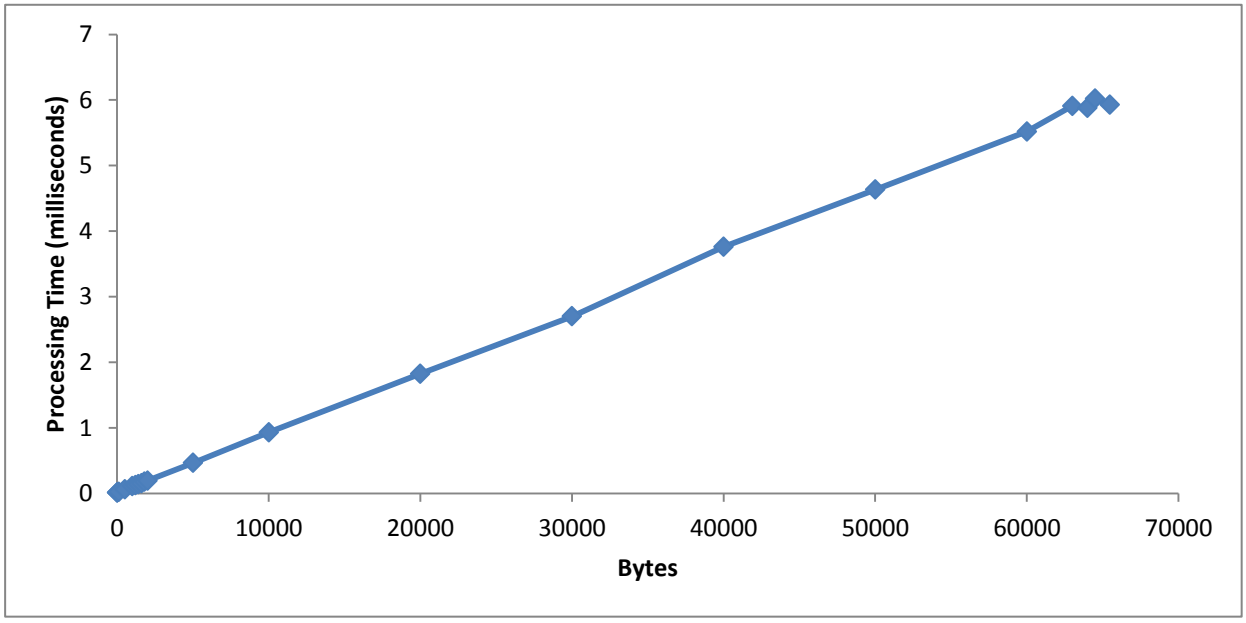

Figure 48. CPU time of transmitting data of different sizes for RPC

The experiment results in this part are also applied to input transmission process on the client.

4.7 Input Reception

The processing time of the Input Reception component is the time it takes the RPC function to receive and process the control input data. The amount of data received by unreliable RPC depends on the players' interactions with Unity games and varies among different game genres and different players. However, the most common control inputs are mouse movements, mouse clicks, keyboard strokes and textfield content. They are represented by strings, integers and Boolean values in Unity and can be converted into bytes by RPC for transmission. The data length of these inputs is far less than 1000 bytes. The experiment results **[18]** shows that the trend of accumulated receive side operation time over different data sizes is similar to the accumulated send side operation time. We can assume the processing time of receiving 1000 bytes of data takes approximately 0.1 milliseconds by referencing the processing time of sending the data at 1000 bytes in Figure 48. The control input data takes less than 0.1 milliseconds to process, an insignificant amount. Thus, we do not further consider the processing time for receiving game control inputs on the Uniquitous server.

4.8 Audio Capture

Unlike the image capture process, the audio capture process does not occur in the Unity main thread. Instead, it runs in a separate thread on an audio engine used by Unity. We implemented the *OnAudioFilterRead* function to capture the data of audio which is being played in game and stored in a buffer. We were not able to use the Unity Profiler observe the processing time of the *OnAudioFilterRead* function since the Profiler only captures information on the code running in the Unity main thread. The *OnAudioFilterRead* passes

2048 floats at a time with the audio sample rate of 44.1 KHz. It returns 23 milliseconds worth of data at a time. The default buffer size of data passing into the *OnAudioFilterRead* function is 2048 and while it is possible to change the size, it is not recommended according to Unity documentation due to performance degradation **[26]**.

4.9 Audio Encoding & Transmission

Uniquitous writes the audio data stored in buffer to a network stream and sends it through a TCP socket to a localhost IP address, used by FFMPEG to accept input audio stream. The time it takes to send data to localhost is negligibly small. So we focused on evaluating audio processing time of FFMPEG. The audio processing time (Ts) includes the time of reading the input stream, the time of audio compression and the time of transmitting the compressed audio data. Ts was measured outside of Unity by running FFMPEG from the command line. We used the Unix command "time" to get timing statistics about running FFMPEG in the Cygwin terminal window. Since we ran FFMPEG independently of Unity, we were not able to capture the audio stream from a Unity game project in real time. Instead we let FFMPEG read audio files already saved on disk rather than reading a network stream from the IP address. These audio files used for measurement were captured from the Car Tutorial and saved as uncompressed audio files in PCM format using FFMPEG.

We saved eight audio files of different sizes (size numbers are shown in Table 4) and for each size, we used the "time" command to run FFMPEG to read the file and stream it to the Uniquitous client 10 times. The "time" command runs a process and upon terminating reports the real time, user time and system time consumed by the process. Real time is the elapsed real time

between invocation and termination. User time is the CPU time of the program executing in user space. System time is the CPU time of the program executing in kernel space. So the sum of user time and system time is Ts, which is the total CPU time of FFMPEG encoding and transmitting audio data.

Audio File Size (KB)	4192	7480	11280	14984	18728	30264	33872	36288

Table 5. Audio data size values for testing

Figure 49 shows the contributions of system time and user time over the total time (Ts). From the total accumulated time for each input file size in Figure 49, changing size of the input data to FFMPEG does not significantly change the length of the processing time.

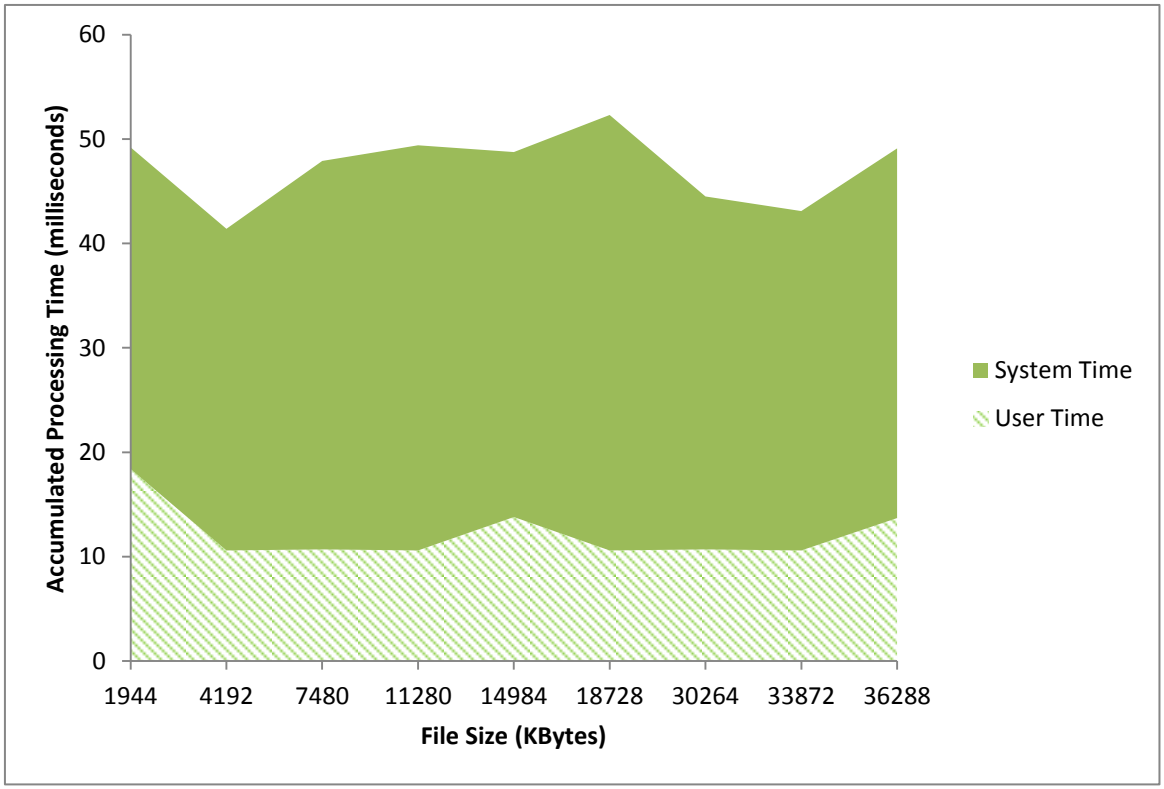

Figure 49. Contributions of system time and user time over the time for FFMPEG to process audio data of different sizes

4.10 Network Estimate

In order to explore the relationship between the frame rate and the network bitrate, we used the measured results from the micro evaluations to estimate the downlink bitrate for different frame rates. The bitrate is computed by multiplying the frame rate with the frame size after compression. With a recommended encoding quality factor of 20, we selected 7 data samples at different resolutions for the Car Tutorial and 5 data samples at different resolutions for the AngryBots. Based on the calculated bitrates, we plotted the graphs in Figure 50 and 51. In the graph, the horizontal axis represents the frame rate and the vertical axis represents the bitrate in Mbps. From Figure 50, for the Car Tutorial, the maximum bitrate is 2.96 Mbps when the resolution is 1366×768 and the frame rate is 6 fps. The minimum bitrate is 1.13 Mbps when the resolution is 210×114 and the frame rate is 45.7 fps. From Figure 51, for the AngryBots, the maximum bitrate is 3.59 Mbps when the resolution is 960×680 and the frame rate is 8.4 fps. The minimum bitrate is 1.11 Mbps when the resolution is 210×114 and the frame rate is 33.1 fps.

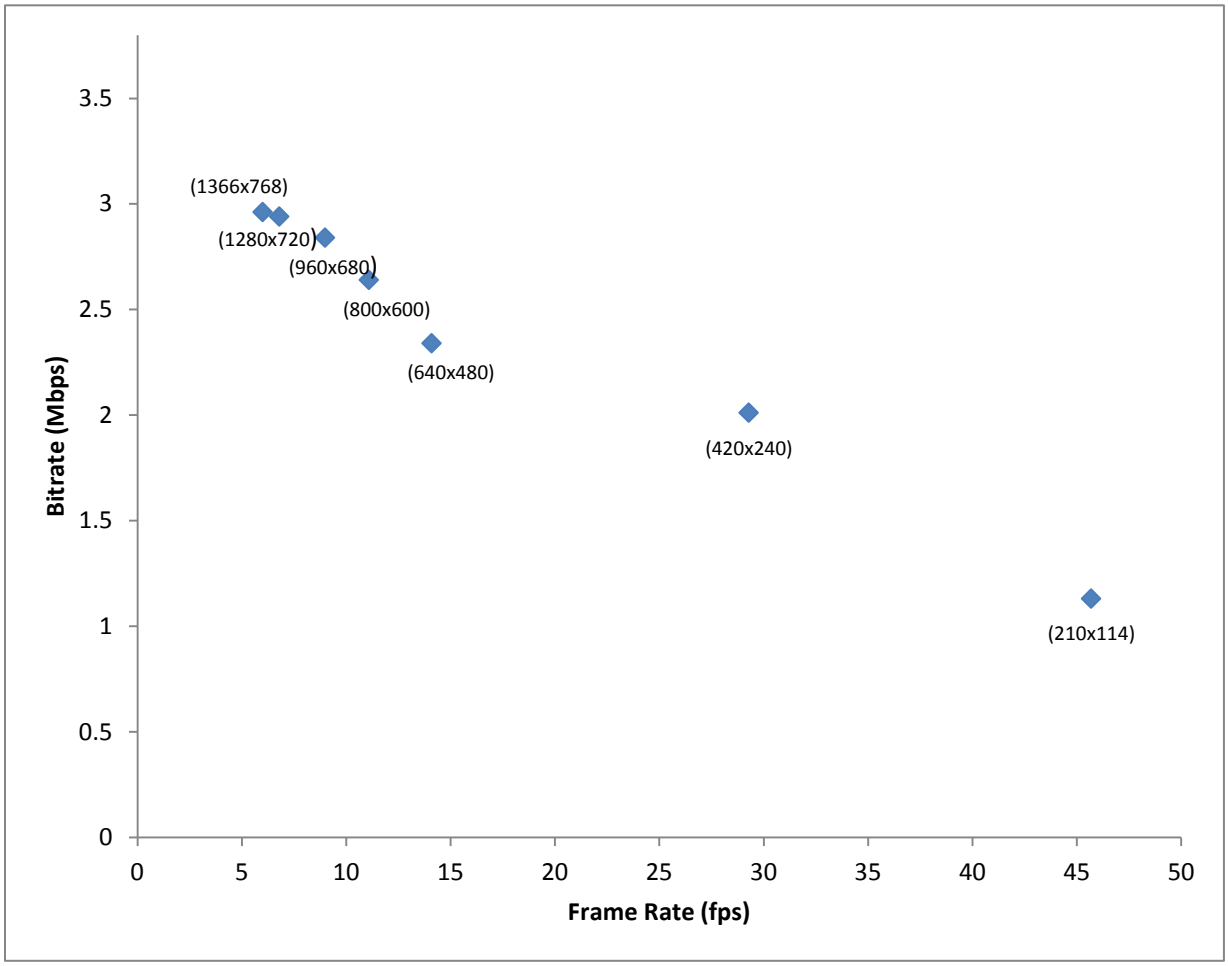

Figure 50. Network bitrate versus frame rate when JPEG encoding factor is 20 (Car Tutorial)

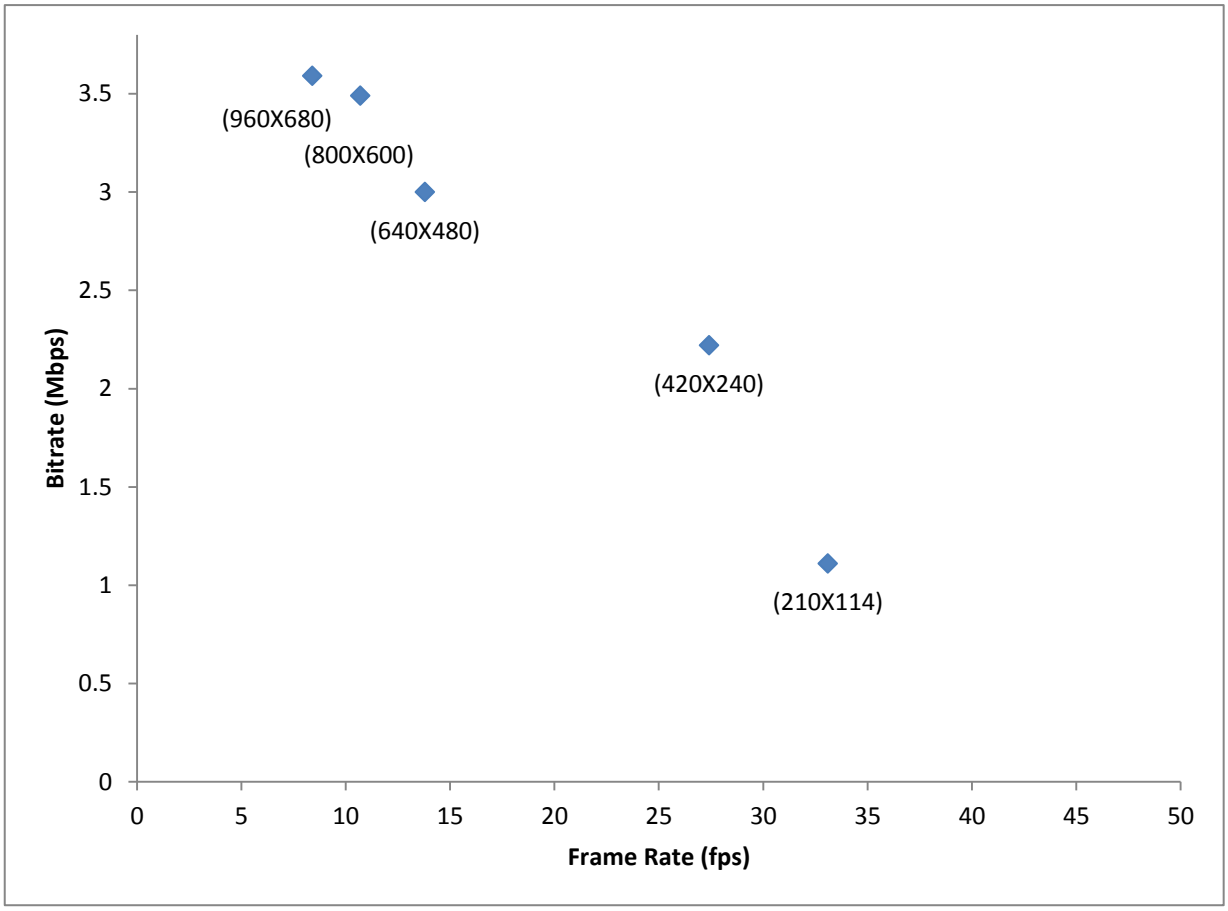

Figure 51. Network bitrate versus frame rate when JPEG encoding factor is 20 (AngryBots)

With a recommended resolution of 640×480 and different JPEG encoding quality factors, we selected 5 data samples for the Car Tutorial and 6 data samples for the AngryBots. Based on the calculated bitrates, we plotted the graphs in Figure 52 and 53. In the graph, the horizontal axis represents the frame rate and the vertical axis represents the bitrate in Mbps. From Figure 52, for the Car Tutorial, the maximum bitrate is 4.58 Mbps when the quality factor is 60 and the frame rate is 13.7 fps. The minimum bitrate is 1.15 Mbps when the quality factor is 1 and and the frame rate is 14.4 fps. From Figure 53, for the AngryBots, the maximum bitrate is 5.79 Mbps when the

quality factor is 60 and the frame rate is 13.1 fps. The minimum bitrate is 1.35 Mbps when the quality factor is 1 and the frame rate is 16.3 fps.

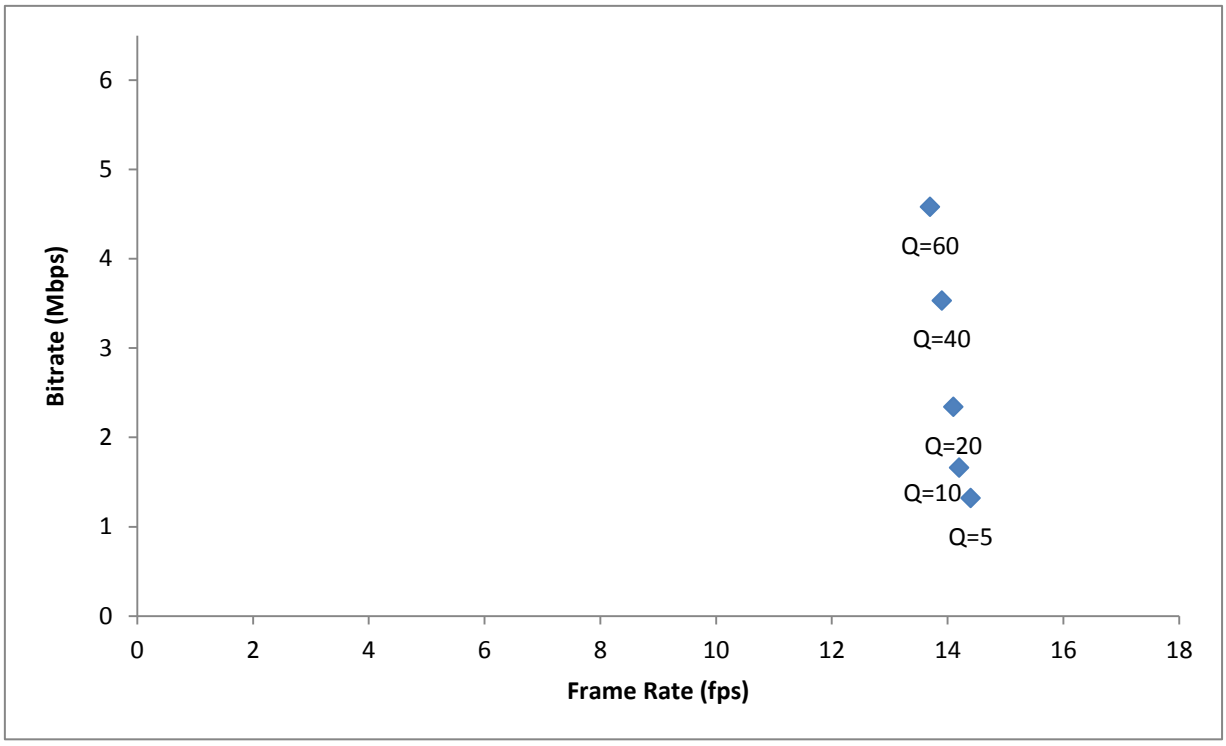

Figure 52. Network bitrate versus frame rate when the resolution is 640×480 (Car Tutorial)

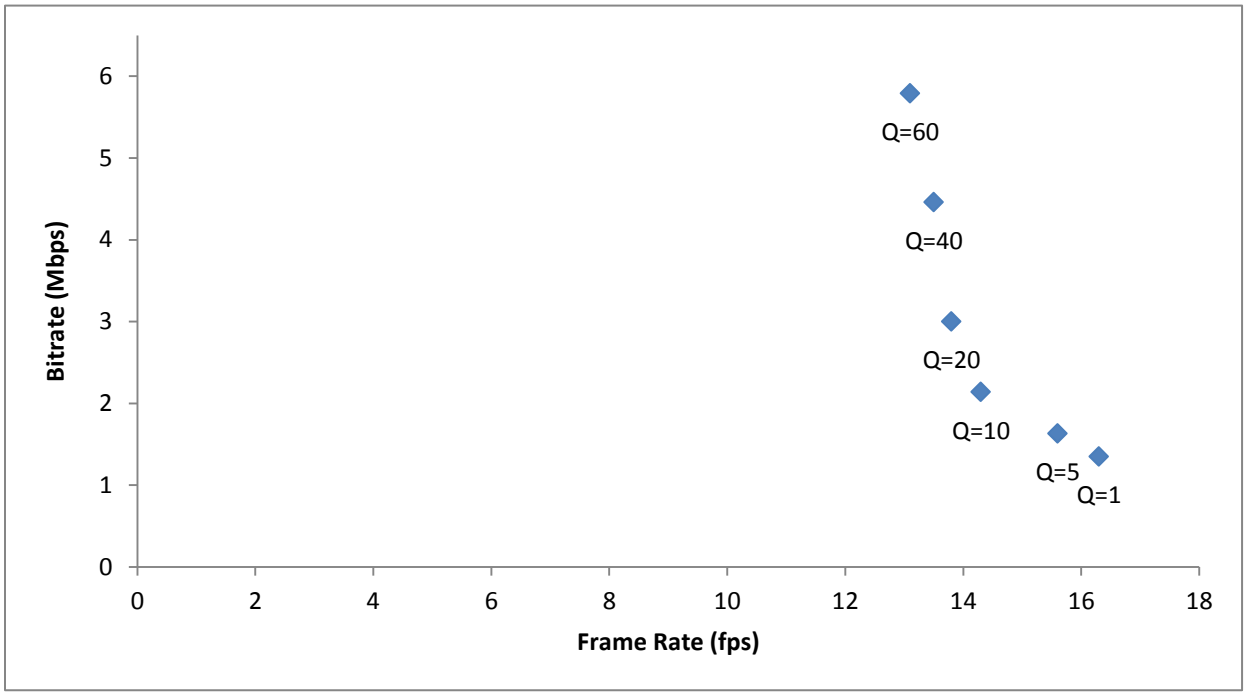

Figure 53. Network bitrate versus frame rate when the resolution is 640×480 (AngryBots)

Based on the measured frame size for previous data samples, we plotted Figure 54 and 55 to predict the trend of network bit rate changes versus the frame rate for different settings of game resolution and game image quality. The results of both projects indicate that when the frame size is fixed, increasing frame rate linearly increases the network bitrate.

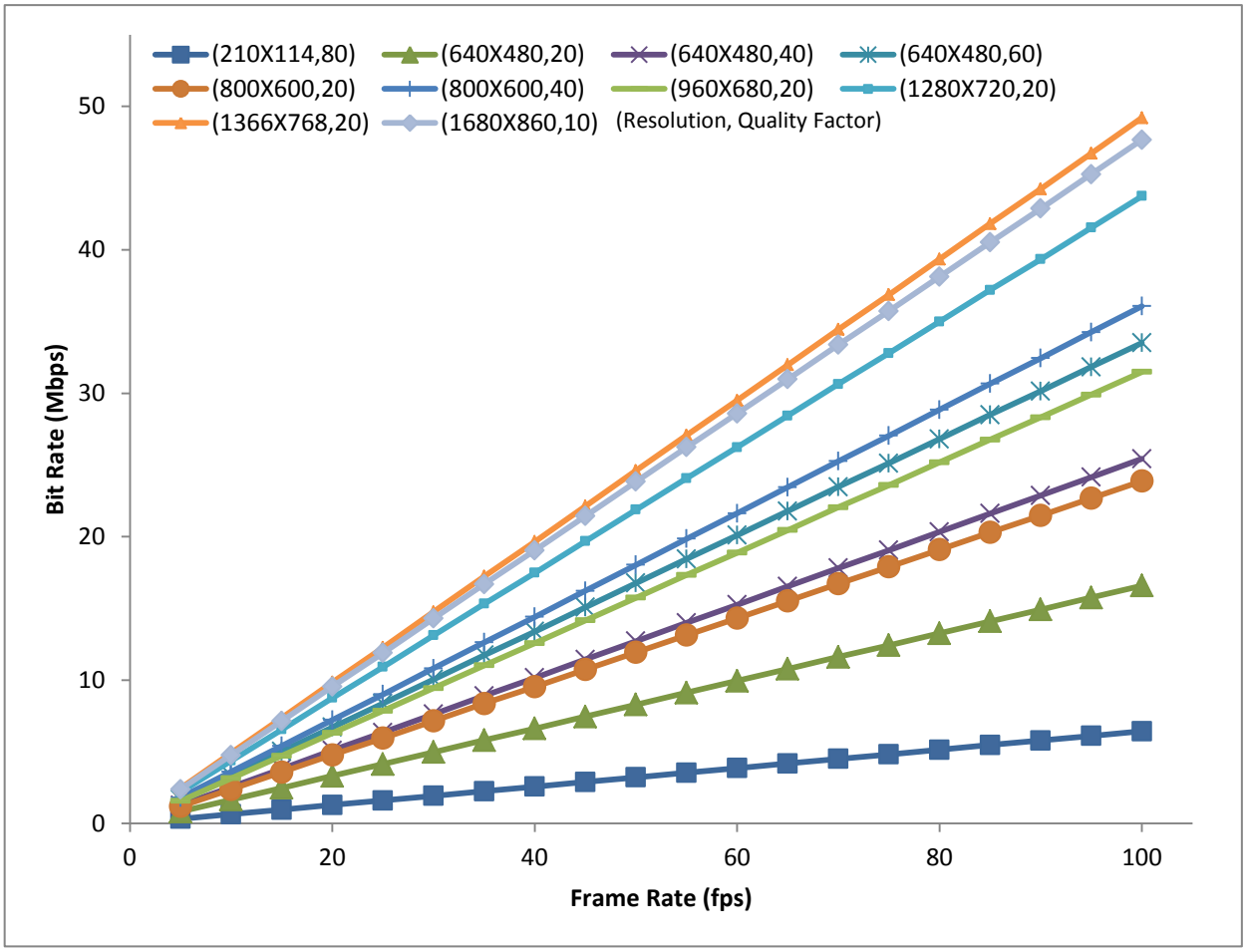

Figure 54. Network bitrate versus frame rate for different settings of Q and R (Car Tutorial)

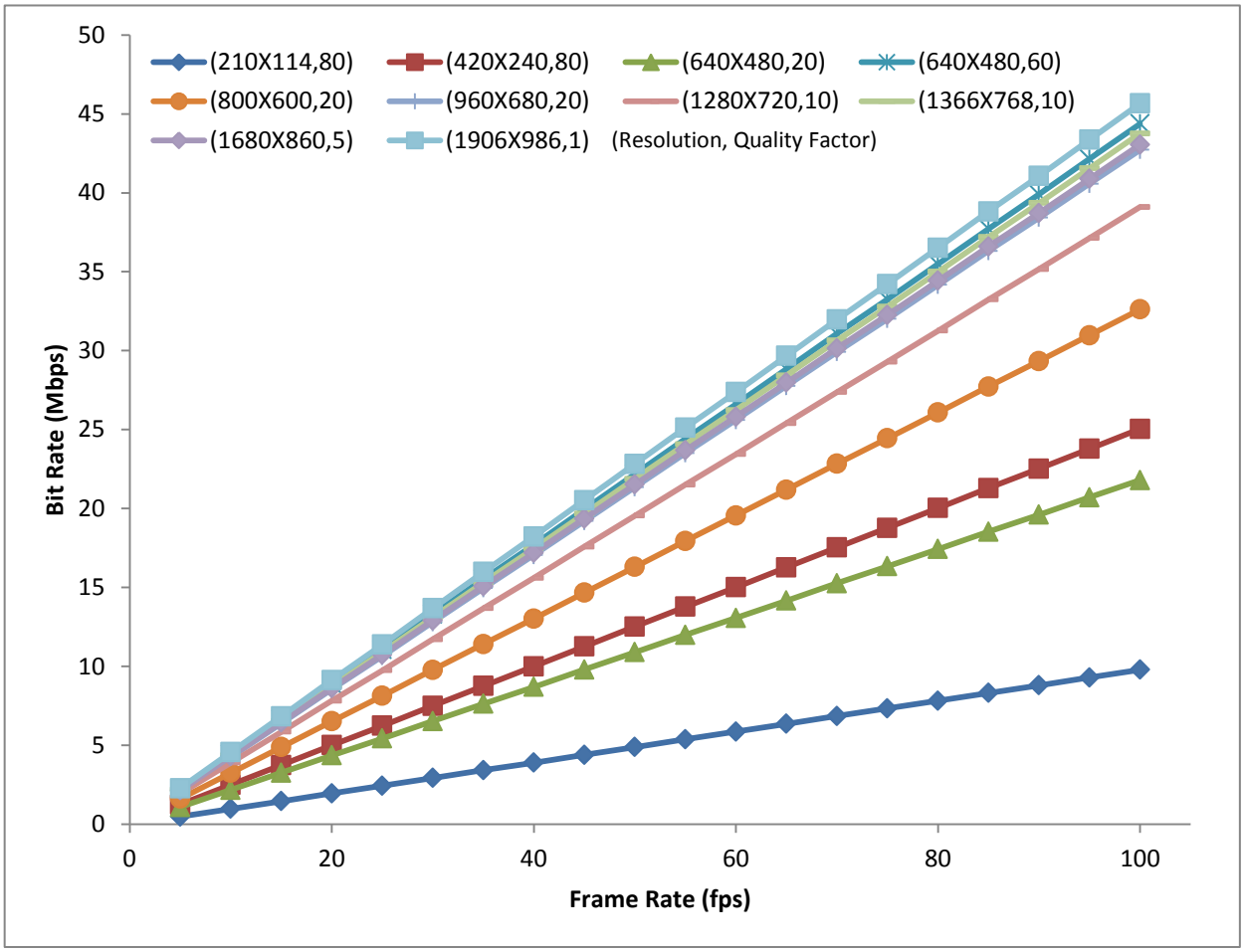

Figure 55. Network bitrate versus frame rate for different settings of Q and R (AngryBots)

Then we set up the Uniquitous and used the optimal settings recommended in previous discussions, where the resolution is 640×480 and the JPEG encoding quality factor is 20. AngryBots was run on the server when the game was being played remotely through the client for about five minutes. During the process, we used Wireshark **[38]** to capture the network packets sent between the Uniquitous server and the client. By filtering out the data packets and extracting out the bitrate data in Wireshark, we were able to draw the graphs in Figure 56 and 57. In the graph, the horizontal axis represents the time in seconds and the vertical axis indicates the network bitrate in Mbps or kbps. Figure 56 shows the downlink bitrate and Figure 57 shows the

uplink bitrate. Both graphs show the bitrate data over 1 minute during "steady state" in the middle of the game. From Figure 56, the downlink bitrate is fluctuating around 3.5 Mbps. From Figure 57, the uplink bitrate is fluctuating around 32 kbps, which is much smaller compared to the downlink traffic. The horizontal line in Figure 56 indicated the calculated bitrate of 3 Mbps from Figure 53 when the frame rate is 13.8 fps. The calculated network bitrate value is lower than all the actual bitrate values because the tested game was played differently. In the micro evaluation, the game character just followed the predefined path from one location to another location with little interaction with the game environment. However, when we were capturing the network packets, the game was played by a human being, which covered more locations and involved various interactions with the game environment like shootings and explosions.

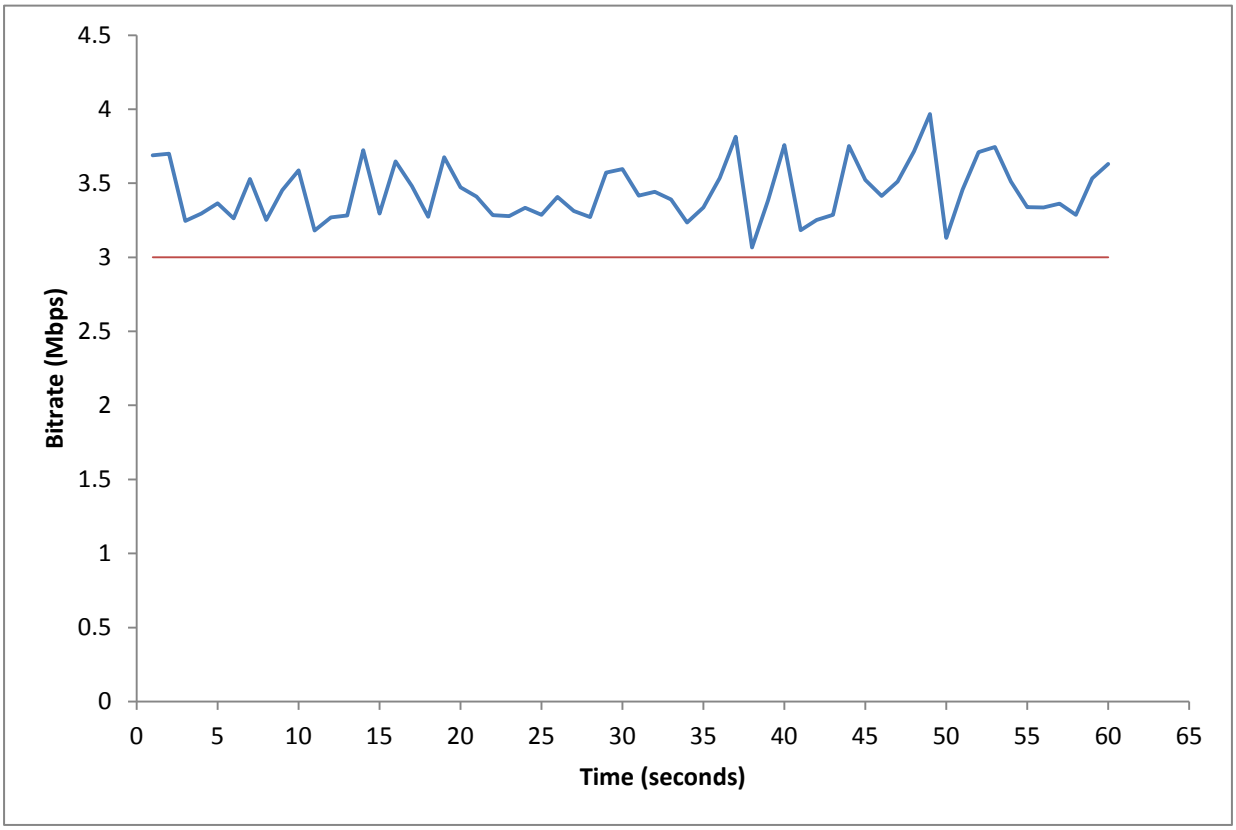

Figure 56. Downlink network bitrate versus time (AngryBots)

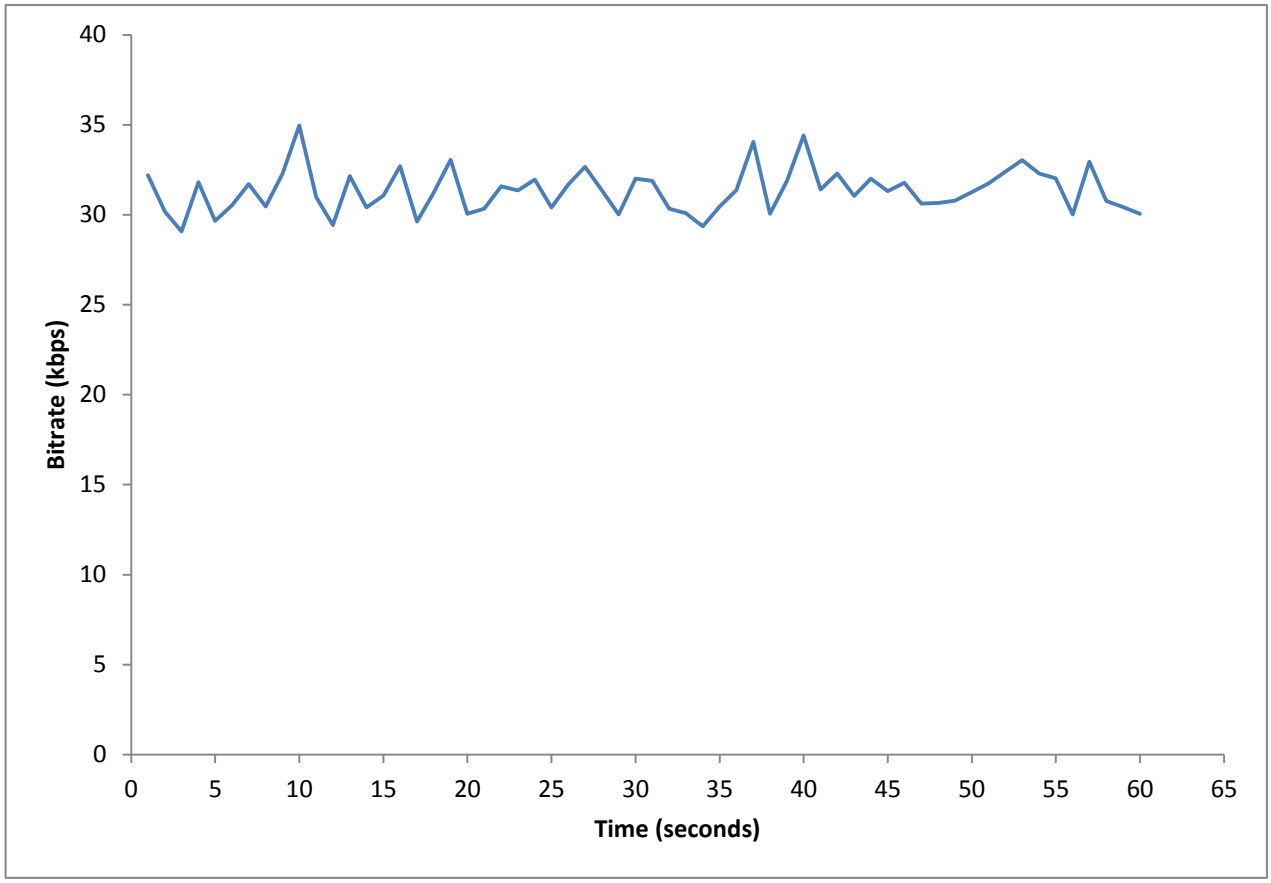

Figure 57. Uplink network bitrate versus time (AngryBots)

5. Macro Evaluation

This section covers analysis and evaluation of performance of Uniquitous in terms of the game quality and frame rate. In subsection 5.1, we conduct experiments to objectively measure the quality of images presented to players and discuss how game quality changes with different settings of the quality factor and resolution. In subsection 5.2, we conduct experiments to measure the frame rate on both the server and the client and describe how frame rate changes on the client under different settings of quality factor and resolution. In subsection 5.3, we discuss the work flow of the server and derive a model predicting the frame rate on the server. Then we built and validated a model for Uniquitous to predict the frame rate on the client based on the given resolution and quality factor settings.

5.1 Game Image Quality

Since we use the JPEG encoder to compress game frames before sending them to the client, there is some image quality loss for the frames displayed on the client. The visual quality of the game images is an essential factor impacting players' gaming experience **[19]**. So we set up experiments to objectively evaluate game image quality in Uniquitous under different system settings such as resolution and JPEG quality factor.

The Peak Signal Noise Ratio (PSNR) value gives an objective measure of compressed image quality. PSNR is defined in the formula below, where x,y represents the width and height of the image and A_{ij}, B_{ij} represent the pixel value at the same location in original image and compressed image, respectively. Mean Square Error (MSE) measures the difference between the

original image and compressed image, so a lower MSE indicates less loss of quality in the compressed image compared to the original image. Since MSE is inversely proportional to PSNR, the image quality is better if the PSNR value is higher.

$$PSNR(dB) = 10 * \log(\frac{255^2}{MSE})$$

$$MSE = \sum_{i=1}^{x}\sum_{j=1}^{y}\frac{\left(\left|A_{ij} - B_{ij}\right|\right)^2}{x * y}$$

We used the same static image captured from the Car Tutorial game which is shown in Figure 7 and displayed it throughout the experiments. For each resolution level, we saved the uncompressed image in PNG format. By changing the quality factor of JPEG encoder, we produced 20 images with different compression ratios. Repeating this for 10 levels of resolution, we ended up with 200 images. For each image sample, we computed the PSNR comparing the compressed image to the uncompressed versions. The tool used for PSNR calculation is ImageMagick **[33]**, which is an open source software suite for manipulating image files. With the PSNR results, we plotted a graph (shown in Figure 58) to depict how image resolution and compression ratio impacts the objective visual quality of images. The x axis represents the quality factor. The higher the quality factor is, the lower the compression ratio of the image. The data points of each trend line represent PSNR values for different JPEG quality factors at a specific resolution level. From the results, the maximum PSNR value is 25.5 when the resolution is 1366×707 and the quality factor is 30 and the minimum PSNR value is 21.5 when the resolution is 286×149 and the quality factor is 5. The PSNR values when game image is in different resolutions do not show a lot difference between each other since they are all between PSNR=20 and PSNR=26.

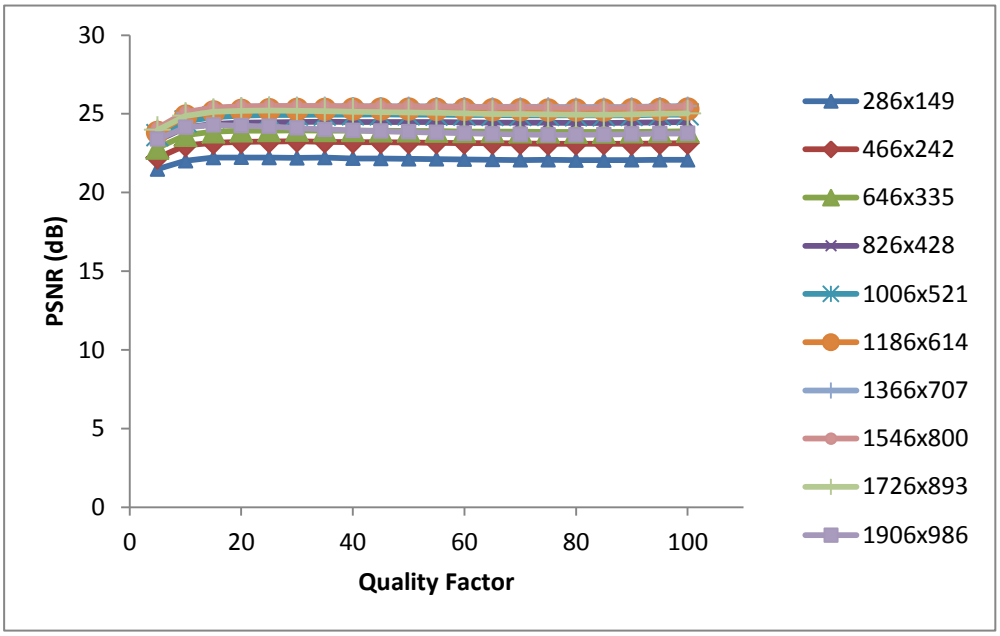

Figure 58. Comparison of PSNR values versus the quality factor among different game resolutions

In order to see the PSNR trend more clearly, we zoomed in the graph by setting the origin of the coordinate as (5, 21) and the new graph is shown in Figure 59. A higher quality factor means less compression for the image and typically a higher PSNR value. However, for most of lines in Figure 59, PSNR increases when the quality factor is increasing from 5 to around 25, decreases when the quality factor is increasing from 25 to 80, and then increases again when quality factor is increasing from 80 to 100. Particularly, for the largest resolution, the line drops down abruptly after quality factor 15 and escalates again after quality factor 75, but the highest PSNR value after that is still lower than the PSNR value at quality factor = 15. This result illustrates that there are other variables affecting the results.

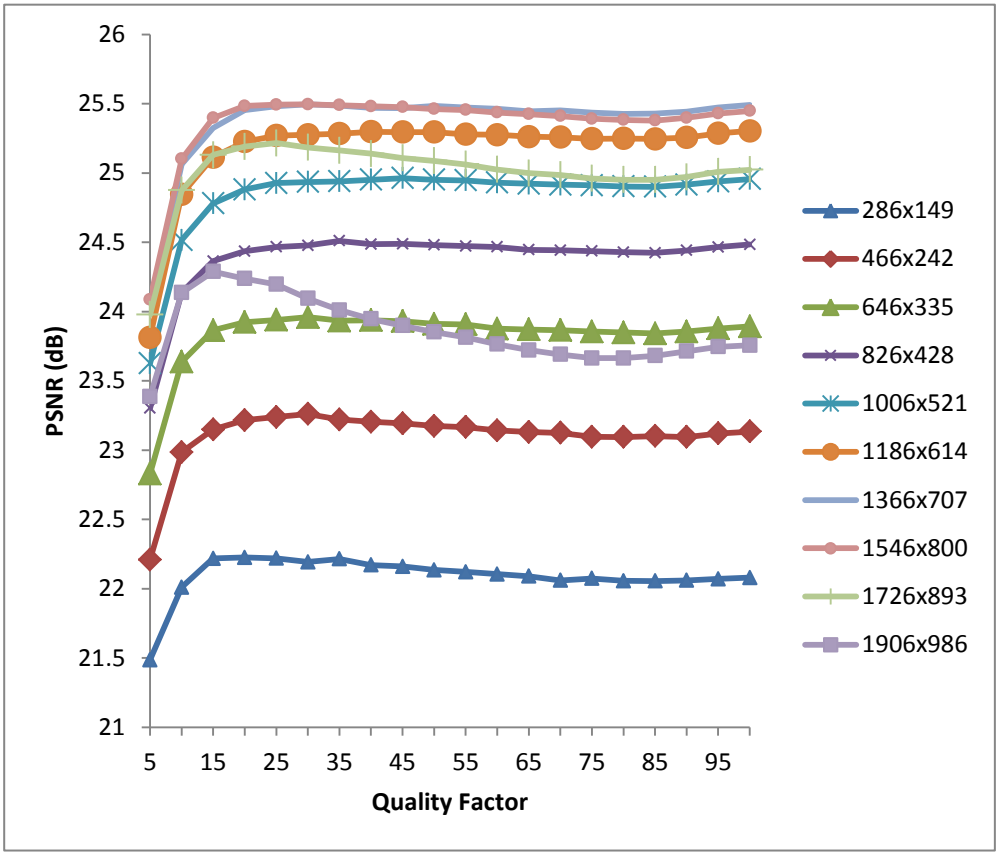

Figure 59. Comparison of PSNR values versus the quality factor among different game resolutions (zoomed in version)

We moved our focus on the tested image itself. The game image we tested is shown in Figure 7, which contains lots of complexity in the background like textures of the mountains and fences. We captured another four images from the game and saved them as uncompressed reference images. Then we tested them at resolution of 1906x986 because the data at this resolution presents the hardest to explain trend in Figure 59. From Figure 60 to Figure 63, the content varies among the four images mainly because they have different complexity in the game environment.

Figure 60. IMG1

Figure 61. IMG2

Figure 62. IMG3

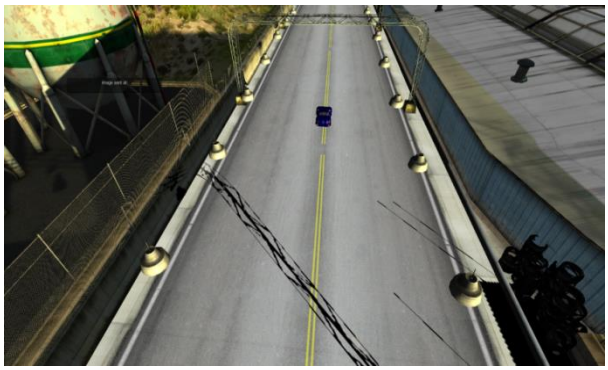

Figure 63. IMG4

We computed PSNR values at different JPEG quality factors for each image and found that the PSNR measurements varies for images containing different content. In Figure 64, lines of IMG2, IMG3 and IMG4 drop down after quality factor is 25 while the line of IMG1 gives the expected result based on the earlier PSNR definition. When the image has a resolution of 1906×986, PSNR measurement is most consistent for IMG1 which has less complexity and is least consistent for IMG3 which has the most complexity among the four tested images. Thus, the effect of PSNR depends on how complex the image is.

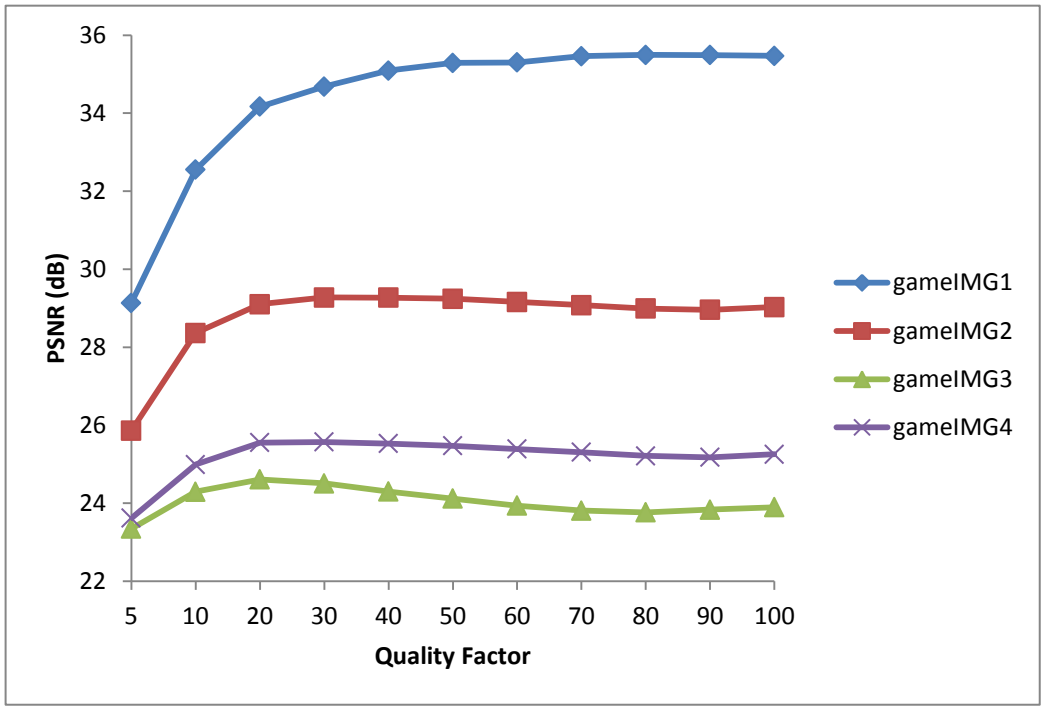

Figure 64. Comparison of PSNR values versus the quality factor among different game images

Considering the inconsistency of PSNR for different images, we used another popular measurement called Structural Similarity Index (SSIM) **[34]** to evaluate the visual quality of the compressed images. Instead of quantifying the visibility of the errors between a distorted image and a reference image as in PSNR, SSIM models image distortion as a combination of loss of correlation, luminance distortion and contrast distortion. To compute SSIM, we use a Matlab implementation provided by the SSIM Website **[35]**. We ran the SSIM code on all image samples based on the game image in Figure 7 to compute the SSIM index for each image at different compression ratios and resolutions. Then we plotted the graph in Figure 65 based on the results. The SSIM index ranges from 0 to 1. The higher the index, the higher the similarity between the compressed image and the reference image, which indicates the quality of the compressed image is better.

In Figure 65, the lines at all levels of resolutions have an increasing trend except the three smallest ones: 286×149, 466×242 and 646×335. For resolution of 286×149, the drop-down range between quality factor 70 and quality factor 100 is almost half of the interval distance between two successive points along y axis. If we consider this drop as a loss of precision on the results, then when it comes to resolutions of 466×242 and 646×335, the loss of precision decreases after quality factor 70. All lines show a marked increase in game image quality when the JPEG quality factor is below 25 and show a modest increase from 25 to 35. After 35, the increase in image quality is subtle. According to the micro evaluation results, higher quality factor requires more time to process each game frame which leads to a degradation of the frame rate. In our case, it is not significantly beneficial to increase game image quality by increasing the JPEG quality factor above 35, so we recommend a quality factor between 15 and 35 for Uniquitous.

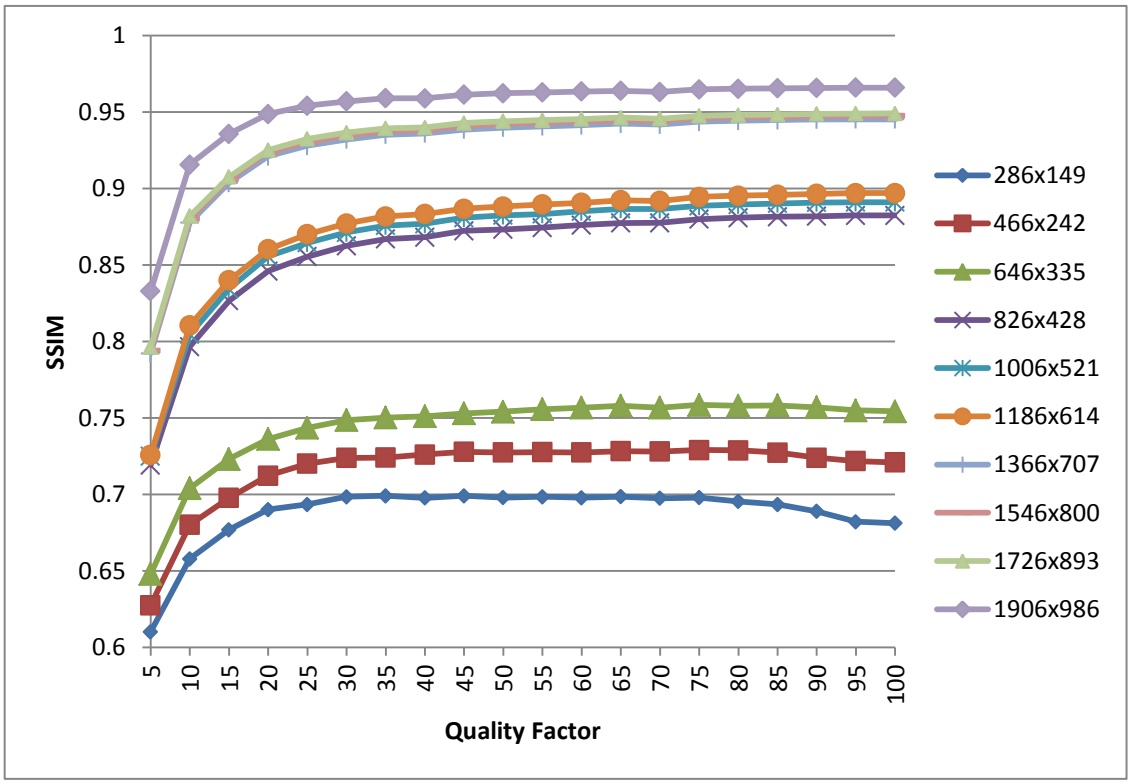

Figure 65. Comparison of SSIM values versus the JPEG quality factor among different game resolutions

5.2 Frame Rate

In the micro-evaluation, we used 72 combinations of settings of two parameters in Table 3 and Table 4 (8 different JPEG quality factors and 9 different game resolutions). For each game project, we have 72 data samples. Each data sample contains a different setting of quality factor and resolution. Due to the limit on the maximum data size of RPC, we only used data samples in which the data size after encoding is smaller than 65,000 bytes. For Car Tutorial, we picked out 44 data samples, which are shown in Figure 66. For AngryBots, we picked out 37 data samples, which are shown in Figure 67.

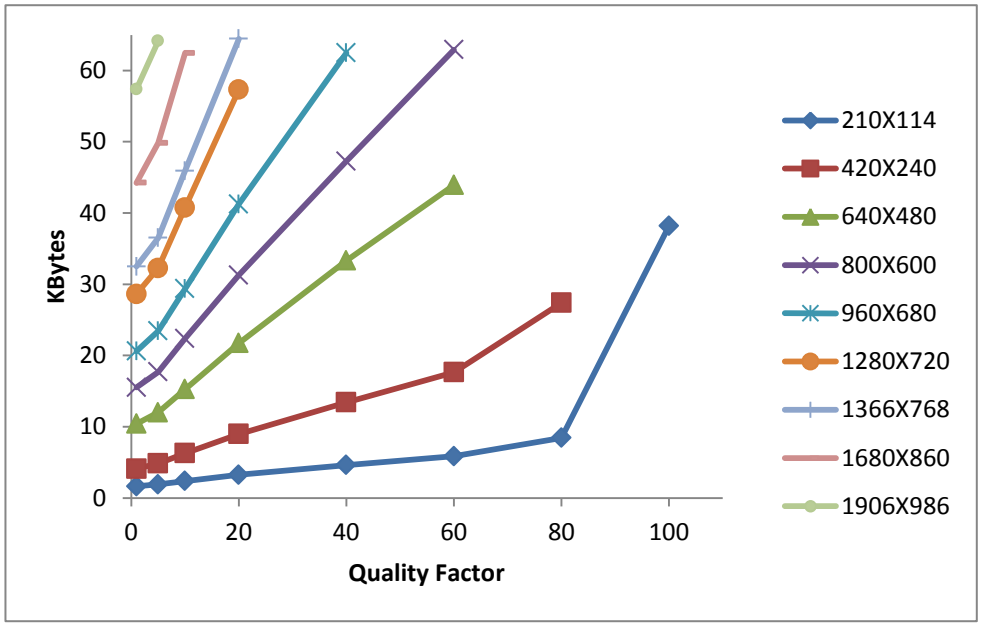

Figure 66. Frame sizes after encoding with different JPEG quality factors at different resolution levels in Car Tutorial

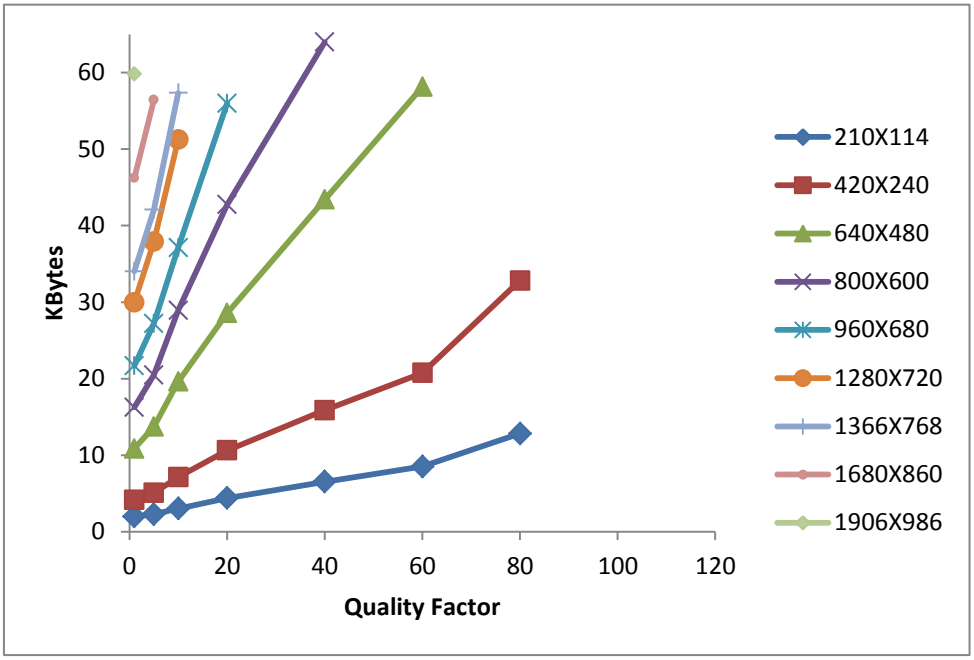

Figure 67. Frame sizes after encoding with different quality factors at different resolution levels in AngryBots

We set up Uniquitous server and client for both projects and measured the frame intervals for the parameters from each data sample. In the Uniquitous client, we put a timestamp before drawing the image on screen and recorded the time after decoding the received image and before displaying the image. We conducted experiments for each data sample. Each experiment starts with the car or player character moving from the same position and ends when the car or player character finished the predefined route. During this period, we recorded the time stamps. The difference between every two successive time stamps is the time interval between two successive frames displayed on the client. Then, we obtained time interval values for each data sample by computing the average value of these time intervals. In order to observe how the frame rate on the client changes with the quality factor at different resolutions, we obtained the frame rates by calculating the inverse of the time interval values we measured, shown in Figure 68 and Figure 69.

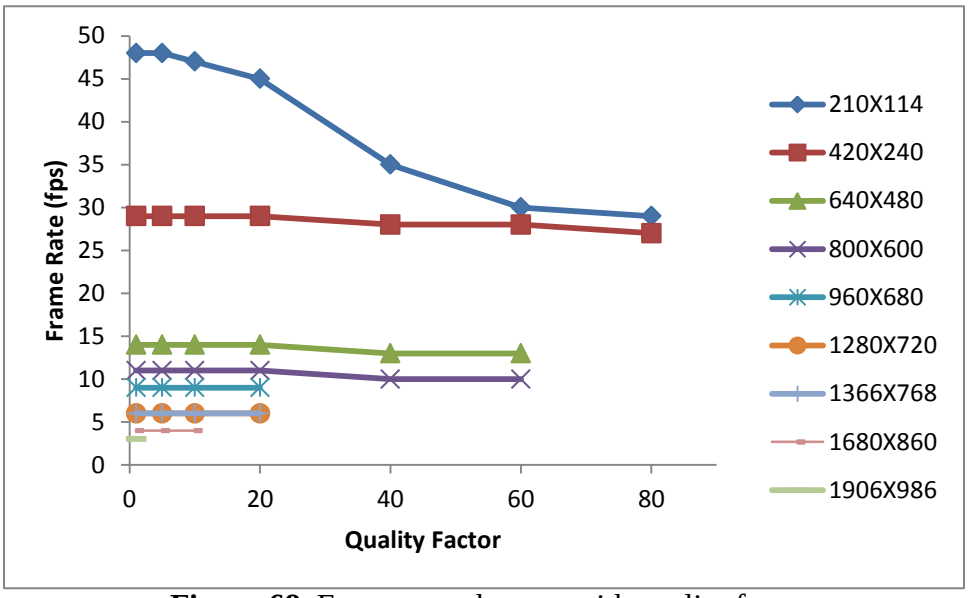

Figure 68. Frame rate changes with quality factor at different resolutions in Car Tutorial

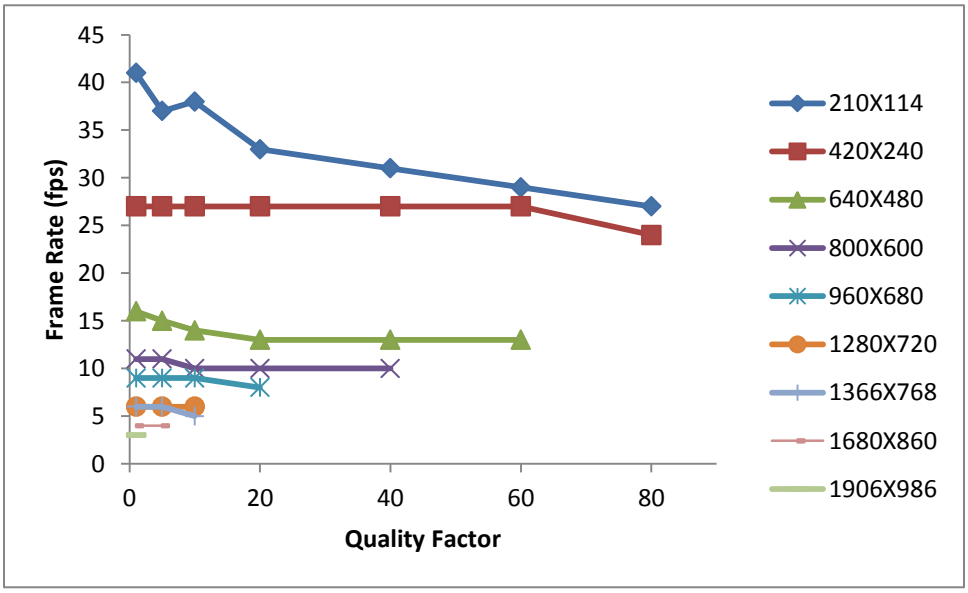

Figure 69. Frame rate changes with quality factor at different resolutions in AngryBots

From our experiment results in Figure 68 and Figure 69, Car Tutorial achieves the maximum frame rate of 48 fps when the resolution is 210×114 and the JPEG quality factor is 1, and the minimum frame rate of 3 fps when the resolution is 1906×986 and quality factor is 1. AngryBots achieves the maximum frame rate of 41 fps when the resolution is 210×114 and the

quality factor is 1 and the minimum frame rate of 3 fps when the resolution is 1906×986 and the quality factor is 1. Generally, the frame rate of Car Tutorial is higher compared to the frame rate of AngryBots.

Claypool et al. **[20]** found that user performance shows a marked drop when frame rates are below 15 fps and shows a modest increase from 15 fps to 30 fps. So we recommend 15 fps as an indication of minimum acceptable performance of Uniquitous in terms of the frame rate. According to Figure 68 and Figure 69, both game projects can approximately achieve 15 fps within the recommended quality factor range of 15 – 35 when the game window is 640×480. Therefore, the recommended resolutions on Uniquitous to achieve an acceptable or higher game frame rate should be no larger than 640×480 pixels.

For illustration, we only show how frame rate changes with different resolutions when the quality factor is 10 for each project in Figure 70 and Figure 71. The graphs show that the frame rate is decreasing with resolution increasing. The lines from Figure 68 and Figure 71 indicate that both increasing game resolution and increasing quality factor degrade the game frame rate on Uniquitous.

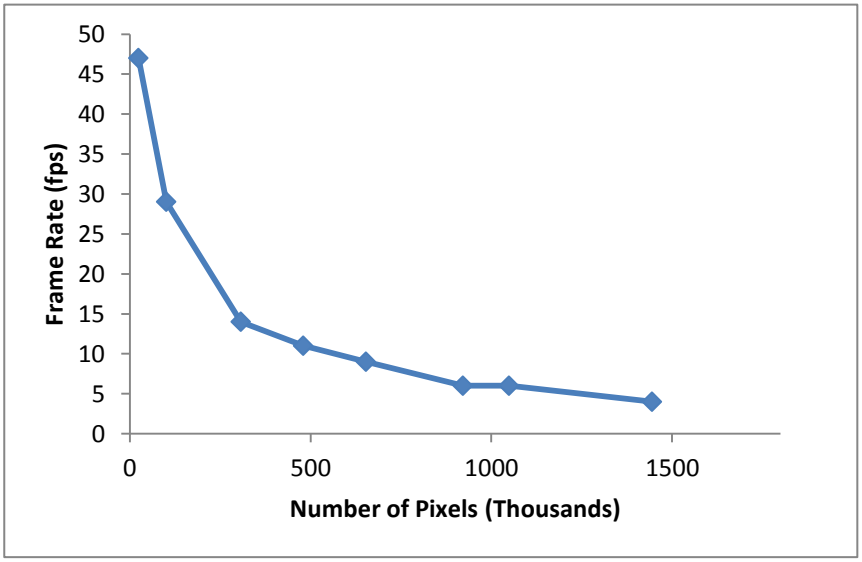

Figure 70. Frame rate versus resolution when the quality factor is 10 in Car Tutorial

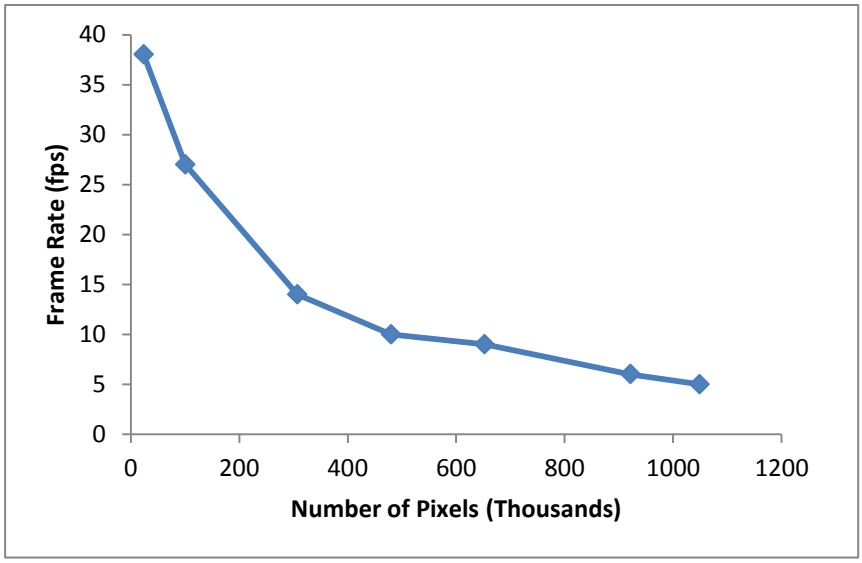

Figure 71. Frame rate versus resolution when the quality factor is 10 in AngryBots

In each experiment, we also put a time stamp before the Image Transmission is executed and recorded the time before each compressed game frame is ready to be sent out on the server. The amount of time it takes to send out each frame is the interval between every two successive time stamps and the average value of these intervals is the frame interval on the server. Then we obtained the frame rates on the server by calculating the inverse of the frame interval values we measured. In Figure 72 and Figure 73, we compare the server frame rate with the client frame rate under the condition of each data sample. The average error between server and client frame rate for each project is 0.1. The figures indicate that the server frame rate is approximately equivalent to the client frame rate.

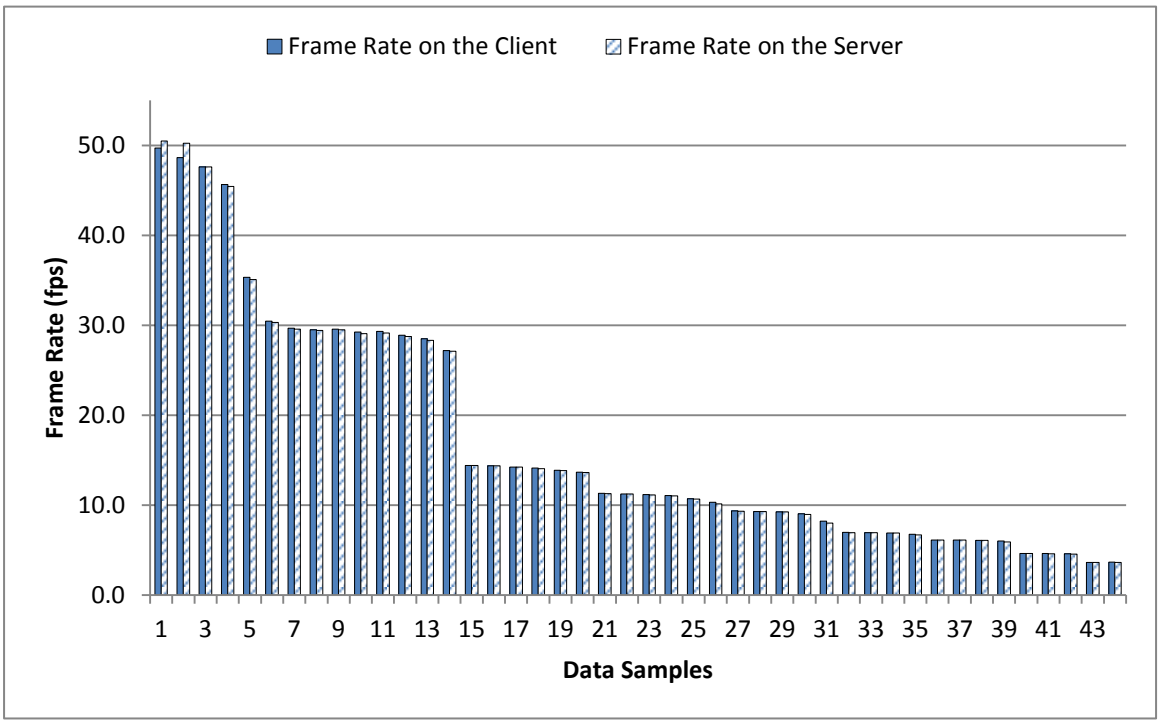

Figure 72. Comparisons between the server frame rate and the client frame rate in Car Tutorial

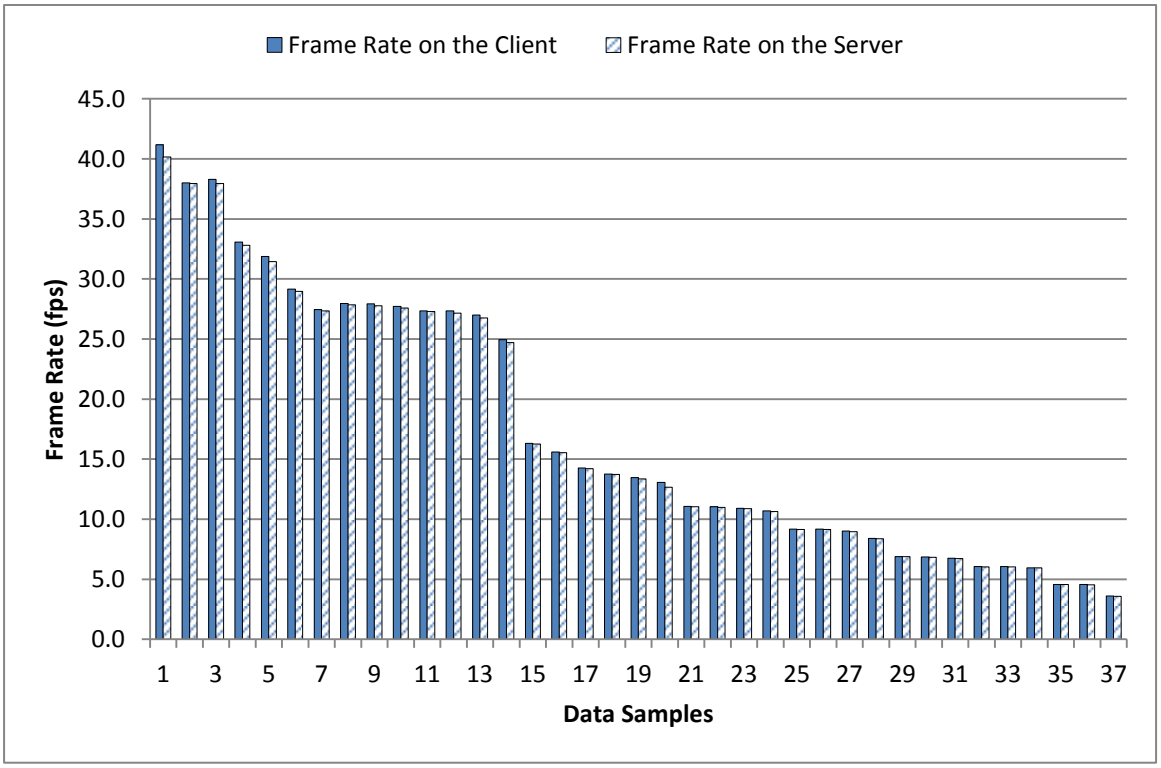

Figure 73. Comparisons between the server frame rate and the client frame rate in AngryBots

5.3 Predicting Frame Rate

Components in the Uniquitous server can be classified into four groups as illustrated in Figure 74. Each of the four groups works in different thread: All the components in group 1 are running in the Unity main thread; Group 2 includes only Image Encoding since the `JPEGEncoder` is running in a separate thread; All components in Group 3 are running in a separate thread in the Unity audio engine; Group 4 includes the audio compression and transmission components running in independent processes started by FFMPEG application. Group 3 and Group 4 run in parallel with each other. Either Group 3 or Group 4 runs in parallel with Group 1 and Group 2. Group 1 and Group 2 run partially in parallel with each other.

While Group 1 and Group 2 components are dealing with the game image, Group 3 and Group 4 components are dealing with the game audio. In order to analyze the frame rate on the Uniquitous server, we explain in detail the work flow between Group 1 and Group 2. In Group 1, only Input Reception, Unity Game and Game Window work simultaneously with JPEG Encoding (Group 2). According to the diagram in Figure 74, Input Reception, Unity Game and Game Window are running on each frame. Screen Capture is executed once and is not executed until both JPEG Encoding and Image Transmission are done. It is similar for Image Transmission, which is only executed after JPEG Encoding is done. Screen Capture and Image Transmission can be blocked by JPEG Encoding because the three are in the same coroutine. The coroutine can pause its execution and return control to Unity, continuing where it left off on the following frame to check if JPEG Encoding is done. Coroutine updates are about every 20 milliseconds when Unity achieves a frame rate of 50 fps.

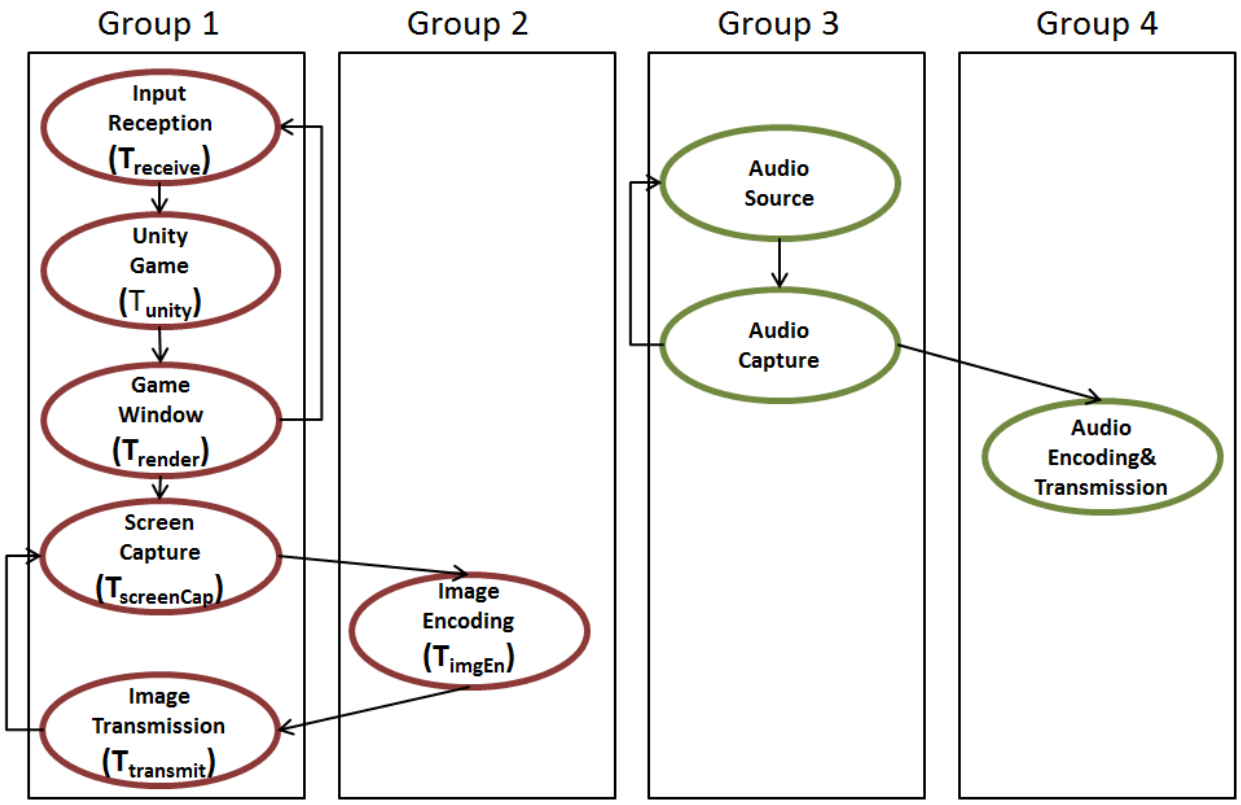

Figure 74. Parallel working structure of Uniquitous server

Assuming sufficient processing on the client, the frame rate of the game on the thin client depends on how often the client receives the next available frame from the server. This in turn, depends on how often the server sends out the next available frame. According to Figure 74, the first three components of Group 1 (Input Reception, Unity Game and Game Window) and Group 2 work in parallel and the processing time on each of the two affects how fast a game frame can be provided by the server. So the frame rate of the game on the thin client is subject to the processing time of either the first three components of Group 1 or Group 2, whichever takes the longest. Based on our analysis, the derived formulas can be used to calculate the frame rate:

$$F = 1/T$$

$$T = \text{Max}\,(T_{1'}, T_2) + T_{screenCap} + T_{transmit}$$

$$T_{1'} = T_{unity} + T_{render}$$

$$T_2 = T_{imgEn}$$

Where $T_{1'}$ is the processing time of the first three components of Group 1. T_2 is the processing times of Group 2. T is the frame interval, which is the sum of $T_{screenCap}$, $T_{transmit}$ and the maximum number between $T_{1'}$ and T_2. By calculating the inverse of T we can obtain the frame rate which is represented by F. $T_{receive}$ is ignored because the time for receiving input is too small compared to the processing time of other components.

Similar to subsection 5.2, we selected 44 data samples for Car Tutorial, which are shown in Figure 68 and selected 37 data samples for AngryBots, which are shown in Figure 69. With the selected data samples, we computed the predicted frame rates (F) for both the Car Tutorial and AngryBots based on our formula.

Using the server frame rate we measured in subsection 5.2 and the predicted frame rate (F) we calculated from our formula, the results are plotted in Figure 75 and Figure 76. For Car Tutorial, the predicted and measured frame rate have a correlation coefficient of 0.948 and the average error percentage is 25.6%. For AngryBots, the predicted and measured frame rate have a correlation coefficient of 0.987 and the average error percentage is 28.1%. Figure 75, 76 show that the trend of the predicted frame rate matches the trend of the measured frame rate trend. The error between the measured frame rate and the predicted frame rate is introduced by the coroutine. In Figure 74, the coroutine allows the Unity main thread to check if JPEG Encoding is done every 20 milliseconds and when it is done, Unity executes Image Transmission. Sometimes JPEG Encoding finishes just after the last check, so it takes 20 more milliseconds to know that

JPEG Encoding is done and sends out the frame even though JPEG Encoding has been done for 20 milliseconds. Therefore, we can have an extra processing delay no larger than 20 milliseconds compared to the predicted frame interval. The blue data points in Figure 75 and 76 show that most measured frame rates are lower than the predicted ones because the predicted frame rates are computed based on the assumption that there is no such an extra delay occurred. So we add a condition to the formulas for calculating the frame rate:

If $T_2 = \text{Max}(T_{1'}, T_2)$,

Then $T = \text{Max}(T_{1'}, T_2) + T_{screenCap} + T_{transmit} + T_{error}$

($T_{error} \in [0, 20]$),

Based on the updated formula, we plotted the data points with the error term in Figure 75 and 76. For Car Tutorial, the average error percentage is reduced from 25.6% to 11.8%. For AngryBots, the average error percentage is reduced from 28.1% to 12.5%. For both game projects, the predicted frame rates at resolutions higher than 210×114 approximate to the actual frame rates. According to the data points scattered on top right corner of each graph, the predicted frame rates at resolution of 210×114 do not change a lot based on the updated formula. The errors of these data points are not reduced because T_2 is larger than $T_{1'}$ at resolution of 210×114 and the processing time measurement for the Unity Project and the Game Window is not accurate enough.

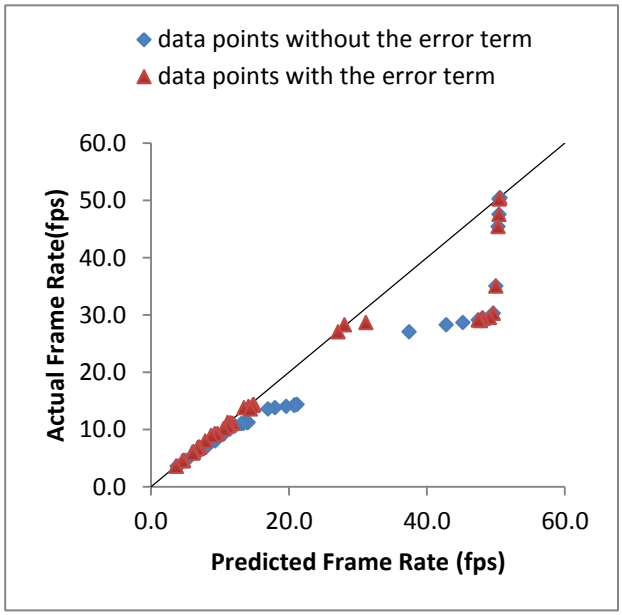

Figure 75. The scatter plot of measured and predicted frame rate for Car Tutorial

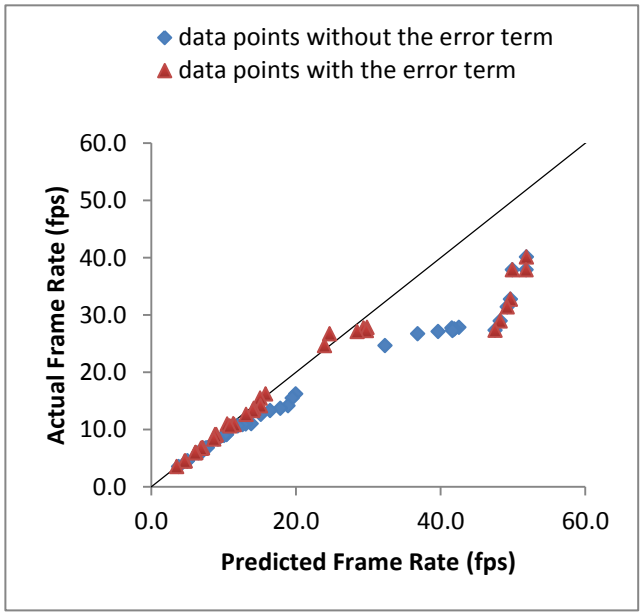

Figure 76. The scatter plot of measured and predicted frame rate for AngryBots

There are four variables affecting players cloud gaming experience: game, image quality, resolution and frame rate. Based on conclusions from **[19, 20]**, frame rate is the most crucial factor affecting users' experience. So with the time interval values we measured, we also built a frame rate predicting model for Uniquitous which can be helpful for users to choose a right settings of quality factor and resolution for desired frame rate.

Subcomponents	Parameters Affecting Its Processing Time
Unity Project	none
Game Window(rendering)	none
Screen Capture	game resolution
Image Transmission	game resolution, quality factor
Image Encoding	game resolution, quality factor

Table 6. Subcomponents of Group 1 and Group 2 and parameters affecting its processing time

According to our derived formulas for predicting the frame rate, the frame rate depends on processing times of the components in Group 1 and Group 2. Based on the information provided

by the Table 6, the processing times of most subcomponents in Group 1 and Group 2 depend on the settings of game resolution and quality factor. Hence, our model's input data are the game resolution and the quality factor and the model's output is the game frame rate on the client. We did not have direct measurement of image transmission time based on game resolution and quality factor. We have recorded the image data size after encoding for each of the 72 combinations of parameter settings. By using the results in Figure 48, which shows the relationship between the transmission time and the data size, we mapped the relationship between the image transmission time and the game resolutions and JPEG quality factors.

With our data samples and corresponding time interval values measured for each data sample, we used Weka LinearRegression classifier with 10-fold cross validation **[36]** to make a linear regression model for each project predicting the frame rate:

1. $F_{predict} = 1 / (0.1348 \times R + 0.118 \times Q + 21.0)$ (Car Tutorial)
2. $F_{predict} = 1/ (0.1361 \times R + 0.1224 \times Q + 22.5)$ (AngryBots)

There are three attributes in formulas above. $F_{predict}$ is the frame rate. R represents the resolution which is the number of pixels divided by 1000. Q is the numeric value of quality factor.

In order to validate our model, we chose a group of new R, Q values we have not tested from Table 3 and Table 4 to test. Under the constraint of data transfer maximum size for RPC we chose 35 pairs of R and Q for Car Tutorial and 30 pairs of R and Q for AngryBots. By using the prediction formulas, we computed $F_{predict}$ for each pair of R and Q for each game. Based on the actual frame rate we measured by running the games, the validation results are plotted in Figure

77 and Figure 78. For Car Tutorial, the actual and predicted frame rate have a correlation coefficient of 0.995 and the average error percentage is 4.79%. For AngryBots, the actual and predicted frame rate have a correlation coefficient of 0.981 and the average error percentage is 9.47%. These figures clearly indicate that the models made for each game generally work well for predicting the frame rate and works better predicting the frame rate lower than 20 fps than a frame rate over 25 fps.

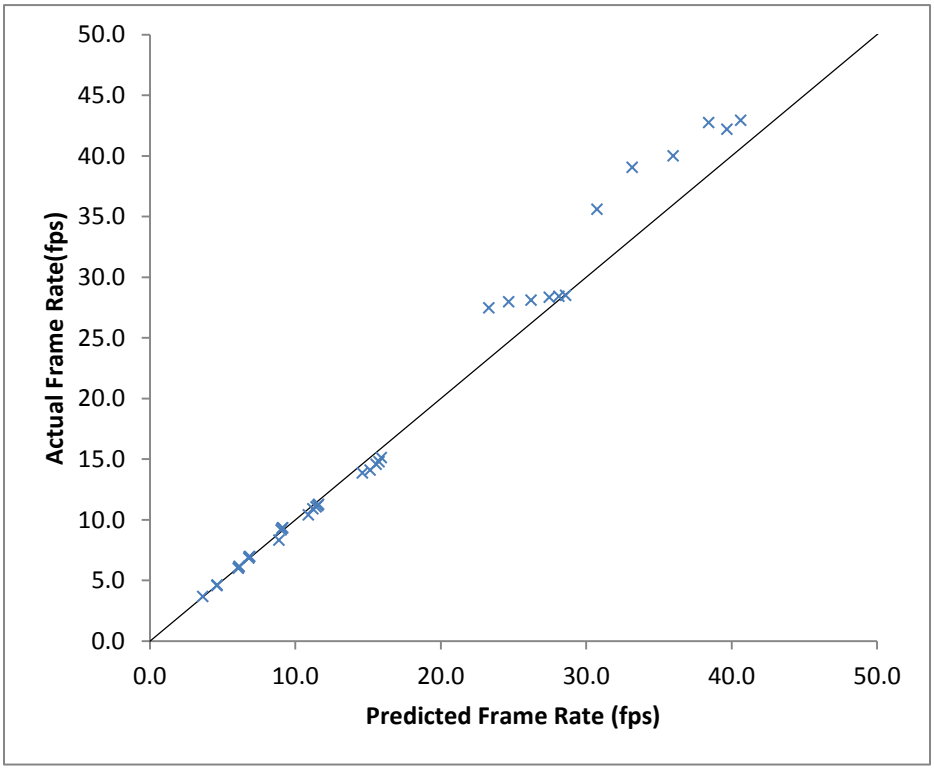

Figure 77. Actual versus predicted frame rate for Car Tutorial

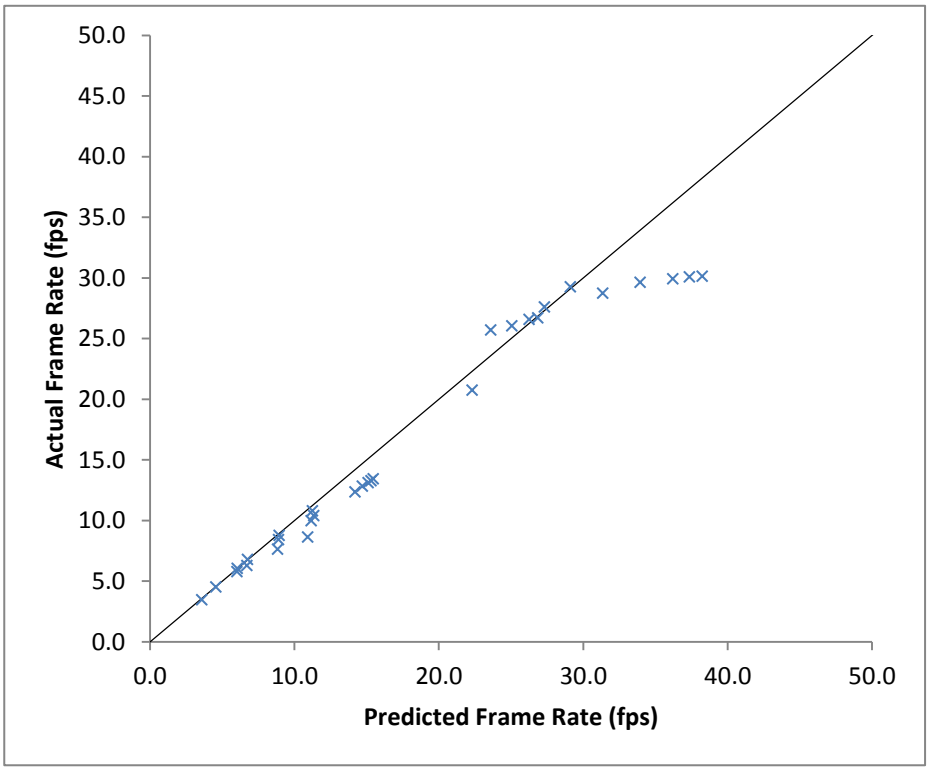

Figure 78. Actual versus predicted frame rate for AngryBots

6. Conclusion and Future Work

Due to proprietary technologies and implementation of commercially available cloud gaming systems such as OnLive, it is difficult for scientists to use them as testbeds to explore cloud game research or for game developers to use them as platforms to create and deploy cloud games. Academic systems, such as GamingAnywhere, allow access to system source code and give the flexibility to modify the system structure and configure parameters, but flexibility is limited as it is difficult to access the game source code or the game content.

In this thesis, we implement an open source cloud gaming system in Unity, called *Uniquitous*. Since Uniquitous blends seamlessly with Unity game development, it provides platform controls not only over the game system but also over the game content in a cloud-based environment. The Uniquitous system is composed of three entities: Unity Project, Uniquitous Server and Uniquitous Thin Client. The Unity Project runs on the Uniquitous server in Unity as a first machine and the Uniquitous Thin Client runs in Unity a second machine. The communications among the components are via three data flows: the image data flow, the audio data flow and the input data flow.

We performed micro evaluation and macro evaluation with two games in Uniquitous in order to understand the performance bottlenecks of cloud gaming systems and predict the performance of Uniquitous in alternative environments. The experiments in the micro and macro evaluation were conducted under different settings of JPEG encoding factor and image resolution. In the micro evaluation, we put time stamps in different places in the system source code to measure processing times for most components on the Uniquitous server. For the Audio Encoding&Transmission component, Unix command “time” was used to measure processing time. In the macro evaluation, we used both PSNR and SSIM to objectively measure game image

quality. In order to evaluate the frame rate, we put time stamps in the system source code to measure the processing time for each frame on both the server and the client and converted the measured times into the frame rates.

In the micro evaluation, for both games, we found that changing the game resolution does not affect the processing time of the Unity Project and Game Window. The processing time of the Screen Capture, Image Transmission and Image Encoding is proportional to the game resolution. Moreover, the processing time of the Image Transmission and Image Encoding is proportional to the game quality. The Unity Project is the most time consuming component on the server when the game quality and game resolution are both low, but with increasing the game quality and game resolution, the Image Encoding becomes the most time consuming component on the server. The evaluation and analysis on all the components on the server are important because their processing times affect the total processing time for each frame sent out by the server and thus affect the frame rate of Uniquitous games on the client.

Based on the conclusion that the frame rate plays a more important role in affecting players' experience compared to the resolution and game quality **[19, 20]**, we explored how the game quality and resolution affect the frame rate. In the macro evaluation, we found that for all resolution levels tested, game quality shows a marked increase when the JPEG image encoder quality factor is between 1 and 15 and shows a modest increase between 15 and 35. After 35, the improvement of the game quality is subtle. According to our model predicting the frame rate on the server, increasing the quality factor decreases the frame rate, so we recommend an image encoder quality factor between 15 and 35 be used in order to maintain a good frame rate. We also found that for all JPEG quality factors tested, resolutions below 640×480 provide frame rates higher than 14 fps and resolutions above 640×480 provide frame rates lower than 11 fps.

Since increasing the resolution leads to the degradation of the frame rate and the acceptable frame rate for players is 15 fps **[20]**, we recommended a resolution no larger than 640×480 be used in order to achieve a frame rate of 15 fps or higher. Based on the analysis of the experimental results, we conclude that the frame rate is inversely proportional to both the game quality and the resolution.

For future work, for improved performance, the Screen Capture component can be taken out of the corotine to work in parallel with the Image Encoding component. This should improve the efficiency of the Image Encoding component since it will get each frame from the Screen Capture component, increasing the achieved frame rate. Also, instead of sending one frame per RPC, the frames can be broken into several chunks of data and sent via RPC in sequence. On the client side, the frame can be recovered by merging the data chunks. In this way, higher game quality and higher resolutions that result in the size of each frame being larger than 65,000 bytes can be used.

There two areas recommended for exploration with the Uniquitous. The first area is that by conducting micro evaluation of Uniquitous for more games from the three genres: avatar-first-person, avatar-third-person and omnipresent **[20, 21]**, we can evaluate each game genre and build a relationship between the game genre and the frame rate. Then, we can redefine the model for predicting the frame rate for Uniquitous with the game genre, quality and resolution as the inputs. Since the Unity iOS/Android networking engine is fully compatible with desktop devices, the second area of future work is to extend and deploy Uniquitous on mobile devices to evaluate performance between two mobile devices or between a mobile device and a desktop device.

References

[1] Distribution and monetization strategies to increase revenues from Cloud Gaming: http://www.cgconfusa.com/report/documents/Content-5minCloudGamingReportHighlights.pdf, Date accessed: 05/01/2014

[2] Gaikai: https://www.gaikai.com/, Date accessed: 05/01/2014

[3] Sony buys Gaikai cloud gaming service for $380 million http://www.engadget.com/2012/07/02/sony-buys-gaikai/, by Sharif Sakr on July 2nd 2012, at 2:55:00 am ET

[4] CES 2014: Gaikai becomes PlayStation Now, streaming games to just about everything http://www.extremetech.com/gaming/174236-ces-2014-gaikai-becomes-playstation-now-streaming-games-to-just-about-everything ,by James Plafke on January 7, 2014 at 3:09 pm

[5] Minimum system requirement for OnLive: https://support.onlive.com/hc/en-us/articles/201229050-Computer-and-Internet-Requirements-for-PC-Mac-, Date accessed: 11/06/2014

[6] Onlive Games: http://games.onlive.com/ Date accessed: 05/01/2014

[7] StreamMyGame: https://streammygame.com/ Date accessed: 05/01/2014

[8] C. Huang, C. Hsu, Y. Chang, and K. Chen. "Gaminganywhere: An open cloud gaming System," *in Proceedings of the ACM Multimedia Systems Conference (MMSys'13)*, Oslo, Norway, February 2013.

[9] Unity3d official website: http://unity3d.com/, Date accessed: 11/06/2014

[10] Unity Fast Facts: http://unity3d.com/company/public-relations, Date accessed: 05/01/2014

[11] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," *In Proceedings of Workshop on Grid Computing Environments (GCE)*, pp. 1, January 2009.

[12] What is cloud computing: http://searchcloudcomputing.techtarget.com/definition/cloud-computing Date accessed: 11/06/2014

[13] R. Shea, J. Liu, E. Ngai, and Y. Cui. "Cloud Gaming: Architecture and Performance," IEEE Network Magazine, 27(4):16–21, July/August 2013.

[14] 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.

[15] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, "An Evaluation of QoE in Cloud Gaming Based on Subjective Tests," in *Inovative Mobile and Internet Service in Ubiquitous Computing (IMIS)*, Seoul, Korea, June 2011.

[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 applica-tions". *In Proc. of ACM NOSSDAV* 2006, Newport, RI, May 2006.

[17] O. Holthe, O. Mogstad, and L. Ronningen. "Geelix LiveGames: Remote playing of video games". *In Proc. Of IEEE Consumer Communications and Networking Conference (CCNC'09)*, Las Vegas, NV, January 2009.

[18] J. Kay and J. Pasquale, "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000," *Proc. 1993 Winter Usenix Conference, pp. 249–258*, January 1993.

[19] Yu-Chun Chang, Po-Han Tseng, Kuan-Ta Chen, and Chin-Laung Lei." Understanding the Performance of Thin-Client Gaming", in Proceedings of *IEEE CQR*, May 2011.

[20] Mark Claypool and Kajal Claypool. "Perspectives, Frame Rates and Resolutions: It's all in the Game", in Proceedings of the *4th ACM International Conference on the Foundations of Digital Games (FDG)*, Florida, USA, April 2009.

[21] M. Claypool and K. T. Claypool, "Latency and player actions in online games," *Communications of The ACM, vol. 49, pp. 40–45*, 2006.

[22] M. Claypool and K. Claypool, "Latency can kill: precision and deadline in online games," in *Multimedia Systems Conference (MMSys)*, Phoenix, Arizona, USA, February 2010.

[23] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei, "Measuring the latency of cloud gaming systems," in *Proceedings of ACM Multimedia*, Scottsdale, AZ, USA, Nov 2011.

[24] JPEG Encoder Source for Unity in C#:
https://code.google.com/p/unity-jpeg- encoder/source/browse/tags/Version1/UnityProject/Assets/?r=21, Date accessed: 05/06/2014

[25] ULink: http://developer.muchdifferent.com/unitypark/uLink/uLink, Date accessed: 10/06/2014

[26] Audio buffer size in Unity: http://docs.unity3d.com/ScriptReference/AudioSettings.SetDSPBufferSize.html, Date accessed: 10/6/2014

[27] FFMPEG Project: http://www.ffmpeg.org/about.html, Date accessed: 05/01/2014

[28] Simple Directmedia Layer: http://www.libsdl.org/, Date accessed: 05/01/2014

[29] Unity Car Tutorial Project:
http://u3d.as/content/unity-technologies/car-tutorial/1qU, Date accessed: 11/06/2014

[30] Unity AngryBots Project:
http://u3d.as/content/unity-technologies/angry-bots/5CF, Date accessed: 11/06/2014

[31] Unity Pro Profiler: http://docs.unity3d.com/Manual/Profiler.html, Date accessed: 10/06/2014

[32] Image sample source: http://img1.mxstatic.com/wallpapers/3cdd865891a11dd74fdf49e4b71dd119_large.jpeg Date accessed: 11/18/2014

[33] ImageMagick: http://www.imagemagick.org/, Date accessed: 10/06/2014

[34] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error measurement to structural similarity", *IEEE Trans. Image Process., vol. 13, no. 4, pp. 600–612*, Apr. 2004.

[35] SSIM official website: https://ece.uwaterloo.ca/~z70wang/research/ssim/, Date accessed: 10/6/2014

[36] Weka Linear Regression Classifier:
http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/LinearRegression.html
Date Accessed: 11/18/2014

[37] M. Claypool, D. Finkel, A. Grant and M. Solano, "On the performance of OnLive Thin Client Games", in *Multimedia System Journal (MMSJ) – Network and Systems Suppport for Games*, Denver, Colorado, USA, Octorber 2013.

[38] Wireshark: www.wireshark.org